UNIVERSITE MOHAMED PREMIER
Faculté Pluridisciplinaire
Nador

Support de Cours

Bases de données avancées
Langage SQL & PL /SQL sous Oracle

Filière : SMI-S6

Pr. Hakima ASAIDI
Département Mathématiques et Informatique

Année universitaire 2014-2015

Sommaire

Partie I : Langage SQL

Introduction

I. SQL langage de définition de données
- I.1 Création de table
- I.2 Définition des contraintes d'intégrité
- I.3 Modification de la structure d'une table
- I.4 Consultation de la structure d'une base
- I.5 Destruction de table

II. SQL langage de manipulation de données
- II.1 Insertion de données
- II.2 Modification de données
- II.3 Suppression de données
- II.4 Interrogation
  - II.4.1 Les clauses de SELECT
  - II.4.2 Gestion des transactions

III. SQL langage de contrôle des données
- III.1 Création d'index
- III.2 Création et utilisation des vues
- III.3 Protection des données : contrôles d'accès
  - III.3.1 Création et suppression d'utilisateurs
  - III.3.2 Création, attribution et suppression de privilèges et rôles

Partie II : Langage PL/SQL

Introduction

I. Le bloc PL/SQL
II. Types des données
III. Structures de contrôle
IV. Curseurs
V. Exceptions
VI. Procédures et fonctions

Introduction

SQL (sigle de *Structured Query Language*, en français **langage de requête structurée**) est un langage informatique normalisé servant à exploiter des bases de données relationnelles. La partie *langage de définition des données* permet de créer et de modifier l'organisation des données dans la base de données.

Outre le langage de définition des données, la partie *langage de manipulation des données* de SQL permet de rechercher, d'ajouter, de modifier ou de supprimer des données dans les bases de données relationnelles, la partie *langage de contrôle de transaction* permet de commencer et de terminer des transactions, et la partie *langage de contrôle des données* permet d'autoriser ou d'interdire l'accès à certaines données à certaines personnes.

Les instructions de manipulation des métadonnées - description de la structure, l'organisation et les caractéristiques de la base de données - commencent avec les mots clés *CREATE*, *ALTER* ou *DROP* qui correspondent aux opérations d'ajouter, modifier ou supprimer une métadonnée. Ces mots clés sont immédiatement suivis du type de métadonnée à manipuler - *TABLE*, *VIEW*, *INDEX*.

Les instructions de manipulation du contenu de la base de données commencent par les mots clés *SELECT*, *UPDATE*, *INSERT* ou *DELETE* qui correspondent respectivement aux opérations de recherche de contenu, modification, ajout et suppression. Divers mots clés tels que *FROM*, *JOIN* et *GROUP* permettent d'indiquer les opérations d'algèbre relationnelle à effectuer en vue d'obtenir le contenu à manipuler.

Les mots clés *GRANT* et *REVOKE* permettent d'autoriser des opérations à certaines personnes, d'ajouter ou de supprimer des autorisations. Tandis que les mots clés *COMMIT* et *ROLLBACK* permettent de confirmer ou annuler l'exécution de transactions.

SQL a été Créé en 1974, normalisé depuis 1986, le langage est reconnu par la grande majorité des systèmes de gestion de bases de données relationnelles (abrégé SGBDR) du marché.

- ✓ Microsoft Access
- ✓ Microsoft SQL Server
- ✓ MySQL
- ✓ Oracle

I. SQL langage de définition de données

Avant d'aborder la création des tables avec les commandes SQL, voici un petit lexique entre le modèle relationnel et SQL :

Modèle relationnel	**SQL**
relation	table
attribut	colonne
tuple	ligne

Quelques règles d'écritures en SQL :

Un nom d'objet (table, base, colonne contrainte, vue, etc…) doit avoir les caractéristiques suivantes :

- Ne pas dépasser 128 caractères
- Commencer par une lettre
- Etre composé de lettre, de chiffre, du caractère '_'
- Ne pas être un mot réservé en SQL
- Pas de caractères accentués

Il faut noter que les noms d'objets sont insensibles à la casse.

Conseils généraux sur la programmation SQL :

- SQL ne fait pas la différence entre les minuscules et les majuscules.
- Le langage autorise autant d'espaces et de saut de lignes que l'on veut entre les mots de la programmation.
- Les commentaires de code sont entre double tirets -- ;
- Il ne faut pas oublier le point-virgule à la fin de chaque instruction, car c'est lui qui déclenche son exécution.

Types de données :

Les principaux types de données ainsi que leurs limites sont décrits dans le tableau suivant :

Type	**Syntaxe**	**Description**	**limites**
Chaines de caractères	CHAR	Un seul caractère	
	CHAR(n)	Chaine de longueur fixe n (complétés par des espaces)	$0 \leq n \leq 2000$
	VARCHAR2(n)	Chaine de longueur variable ≤ n (taille ajustée au contenu)	$0 \leq n \leq 4000$
Nombres	NUMBER(p)	Entier à p chiffres décimaux	$1 \leq p \leq 38$
	NUMBER(p,s)	Nombre réel à virgule fixe à p chiffres décalés de s chiffres après chiffres après la virgule	$-84 \leq s \leq 127$
	NUMBER	Nombre réel à virgule flottante avec environ 16 chiffres significatifs	
Date et heure	DATE	Stockée sous forme Century Year Month Day Hour Minute Second	

Table 1: Principaux types de données Oracle

I.2 Création de table

L'objectif de ce paragraphe est d'apprendre à transmettre à un S.G.B.D. R. (Système de Gestion de Bases de Données Relationnelles) la structure d'une base de données relationnelle. Cette déclaration consiste :

- à **indiquer les champs** (encore appelés colonnes ou attributs) qui composent chacune des tables du modèle relationnel. Dans cette première étape, la définition d'un champ se limitera à la donnée de son nom et de son type ; ces deux informations étant issues du dictionnaire des données établi lors de la phase d'analyse conceptuelle.
- à **implémenter des contraintes d'intégrité** destinées à garantir la cohérence des données mémorisées ; dans cette seconde étape seront mises en place (entre autres) les notions de clé primaire et de clé étrangère représentées sur le modèle relationnel textuel respectivement par un soulignement et un #.

Pour créer une table en SQL, il existe l'instruction CREATE TABLE dans laquelle sont précisés pour chaque colonne de la table : son intitulé, son type de donnée et une ou plusieurs contraintes.

CREATE TABLE <nom de la table> (<nom de colonne> <type> <contrainte> , <nom de colonne> <type> <contrainte>,<contrainte> ,…)**;**

où <contrainte> représente la liste des contraintes d'intégrité structurelles concernant les colonnes de la table crée.

Exemple :

Considérons le schéma relationnel suivant :

COMMANDE(NoCommande, DateCommande, DateLivraison, #NoClient)

CLIENT (NoClient, NomClient, TypeClient)

CREATE TABLE Client **(**NoClient Integer**,** NomClient Varchar(25)**,** TypeClient VarChar(15) **)**

CREATE TABLE Commande (NoCommande Integer, DateCommande Date, DateLivraison Date, NoClient Integer)

I.2 Définition des contraintes d'intégrités

Une contrainte d'intégrité est une clause permettant de contraindre la modification de tables, faite par l'intermédiaire de requêtes d'utilisateurs, afin que les données saisies dans la base soient conformes aux données attendues. Ces contraintes doivent être exprimées dès la création de la table grâce aux mots clés suivants :

CONSTRAINT

DEFAULT

NOT NULL

UNIQUE

CHECK

- ✓ **Contrainte du domaine**

On entend par domaine le type des attributs d'une table.

Il ne peut y avoir de comparaison entre deux valeurs d'attributs si ces derniers ne sont pas définit sur le même domaine. Le SGBD-R se charge de vérifier la validité des valeurs d'attributs.

Un enregistrement ne peut être inséré dans une table que si chaque champ de l'enregistrement vérifie la contrainte d'intégrité de domaine de la colonne pour laquelle il est destiné.

- ✓ **Définir une valeur par défaut**

Le langage SQL permet de définir une valeur par défaut lorsqu'un champ de la base n'est pas renseigné grâce à la clause **DEFAULT**. Cela permet notamment de faciliter la création de tables, ainsi que de garantir qu'un champ ne sera pas vide.

La clause **DEFAULT** doit être suivie par la valeur à affecter. Cette valeur peut être un des types suivants :

Constante numérique

Constante alphanumérique (chaîne de caractères)

Le mot clé **USER** (nom de l'utilisateur)

Le mot clé **NULL**

Le mot clé **CURRENT_DATE** (date de saisie)

Le mot clé **CURRENT_TIME** (heure de saisie)

Le mot clé **CURRENT_TIMESTAMP** (date et heure de saisie)

- ✓ **Forcer la saisie d'un champ**

Le mot clé **NOT NULL** permet de spécifier qu'un champ doit être saisi, c'est-à-dire que le SGBD refusera d'insérer des tuples dont un champ comportant la clause **NOT NULL** n'est pas renseigné.

- ✓ **Emettre une condition sur un champ**

Il est possible de faire un test sur un champ grâce à la clause **CHECK()** comportant une condition logique portant sur une valeur entre les parenthèses. Si la valeur saisie est différente de NULL, le SGBD va effectuer un test grâce à la condition logique.

CHECK (attribut <condition>)

avec <condition> qui peut être une expression booléenne "simple" ou de la forme **IN** (liste de valeurs) ou **BETWEEN** <borne inférieure> **AND** <borne supérieure>.

- ✓ **Tester l'unicité d'une valeur**

La clause **UNIQUE** permet de vérifier que la valeur saisie pour un champ n'existe pas déjà dans la table. Cela permet de garantir que toutes les valeurs d'une colonne d'une table seront différentes.

Exemple : création de table avec contrainte

Voici un exemple permettant de voir la syntaxe d'une instruction de création de table avec contraintes :

```
CREATE TABLE clients(
Nom char(30) NOT NULL,
Prenom char(30) NOT NULL,
Age integer CHECK (age < 100),
Email char(50) NOT NULL, CHECK (Email LIKE "%@%"));
```

- ✓ **Définition de clés**

Grâce à SQL, il est possible de définir des clés, c'est-à-dire spécifier la (ou les) colonne(s) dont la connaissance permet de désigner précisément un et un seul tuple (une ligne).

L'ensemble des colonnes faisant partie de la table en cours permettant de désigner de façon unique un tuple est appelé **clé primaire** et se définit grâce à la clause **PRIMARY KEY** suivie de la liste de colonnes, séparées par des virgules, entre parenthèses. Ces colonnes ne peuvent alors plus prendre la valeur **NULL** et doivent être telles que deux lignes ne puissent avoir simultanément la même combinaison de valeurs pour ces colonnes.

PRIMARY KEY (colonne1, colonne2, ...)

Lorsqu'une liste de colonnes de la table en cours de définition permet de définir la clé primaire d'une table étrangère, on parle alors de **clé étrangère**, et on utilise la clause

FOREIGN KEY suivie de la liste de colonnes de la table en cours de définition, séparées par des virgules, entre parenthèses, puis de la clause **REFERENCES** suivie du nom de la table étrangère et de la liste de ses colonnes correspondantes, séparées par des virgules, entre parenthèses.

```
FOREIGN KEY (colonne1, colonne2, ...)
REFERENCES Nom_de_la_table_etrangere(colonne1,colonne2,...)
```

Attachée à un champ "clé étrangère" cette contrainte garantit que toute valeur prise par ce champ appartienne à l'ensemble des valeurs de la clé primaire.

- ✓ **Nommer une contrainte**

Il est possible de donner un nom à une contrainte grâce au mot clé **CONSTRAINT** suivi du nom que l'on donne à la contrainte, de telle manière à ce que le nom donné s'affiche en cas de non respect de l'intégrité, c'est-à-dire lorsque la clause que l'on a spécifiée n'est pas validée.

Si la clause **CONSTRAINT** n'est pas spécifiée, un nom sera donné arbitrairement par le SGBD.

Toutefois, le nom donné par le SGBD risque fortement de ne pas être compréhensible, et ne sera vraisemblablement pas compris lorsqu'il y aura une erreur d'intégrité. La stipulation de cette clause est donc fortement conseillée.

CREATE TABLE <nom de la table> (<nom de colonne> <type> **CONSTRAINT** <nom de la contrainte> <contrainte> ;

Pour nommer les contraintes, il est d'usage de suivre la norme indiquée ci-dessous ; dans une même base de données on ne peut pas avoir deux contraintes qui portent le même nom.

Les abréviations les plus fréquemment utilisées pour chaque type de contraintes sont :

PK Intégrité d'entité (Primary Key : clé primaire)

CK Intégrité de domaine (ChecK : contrôle)

FK Intégrité référentielle (Foreign Key : clé étrangère)

PK_Client est le nom associé à la contrainte de clé primaire de la table client.

Syntaxe: **CONSTRAINT FK_**NomChamp **FOREIGN KEY** (NomChamp) **REFERENCES** NomTable (NomChamp) **[ON DELETE CASCADE**]

L'option ON DELETE CASCADE permet de supprimer une ligne d'une table ainsi que toutes les lignes liées dans une autre table.

Exemple :

```
CREATE TABLE Client
(NoClient Integer,
NomClient VARCHAR(25) NOT NULL,
TypeClient VARCHAR(15) NOT NULL,
CONSTRAINT PK_Client PRIMARY KEY (NoClient),
CONSTRAINT CK_Client CHECK (TypeClient = "PARTICULIER " OR TypeClient
=" PROFESSIONNEL ") ;
```

- **Règles à respecter pour garantir l'intégrité référentielle**

Instruction	**Table « parent »**	**Table « fils »**
Insert	Correcte si la clé primaire est unique	Correcte si la clé étrangère est référencée dans la table père ou est nulle

update	Correcte si l'instruction ne laisse pas d'enregistrements dans la table fils ayant une clé étrangère non référenciée	Correcte si la nouvelle clé étrangère référence un enregistrement père existant
delete	Correcte si aucun enregistrement de la table fils ne référence le ou les enregistrements détruits	Correcte sans condition
Delete cascade	Correcte sans condition	Correcte sans condition
Delete set null	Correcte sans condition	Correcte sans condition

L'option ON DELETE CASCADE permet de supprimer une ligne d'une table ainsi que tous les lignes liées dans une autre table.

Exemple :

On suppose que le contenu des tables CLIENT et COMMANDE est limité aux lignes suivantes:

CLIENT

NoClient	***NomClient***	***TypeClient***
1	*DUBOIS*	*PROFESSIONNEL*
2	*DELAGE*	*PARTICULIER*
3	*DUPONT*	*PROFESSIONNEL*

COMMANDE

N°Commande	*DateLivraison*	*DateCommande*	*NoClient*
101	*15/12/1999*		*3*
102	*17/12/1999*	*18/12/1999*	*1*
103	*17/12/1999*	*22/12/1999*	*1*

La suppression du client Numéro 1 dans la table CLIENT pourrait :

- soit entraîner la suppression des commandes 102 et103 dans la table COMMANDE si l'option ON DELETE CASCADE est mentionnée,
- soit entraîner un refus de suppression de ce client si l'option ON DELETE CASCADE n'est pas mentionnée ; dans ce cas il faut préalablement supprimer les commandes 102 et 103 dans la table COMMANDE pour ensuite pouvoir supprimer le client Numéro 1.
- soit entraîner la suppression de ce client et de vider le champ *NoClient* pour les commandes 102 et103 dans la table COMMANDE si l'option ON DELETE SET NULL est mentionnée.

V. Modification de la structure d'une table

- ✓ **Ajout :**

ALTER TABLE <nom de la table> **ADD** (<nom de colonne> <type>[, <contrainte>]…);

ALTER TABLE <nom de la table> **ADD** (<contrainte> , <contrainte>…);

- ✓ **Modification :**

ALTER TABLE <nom de la table> **MODIFY** ([<nom de colonne> <nouveau type> ,<nom de colonne> <nouveau type>,…);

VI. Consultation de la structure d'une base

DESCRIBE <nom de la table>;

VII. Destruction de table

DROP TABLE <nom de la table>;

II. SQL langage de manipulation de données

II.1 Insertion de données

Pour insérer une ou plusieurs lignes dans une seule table, SQL offre l'instruction INSERT INTO. Lorsque l'on connait directement les valeurs à insérer, on utilise l'instruction INSERT INTO…VALUES.

INSERT INTO <nom de table> (colonne, …) **VALUES** (valeur, …)

On peut également insérer dans une table, le résultat d'une requête SELECT, auquel cas plusieurs lignes peuvent être insérer à la fois.

INSERT INTO <nom de table> (colonne, …) **SELECT…**

Plusieurs précautions doivent être prises en considérations :

- Les valeurs qui sont données via VALUES doivent être dans le même ordre que les colonnes qui sont précisés dans INTO.
- si la moindre valeur insérée ne vérifie pas les contraintes d'intégrités de la table, alors l'instruction INSERT INTO est refusée en entier par le serveur ORACLE.
- les colonnes qui figurent dans la clause SELECT doivent être compatibles en type de données, en nombre et en ordre avec celles qui sont précisés dans la clause INTO.

Exemple :

```
INSERT INTO clients (num_client, nom_client, prenom_client) VALUES (10, 'morabit',' ahmed') ;
```

Pour que num_client reçoie un entier qui s'incrémente automatiquement, il suffit de créer, avant la première insertion, une séquence pour cette table.

```
CREATE SEQUENCE sq_clients

MINVALUE 1

MAXVALUE 99

START WITH 1

INCREMENT BY 1

INSERT INTO clients (Num, Nom, Prenom) VALUES (sq_clients.NEXTVAL, 'morabit', 'ahmed') ;
```

II.2 Modification de données

Pour modifier la valeur d'une ou plusieurs colonnes, d'une ou plusieurs lignes, d'une seule table, on utilise l'instruction UPDATE…SET…WHERE.

UPDATE <nom de table> **SET** colonne = valeur, … [**WHERE** condition]
UPDATE <nom de table> **SET** colonne = **SELECT…**

Exemple :

UPDATE articles SET prix_unitaire=prix_unitaire*0 .152449 ;

UPDATE articles SET prix_unitaire=prix_unitaire*2 WHERE couleur=rouge ;

II.3 Suppression de données

Pour supprimer une ou plusieurs lignes dans une seule table, on utilise la syntaxe suivante :

DELETE FROM <nom de table> [**WHERE** condition];

Exercice :

1. Créer les tables suivantes :

   Companie(**comp**, nrue, ville, nomCom)
   Pilote(**brevet**, nom, nbHVol, #***comp***)
   Avion(**immat**, typeAvion, nbhVol, #***comp*** (not null))
   Affreter(#***comp***,# ***immat***,**dateAff**, nbPas)

2. Insérer dans la table Pilote les enregistrements suivants :
   ('PL-1', 'Paul1', 1000, 'SING')
   ('PL-2', 'Un connu', 0, 'NULL')
   ('PL-3', 'Paul2', 0, ' ?')

3. Insérer dans la table Avion les enregistrements suivants :
   ('F-WTSS, 'Concorde', 6570, 'SING')

4. Insérer dans la table Affreter les enregistrements suivants :
   ('AF, 'F-WTSS', '15-05-2003', 82)

5. Modifier la table Pilote en remplaçant *compa* par 'TOTO' pour brevet='PL-1'.
6. Modifier la table Pilote en remplaçant *compa* par 'SING' pour brevet='PL-2'.

7. Modifier la table Pilote en remplaçant *brevet* par ‘PL3bis’ pour brevet=’PL-1’.
8. Supprimer les enregistrements de la table Avion.

II.4 Interrogation

Pour afficher les données sous forme de tableau dont les colonnes portent un intitulé. On utilise l’instruction suivante :

SELECT [**DISTINCT**] <nom de colonne>[, <nom de colonne>]…
FROM <nom de table>[, <nom de table>]…
WHERE <condition>
GROUP BY <nom de colonne>[, <nom de colonne>]…
HAVING <condition avec calculs verticaux>
ORDER BY <nom de colonne>[, <nom de colonne>]…

II.4.1 Les clauses de SELECT

Base utilisée :
PILOTE(NUMPIL, NOMPIL, PRENOMPIL, ADRESSE, SALAIRE, PRIME)
AVION(NUMAV, NOMAV, CAPACITE, LOCALISATION)
VOL(NUMVOL,#***NUMPIL***,#***NUMAV***, DATE_VOL, HEURE_DEP, HEURE_ARR, VILLE_DEP, VILLE_ARR).
On suppose qu'un vol, référencé par son numéro NUMVOL, est effectué par un unique pilote, de numéro NUMPIL, sur un avion identifié par son numéro NUMAV.
Rappel de la syntaxe :

SELECT [**DISTINCT**] <nom de colonne>[, <nom de colonne>]…
FROM <nom de table>[, <nom de table>]…
[WHERE <condition>**]**
[GROUP BY <nom de colonne>[, <nom de colonne>]…
[HAVING <condition avec calculs verticaux>**]]**
[ORDER BY <nom de colonne>[, <nom de colonne >]…**]**

II.4.1.1 Expression des projections

SELECT *FROM *table*

où *table* est le nom de la table à consulter. Le caractère * signifie qu'aucune projection n'est réalisée, i.e. que tous les attributs de la table font partie du résultat.

Exemple : donner toutes les informations sur les pilotes.

SELECT ***FROM** PILOTE ;

La liste d'attributs sur laquelle la projection est effectuée doit être précisée après la clause **SELECT**.

Exemple : donner le nom et l'adresse des pilotes.

SELECT NOMPIL, ADRESSE **FROM** PILOTE;

- **Alias:**

SELECT *colonne [alias]*,…

Si alias est composé de plusieurs mots (séparés par des blancs), il faut utiliser des guillemets.

Exemple : sélectionner l'identificateur et le nom de chaque pilote.

SELECT NUMPIL "Num pilote", NOMPIL Nom_du_pilote **FROM** PILOTE ;

Suppression des duplicats: mot clef SQL **DISTINCT**

Exemple : quelles sont toutes les villes de départ des vols ?

SELECT DISTINCT VILLE_DEP **FROM** VOL ;

II.4.1.2 Expression des sélections

- **Clause WHERE**

Exemple : donner le nom des pilotes qui habitent à Marseille.

SELECT NOMPIL **FROM** PILOTE **WHERE** ADRESSE = 'MARSEILLE' ;

Exemple : donner le nom et l'adresse des pilotes qui gagnent plus de 3.000.

SELECT NOMPIL, adresse **FROM** PILOTE **WHERE** SALAIRE > 3000;

Autres opérateurs spécifiques :

- **IS NULL** : teste si la valeur d'une colonne est une valeur nulle (inconnue).

Exemple : rechercher le nom des pilotes dont l'adresse est inconnue.

SELECT NOMPIL **FROM** PILOTE **WHERE** ADRESSE **IS NULL**

- **IN** (liste) : teste si la valeur d'une colonne coïncide avec l'une des valeurs de la liste.

Exemple : rechercher les avions de nom A310, A320, A330 et A340.

SELECT * **FROM** AVION **WHERE** NOMAV **IN** ('A310', 'A320', 'A330', 'A340');

- **BETWEEN** v1 **AND** v2 : teste si la valeur d'une colonne est comprise entre les valeurs v1 et v2 (v1 <= valeur <= v2).

Exemple : quels sont les noms des pilotes qui gagnent entre 3.000 et 5.000 ?

SELECT NOMPIL **FROM** PILOTE **WHERE** SALAIRE **BETWEEN** 3000 **AND** 5000

- **LIKE** 'chaîne_générique'. Le symbole **%** remplace une chaîne de caractère quelconque, y compris le vide.

Exemple : quelle est la capacité des avions de type Airbus ?

SELECT CAPACITE **FROM** AVION **WHERE** NOMAV **LIKE** 'A%';

Négation des opérateurs spécifiques : **NOT**.

Nous obtenons alors **IS NOT NULL**, **NOT IN**, **NOT LIKE** et **NOT BETWEEN**.

Exemple : quels sont les noms des avions différents de A310, A320, A330 et A340 ?

SELECT NOMAV **FROM** AVION **WHERE** NOMAV **NOT IN** ('A310','A320','A330','A340');

Opérateurs logiques : **AND** et **OR**. Le **AND** est prioritaire et les parenthèses doivent être utilisé pour modifier l'ordre d'évaluation.

Exemple : quels sont les vols au départ de Marseille desservant Paris ?

SELECT * **FROM** VOL **WHERE** VILLE_DEP = 'MARSEILLE' **AND** VILLE_ARR ='PARIS' ;

Requête paramétrées : les paramètres de la requête seront quantifiés au moment de l'exécution de la requête. Pour cela, il suffit dans le texte de la requête de substituer aux différentes constantes de sélection des symboles de variables qui doivent systématiquement commencer par le caractère &. Si la constante de sélection est alphanumérique, on peut spécifier '&var' dans la requête, l'utilisateur n'aura plus qu'à saisir la valeur

Exemple : quels sont les vols au départ d'une ville et dont l'heure d'arrivée est inférieure à une certaine heure ?

SELECT * **FROM** VOL **WHERE** VILLE_DEP = '&1' **AND** HEURE_ARR < &2

Lors de l'exécution de cette requête, le système demande à l'utilisateur de lui indiquer une ville de départ et une heure d'arrivée.

II.4.1.3 Calculs horizontaux

Des expressions arithmétiques peuvent être utilisées dans les clauses **WHERE** et **SELECT**.
Ces expressions de calcul horizontal sont évaluées puis affichées et/ou testées pour chaque tuple appartenant au résultat.

Exemple : donner le revenu mensuel des pilotes Bordelais.

> **SELECT** NUMPIL, NOMPIL, SALAIRE + PRIME **FROM** PILOTE **WHERE** ADRESSE = 'BORDEAUX'

Exemple : quels sont les pilotes qui avec une augmentation de 10% de leur prime gagnent moins de 5.000 € ? Donner leur numéro, leurs revenus actuel et simulé.

> **SELECT** NUMPIL, SALAIRE+PRIME, SALAIRE + (PRIME*1.1) **FROM** PILOTE **WHERE** SALAIRE + (PRIME*1.1) < 5000

Expression d'un calcul :
Les opérateurs arithmétiques +, -, *, et / sont disponibles en **SQL**. De plus, l'opérateurs || (concaténation de chaînes de caractères: C1 || C2 correspond à concaténation de C1 et C2). Enfin, les opérateurs + et - peuvent être utilisés pour ajouter ou soustraire un nombre de jour à une date. L'opérateur - peut être utilisé entre deux dates et rend le nombre de jours entre les deux dates arguments.
Les fonctions disponibles en SQL dépendent du SGBD. Sous ORACLE, les fonctions suivantes sont disponibles :
ABS(n) la valeur absolue de *n* ;
FLOOR(n) la partie entière de *n* ;
POWER(m, n) m à la puissance *n* ;
TRUNC(n[, m]) *n* tronqué à *m* décimales après le point décimal. Si *m* est négatif, la troncature se fait avant le point décimal ;
ROUND(n [, d]) arrondit *n* à dix puissance -*d* ;
CEIL(n) entier directement supérieur ou égal à *n* ;
MOD(n, m) *n* modulo *m* ;
SIGN(n) 1 si $n > 0$, 0 si $n = 0$, -1 si $n < 0$;
SQRT(n) racine carrée de *n* (NULL si $n < 0$) ;
GREATEST(n1, n2,…) la plus grande valeur de la suite ;
LEAST(n1, n2,…) la plus petite valeur de la liste ;

NVL(n1, n2) permet de substituer la valeur *n2* à *n1*, au cas où cette dernière est une valeur nulle ;

LENGTH(ch) longueur de la chaîne ;

SUBSTR(ch, pos [, long]) extraction d'une sous-chaîne de *ch* à partir de la position *pos* en donnant le nombre de caractères à extraire *long* ;

INSTR(ch, ssch [, pos [, n]]) position de la sous-chaîne dans la chaîne ;

UPPER(ch) mise en majuscules ;

LOWER(ch) mise en minuscules ;

INITCAP(ch) initiale en majuscules ;

LPAD(ch, long [, car]) complète *ch* à gauche à la longueur *long* par *car* ;

RPAD(ch, long [, car]) complète *ch* à droite à la longueur *long* par *car* ;

LTRIM(ch, car) élague à gauche *ch* des caractères *car* ;

RTRIM(ch, car) élague à droite *ch* des caractères *car* ;

TRANSLATE(ch, car_source, car_cible) change *car_source* par *car_cible* ;

TO_CHAR(nombre, ch) convertit un nombre ou une date en chaîne de caractères ;

TO_NUMBER(ch) convertit la chaîne en numérique ;

ASCII(ch) code ASCII du premier caractère de la chaîne ;

CHR(n) conversion en caractère d'un code ASCII ;

TO_DATE(ch[, fmt]) conversion d'une chaîne de caractères en date ;

ADD_MONTHS(date, nombre) ajout d'un nombre de mois à une date ;

MONTHS_BETWEEN(date1, date2) nombre de mois entre *date1* et *date2* ;

LAST_DAY(date) date du dernier jour du mois ;

NEXT_DAY(date, nom du jour) date du prochain jour de la semaine ;

SYSDATE date du jour.

Exemple : donner la partie entière des salaires des pilotes.

SELECT NOMPIL,**FLOOR**(SALAIRE) "Salaire:partie entière" **FROM** PILOTE

ATTENTION : Tous les SGBD n'évaluent pas correctement les expressions arithmétiques que si les valeurs de ses arguments ne sont pas **NULL**. Pour éviter tout problème, il convient d'utiliser la fonction **NVL** (décrite ci avant) qui permet de substituer une valeur par défaut aux valeurs nulles éventuelles.

***Exemple** : l'attribut Prime pouvant avoir des valeurs nulles, la requête donnant le revenu mensuel des pilotes toulousains doit se formuler de la façon suivante.*

SELECT NUMPIL, NOMPIL, SALAIRE + NVL(PRIME,0) **FROM** PILOTE

WHERE ADRESSE = 'TOULOUSE'

II.4.1.4 Calculs verticaux (fonctions agrégatives)

Les fonctions agrégatives s'appliquent à un ensemble de valeurs d'un attribut de type numérique sauf pour la fonction de comptage **COUNT** pour laquelle le type de l'attribut argument est indifférent.

Contrairement aux calculs horizontaux, le résultat d'une fonction agrégative est évalué une seule fois pour tous les tuples.

Les fonctions agrégatives disponibles sont généralement les suivantes :

SUM somme,

AVG moyenne arithmétique,

COUNT nombre ou cardinalité,

MAX valeur maximale,

MIN valeur minimale,

STDDEV écart type,

VARIANCE variance.

Chacune de ces fonctions a comme argument un nom d'attribut ou une expression arithmétique. Les valeurs nulles ne posent pas de problème d'évaluation.

La fonction **COUNT** peut prendre comme argument le caractère *, dans ce cas, elle rend comme résultat le nombre de lignes sélectionnées par le bloc.

Exemple : quel est le salaire moyen des pilotes Marseillais.

SELECT AVG(SALAIRE)
FROM PILOTE
WHERE ADRESSE = 'MARSEILLE'

Exemple : trouver le nombre de vols au départ de Marseille.

SELECT COUNT(NUMVOL)
FROM VOL
WHERE VILLE_DEP = 'MARSEILLE'

Dans cette requête, **COUNT**(*) peut être utilisé à la place de **COUNT**(NUMVOL)

La colonne ou l'expression à laquelle est appliquée une fonction agrégative peut avoir des valeurs redondantes. Pour indiquer qu'il faut considérer seulement les valeurs distinctes, il faut faire précéder la colonne ou l'expression de **DISTINCT**.

Exemple : combien de destinations sont desservies au départ de Bordeaux ?

SELECT COUNT (**DISTINCT** VILLE_ARR) **FROM** VOL **WHERE** VILLE_DEP = 'BORDEAUX'

Exercice2 : Interrogation de la base de données.

Considérons la base de données suivante :

Employe(**num_emp**,nom_emp, fonction, manager, date_emb, salaire, commission,***num_dept***)

Departement(***num_dept***, nom_dept, locale)

Ecrire les scripts SQL qui permettent d'obtenir les informations suivantes.

1. Informations sur les employés dont la fonction est "MANAGER" dans les départements 20 et 30.
2. Liste des employés qui n'ont pas la fonction "MANAGER" et qui ont été embauchés en 81.
3. Liste des employés ayant un "M" et un "A" dans leur nom.
4. Liste des employés ayant deux "A" dans leur nom
5. Liste des employés ayant une commission
6. Liste des employés travaillant à "DALLAS"
7. Noms et dates d'embauche des employés embauchés avant leur manager, avec le nom et la date d'embauche du manager
8. Noms et dates d'embauche des employés embauchés avant 'BLAKE'
9. Employés embauchés le même jour que 'FORD'
10. Employés ayant le même manager que 'CLARK'
11. Employés embauchés avant tous les employés du département 10
12. Employés ayant le même job et même manager que 'TURNER'
13. Employés de département 'RESEARCH' embauchés le même jour que quelqu'un du département 'SALES'.
14. Employés gagnant plus que leur manager
15. Liste des noms des employés avec les salaires tronqués au millier

II.4.1.5 Expression des jointures sous forme prédicative

Les jointures permettent d'extraire des données issues de plusieurs tables. La clause **FROM** contient tous les noms de tables à fusionner. La clause **WHERE** exprime le critère de jointure sous forme de condition.

ATTENTION : si deux tables sont mentionnées dans la clause **FROM** et qu'aucune jointure n'est spécifiée, le système effectue le produit cartésien des deux relations (chaque tuple de la

première est mis en correspondance avec tous les tuples de la seconde), le résultat est donc faux car les liens sémantiques entre relations ne sont pas utilisés. Donc respectez et vérifiez la règle suivante.

Si *n* tables apparaissent dans la clause **FROM**, il faut au moins (n – 1) opérations de jointure.

La forme générale d'une requête de jointure est :

SELECT *colonne,…***FROM** *table1, table2…* **WHERE** <*condition*>

où <*condition*> est de la forme attribut1 attribut2.

Exemple : quel est le numéro et le nom des pilotes résidant dans la ville de localisation de l'avion n° 33 ?

SELECT NUMPIL, NOMPIL **FROM** PILOTE, AVION **WHERE** ADRESSE = LOCALISATION **AND** NUMAV = 33

Les noms des colonnes dans la clause **SELECT** et **WHERE** doivent être uniques. Ne pas oublier de lever l'ambiguïté à l'aide d'alias ou en préfixant les noms de colonnes.

Exemple : donner les noms des pilotes faisant des vols au départ de Marseille sur des Airbus ?

SELECT DISTINCT NOMPIL **FROM** PILOTE, VOL, AVION **WHERE** VILLE_DEP = 'MARSEILLE' **AND** NOMAV **LIKE** 'A%' **AND** PILOTE.NUMPIL = VOL.NUMPIL **AND** VOL.NUMAV = AVION. NUMAV

Alias de nom de table : se place dans la clause **FROM** après le nom de la table.

Exemple : quels sont les avions localisés dans la même ville que l'avion numéro 103 ?

SELECT AUTRES.NUMAV, AUTRES.NOMAV
FROM AVION AUTRES, AVION AV103
WHERE AV103.NUMAV = 103
AND AUTRES.NUMAV <> 103
AND AV103.LOCALISATION = AUTRES.LOCALISATION

Dans cette requête l'alias AV103 est utilisée pour retrouver l'avion de numéro 103 et l'alias AUTRES permet de balayer tous les tuples de AVION pour faire la comparaison des localisations.

Exemple : quelles sont les correspondances (villes d'arrivée) accessibles à partir de la ville d'arrivée du vol IT100 ?

SELECT DISTINCT AUTRES.VILLE_ARR

FROM VOL AUTRES, VOL VOLIT100
WHERE VOLIT100.NUMVOL = 'IT100' **AND**
VOLIT100.VILLE_ARR = AUTRES.VILLE_DEP

La requête recherche les vols dont la ville de départ correspond à la ville d'arrivée du vol IT100 puis ne conserve que les villes d'arrivée de ces vols.

II.4.1.6 Autre expression de jointures : forme imbriquée

1. Premier cas :

Le résultat de la sous-requête est formé *d'une seule valeur*. C'est surtout le cas de sous-requêtes utilisant une fonction agrégat dans la clause **SELECT**. L'attribut spécifié dans le **WHERE** est simplement comparé au résultat du **SELECT** imbriqué à l'aide de l'opérateur de comparaison voulu.

Exemple : quels sont les noms des pilotes gagnant plus que le salaire moyen des pilotes ?

SELECT NOMPIL **FROM** PILOTE **WHERE** SALAIRE > (**SELECT AVG**(SALAIRE) **FROM** PILOTE)

2. Deuxième cas :

Le résultat de la sous-requête est formé *d'une liste de valeurs*. Dans ce cas, le résultat de la comparaison, dans la clause **WHERE**, peut être considéré comme vrai :

- ✓ soit si la condition doit être vérifiée avec *au moins une des valeurs résultats* de la sous-requête. Dans ce cas, la sous-requête doit être précédée du comparateur suivi de **ANY** (remarque : = **ANY** est équivalent à **IN**) ;
- ✓ soit si la condition doit être vérifiée pour *toutes les valeurs résultats* de la sous requête, alors la sous-requête doit être précédée du comparateur suivi de **ALL**.

Exemple : quels sont les noms des pilotes en service au départ de Marseille ?

SELECT NOMPIL **FROM** PILOTE **WHERE** NUMPIL **IN** (**SELECT DISTINCT** NUMPIL **FROM** VOL **WHERE** VILLE_DEP = 'MARSEILLE');

Exemple : quels sont les numéros des avions localisés à Marseille dont la capacité est supérieure à celle de l'un des appareils effectuant un Paris-Marseille ?

SELECT NUMAV **FROM** AVION **WHERE** LOCALISATION = 'MARSEILLE' **AND** CAP > **ANY** (**SELECT DISTINCT** CAP **FROM** AVION **WHERE** NUMAV = **ANY** (**SELECT DISTINCT** NUMAV **FROM** VOL **WHERE** VILLE-DEP = 'PARIS' AND VILLE_ARR = 'MARSEILLE'))

Exemple : quels sont les noms des pilotes Marseillais qui gagnent plus que tous les pilotes parisiens ?

```
SELECT NOMPIL
FROM PILOTE
WHERE ADRESSE = 'MARSEILLE' AND SALAIRE > ALL
(SELECT DISTINCT SALAIRE
FROM PILOTE
WHERE ADRESSE = 'PARIS')
```

Exemple : donner le nom des pilotes Marseillais qui gagnent plus qu'un pilote parisien.

```
SELECT NOMPIL
FROM PILOTE
WHERE ADRESSE = 'MARSEILLE' AND SALAIRE > ANY
(SELECT SALAIRE
FROM PILOTE
WHERE ADRESSE = 'PARIS')
```

3. Troisième cas :

- ✓ Le résultat de la sous-requête est un *ensemble de colonnes*.
- ✓ le nombre de colonnes de la clause **WHERE** doit être identique à celui de la clause **SELECT** de la sous-requête. Ces colonnes doivent être mises entre parenthèses,
- ✓ la comparaison se fait entre les valeurs des colonnes de la requête et celle de la sous-requête deux à deux,
- ✓ il faut que l'opérateur de comparaison soit le même pour tous les attributs concernés.

Exemple : rechercher le nom des pilotes ayant même adresse et même salaire que Dupont.

```
SELECT NOMPIL
FROM PILOTE
WHERE NOMPIL <> 'DUPONT'
AND (ADRESSE, SALAIRE) IN
(SELECT ADRESSE, SALAIRE
FROM PILOTE
WHERE NOMPIL = 'DUPONT')
```

II.4.1.7 Tri des résultats

Il est possible avec SQL d'ordonner les résultats. Cet ordre peut être croissant (**ASC**) ou décroissant (**DESC**) sur une ou plusieurs colonnes ou expressions.

ORDER BY *expression* [**ASC** | **DESC**],…

L'argument de **ORDER BY** peut être un nom de colonne ou une expression basée sur une ou plusieurs colonnes mentionnées dans la clause **SELECT**.

Exemple : En une seule requête, donner la liste des pilotes Marseillais par ordre de salaire décroissant et par ordre alphabétique des noms.

SELECT NOMPIL, SALAIRE
FROM PILOTE
WHERE ADRESSE = 'MARSEILLE'
ORDER BY SALAIRE **DESC**, NOMPIL

II.4.1.8 Test d'absence de données

Pour vérifier qu'une donnée est absente dans la base, le cas le plus simple est celui où l'existence de tuples est connue et on cherche si un attribut a des valeurs manquantes.
Il suffit alors d'utiliser le prédicat **IS NULL**. Mais cette solution n'est correcte que si les tuples existent. Elle ne peut en aucun cas s'appliquer si on cherche à vérifier l'absence de tuples ou l'absence de valeur quand on ne sait pas si des tuples existent.

Exemple : Les requêtes suivantes ne peuvent pas être formulées avec **IS NULL**.
Quels sont les pilotes n'effectuant aucun vol ? Quelles sont les villes de départ dans lesquelles aucun avion n'est localisé ?
Pour formuler des requêtes de type " un élément n'appartient pas à un ensemble donné ", trois techniques peuvent être utilisées. La première consiste à utiliser une jointure imbriquée avec **NOT IN**. La sous-requête est utilisée pour calculer l'ensemble de recherche et le bloc de niveau supérieur extrait les éléments n'appartenant pas à cet ensemble.

Exemple : quels sont les pilotes n'effectuant aucun vol ?

SELECT NUMPIL, NOMPIL
FROM PILOTE
WHERE NUMPIL **NOT IN**
(**SELECT** NUMPIL **FROM** VOL)

Une deuxième approche fait appel au prédicat **NOT EXISTS** qui s'applique à un bloc imbriqué et rend la valeur vrai si le résultat de la sous-requête est vide (et faux sinon). Il faut faire très attention à ce type de requête car **NOT EXISTS** ne dispense pas d'exprimer des jointures. Ce danger est détaillé à travers l'exemple suivant.

Exemple : reprenons la requête précédente. La formulation suivante est fausse.

SELECT NUMPIL, NOMPIL
FROM PILOTE
WHERE NOT EXISTS
(**SELECT** NUMPIL **FROM** VOL)

Il suffit qu'un vol soit créé dans la relation VOL pour que la requête précédente retourne systématiquement un résultat vide. En effet le bloc imbriqué rendra une valeur pour NUMPIL et le **NOT EXISTS** sera toujours évalué à faux, donc aucun pilote ne sera retourné par le premier bloc.

Le problème de cette requête est que le lien entre les éléments cherchés dans les deux blocs n'est pas spécifié. Or, il faut indiquer au système que le 2ème bloc doit être évalué pour chacun des pilotes examinés par le 1er bloc. Pour cela, on introduit un alias pour la relation PILOTE et une jointure est exprimée dans le 2ème bloc en utilisant cet alias. La formulation correcte est la suivante :

SELECT NUMPIL, NOMPIL
FROM PILOTE PIL
WHERE NOT EXISTS
(**SELECT** NUMPIL
FROM VOL
WHERE VOL.NUMPIL = PIL.NUMPIL)

Enfin, il est possible d'avoir recours à l'opérateur de différence (**MINUS**).

SELECT NUMPIL **FROM** PILOTE
MINUS
SELECT NUMPIL **FROM** VOL

II.4.1.9 Classification ou partitionnement

La classification permet de regrouper les lignes d'une table dans des classes d'équivalence ou sous-tables ayant chacune la même valeur pour la colonne de la classification. Ces classes

forment une partition de l'extension de la relation considérée (i.e. l'intersection des classes est vide et leur union est égale à la relation initiale).

Exemple : considérons la relation VOL illustrée par la figure 1. Partitionner cette relation sur l'attribut NUMPIL consiste à regrouper au sein d'une même classe tous les vols assurés par le même pilote. NUMPIL prenant 4 valeurs distinctes, 4 classes sont introduites. Elles sont mises en évidence dans la figure 2 et regroupent respectivement les vols des pilotes n° 100, 102, 105 et 124.

NUMVOL	NUMPIL	…
IT100	100	
AF101	100	
IT101	102	
BA003	105	
BA045	105	
IT305	102	
AF421	124	
BA047	105	
BA087	105	

Figure 1 : relation exemple VOL

NUMVOL	NUMPIL	…
IT100 AF101	100 100	
IT101 IT305	102 102	
BA003 BA045 BA047 BA087	105 105 105 105	
AF421	124	

Figure 2 : regroupement selon NUMPIL

En SQL, l'opérateur de partitionnement s'exprime par la clause **GROUP BY** qui doit suivre la clause **WHERE** (ou **FROM** si **WHERE** est absente). Sa syntaxe est :

GROUP BY *colonne1*, *[colonne2,…]*

Les colonnes indiquées dans **SELECT**, sauf les attributs arguments des fonctions agrégatives, doivent être mentionnées dans **GROUP BY**.

Il est possible de faire un regroupement multi-colonnes en mentionnant plusieurs colonnes dans la clause **GROUP BY**. Une classe est alors créée pour chaque combinaison distincte de valeurs de la combinaison d'attributs.

En présence de la clause **GROUP BY**, les fonctions agrégatives s'appliquent à l'ensemble des valeurs de chaque classe d'équivalence.

Exemple : quel est le nombre de vols effectués par chaque pilote ?

```
SELECT NUMPIL, COUNT(NUMVOL)
FROM VOL
GROUP BY NUMPIL
```

Dans ce cas, le résultat de la requête comporte une ligne par numéro de pilote présent dans la relation VOL.

Exemple : combien de fois chaque pilote conduit-il chaque avion ?

SELECT NUMPIL, NUMAV, **COUNT**(NUMVOL)
FROM VOL
GROUP BY NUMPIL, NUMAV

Nous obtenons ici autant de lignes par pilote qu'il y a d'avions distincts conduits le pilote considéré. Chaque classe d'équivalence créée par le ***GROUP BY*** *regroupe tous les vols ayant même pilote et même avion.*

Partitionner une relation sur un attribut clé primaire est évidemment complètement inutile : chaque classe d'équivalence est réduite à un seul tuple !

De même, mentionner un **DISTINCT** devant les attributs de la clause **SELECT** ne sert à rien si le bloc comporte un **GROUP BY** : ces attributs étant aussi les attributs de partitionnement, il n'y aura pas de duplicats parmi les valeurs ou combinaisons de valeurs retournées. Il faut par contre faire très attention à l'argument des fonctions agrégatives. L'oubli d'un **DISTINCT** peut rendre la requête fausse.

Exemple : donner le nombre de destinations desservies par chaque avion.

SELECT NUMAV, **COUNT**(**DISTINCT** VILLE_ARR)
FROM VOL
GROUP BY NUMAV

Si le **DISTINCT** est oublié, c'est le nombre de vols qui sera compté et la requête sera fausse.

II.4.1.10 Recherche dans les sous-tables

Des conditions de sélection peuvent être appliquées aux sous-tables engendrées par la clause **GROUP BY**, comme c'est le cas avec la clause **WHERE** pour les tables. Cette sélection s'effectue avec la clause **HAVING** qui doit suivre la clause **GROUP BY**. Sa syntaxe est :

HAVING *condition*

La condition permet de comparer une valeur obtenue à partir de la sous-table à une constante ou à une autre valeur résultant d'une sous-requête.

Exemple : donner le nombre de vols, s'il est supérieur à 5, par pilote.

SELECT NUMPIL, **COUNT**(NUMVOL)
FROM VOL

GROUP BY NUMPIL

HAVING COUNT(NUMVOL) > 5

Exemple : quelles sont les villes à partir desquelles le nombre de villes desservies est le plus grand ?

SELECT VILLE_DEP

FROM VOL

GROUP BY VILLE_DEP

HAVING COUNT(**DISTINCT** VILLE_ARR) >= **ALL**

(**SELECT COUNT**(**DISTINCT** VILLE_ARR)

FROM VOL

GROUP BY VILLE_DEP)

Même si les clauses **WHERE** et **HAVING** introduisent des conditions de sélection mais elles sont différentes. Si la condition doit être vérifiée par chaque tuple d'une classe d'équivalence, il faut la spécifier dans le **WHERE**. Si la condition doit être vérifiée globalement pour la classe, elle doit être exprimée dans la clause **HAVING**.

II.4.1.11 Recherche dans une arborescence

La consultation des données de structure arborescente est une spécificité du SGBD ORACLE. Une telle structure de données est très fréquente dans le monde réel. Elle correspond à des compositions ou des liens hiérarchiques.

Exemple : supposons que l'on souhaite intégrer dans la base des informations sur la répartition géographique des villes desservies. La figure 3 illustre l'organisation hiérarchique des valeurs des différentes localisations à répertorier.

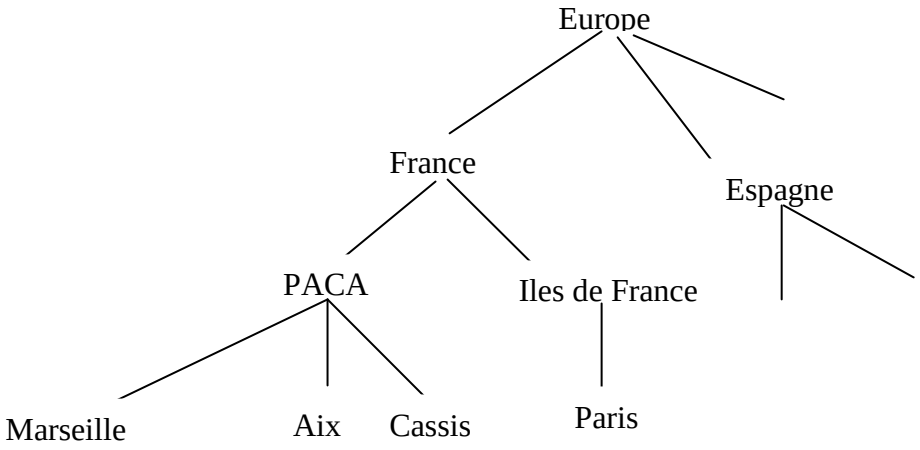

Figure 3 : hiérarchie des localisations

La relation LIEU est ajoutée au schéma initial de la base pour représenter de manière “ plate ” la structure arborescente donnée ci-dessus. Cette relation a le schéma suivant :

LIEU (LIEUSPEC, #***LIEUGEN***).

L’attribut LIEUGEN est défini sur le même domaine que LIEUSPEC. C’est donc une clef étrangère référençant la clef primaire de sa propre relation.

L'extension de cette relation, correspondant à la hiérarchie précédente, est la suivante :

LIEUSPEC	LIEUGEN
EUROPE	NULL
FRANCE	EUROPE
ESPAGNE	EUROPE
PACA	FRANCE
ILE-DE-FRANCE	FRANCE
MARSEILLE	PACA
AIX	PACA
CASSIS	PACA
PARIS	ILE-DE-FRANCE

Figure 4 : représentation relationnelle de la hiérarchie des localisations

La syntaxe générale d'une requête de recherche dans une arborescence est :

SELECT colonne,…
FROM table [alias],…
[**WHERE** condition]
CONNECT BY [**PRIOR**] colonne1 = [**PRIOR**] colonne 2
[**AND** condition]
[**START WITH** condition]
[**ORDER BY LEVEL**]

La clause **START WITH** fixe le point de départ de la recherche dans l'arborescence.

La clause **CONNECT BY PRIOR** fixe le sens de parcours de l'arborescence. Ce parcours peut se faire :

- de haut en bas, i.e. qu'il correspond à la recherche de sous-arbres ;
- de bas en haut, i.e. qu'il correspond à la recherche du ou des “ ancêtres ”.

Le parcours de haut en bas est spécifié en faisant précéder la colonne représentant un “ fils ” dans l'arborescence par le mot-clef **PRIOR** comme suit :

CONNECT BY PRIOR <fils> = <père> ou **CONNECT BY** <père> = **PRIOR** <fils>

Le parcours de bas en haut est spécifié en faisant précéder la colonne représentant un " père " par le mot-clef **PRIOR**. Il peut s'exprimer aussi de deux façons :

CONNECT BY <fils> = **PRIOR** <père> ou **CONNECT BY PRIOR** <père> = <fils>

Remarque : avec la représentation relationnelle plate, le <fils> dans une structure hiérarchique correspond à la clef primaire de la relation, le <père> à la clef étrangère. Lorsque la valeur de l'attribut <fils> est la racine de la hiérarchie, la valeur de l'attribut <père> est une valeur nulle : il s'agit d'un cas exceptionnel où l'on admettra une valeur nulle pour une clef étrangère.

Exemple : donner la liste des villes desservies en France.

```
SELECT LIEUSPEC
FROM LIEU
CONNECT BY PRIOR LIEUSPEC = LIEUGEN
START WITH LIEUGEN = 'FRANCE'
```

Exemple : où se situe (continent, pays, région) la ville de Valladolid ?

```
SELECT LIEUGEN
FROM LIEU
CONNECT BY LIEUSPEC = PRIOR LIEUGEN
START WITH LIEUSPEC = 'VALLADOLID'
```

La clause facultative [**AND** condition] permet de spécifier une condition évaluée lors de la recherche dans un sous-arbre. Plus précisément, si la condition n'est pas satisfaite pour un nœud de la hiérarchie alors qu'aucun de ses descendants ne sera plus examiner.

Exemple : donner la liste des villes desservies en France en dehors de la région AQUITAINE.

```
SELECT LIEUSPEC
FROM LIEU
CONNECT BY PRIOR LIEUSPEC = LIEUGEN
AND LIEUGEN <> 'AQUITAINE'
START WITH LIEUGEN = 'FRANCE'
ORDER BY LEVEL
```

Dès qu'un nœud ne vérifiant pas la condition donnée est trouvé, le sous-arbre dont ce nœud est racine est éliminé de la recherche, donc la région AQUITAINE et toutes les villes de cette région n'appartiendront pas au résultat.

ORACLE associe une numérotation aux nœuds d'une arborescence : c'est la notion de niveau (**LEVEL**). En fait le premier nœud répondant à la requête est considéré de niveau 1, ses fils

(dans le cas d'une recherche de haut en bas) ou son père (dans le cas d'une recherche de bas en haut) sont considérés de niveau 2, et ainsi de suite.

Cette numérotation peut être utilisée pour classer les résultats (clause **ORDER BY LEVEL**) et surtout pour en donner une meilleure présentation. Celle-ci s'obtient en utilisant la fonction **LPAD** qui permet une indentation.

Exemple : pour obtenir la liste des villes desservies en France mais avec présentation plus claire, nous pouvons formuler la requête comme suit :

```
SELECT LPAD('-', 2*LEVEL,' ')|| LIEUSPEC
FROM LIEU
CONNECT BY PRIOR LIEUSPEC = LIEUGEN
START WITH LIEUGEN = 'FRANCE'
ORDER BY LEVEL
```

|| est le symbole de concaténation de deux chaînes de caractères.

La fonction **LPAD** permet d'ajouter à la chaîne '-', un nombre d'espaces égal au double du niveau, devant chaque valeur de LIEUSPEC pour mieux visualiser la division géographique.

II.4.1.12 Expression des divisions

La division consiste à rechercher les tuples qui se trouvent associés à tous les tuples d'un ensemble particulier (le diviseur). SQL ne propose pas d'opérateur de division (quantificateur universel "tous les"). Il est néanmoins possible d'exprimer cette opération mais de manière peu naturelle. Les deux techniques utilisées consistent à reformuler la requête différemment, on parle de paraphrasage. Elles ne sont pas systématiquement applicables (suivant l'interrogation posée).

La première technique a recours aux partitionnements et fonctions agrégatives. Pour savoir si un tuple *t* est associé à tous les tuples du diviseur, l'idée est de compter d'une part le nombre de tuples associés à *t* et d'autre part le nombre de tuples du diviseur puis de comparer les valeurs obtenues.

Exemple : quels sont les numéros des pilotes qui conduisent tous les avions de la compagnie?

Pour l'exprimer en SQL avec la première technique exposée, cette requête est traduite de la manière suivante : " Quels sont les pilotes qui conduisent autant d'avions que la compagnie en possède ? ".

```
SELECT NUMPIL
FROM VOL
```

GROUP BY NUMPIL
HAVING COUNT(**DISTINCT** NUMAV) =
(**SELECT COUNT**(NUMAV)
FROM AVION)

Le comptage dans la clause **HAVING** permet pour chaque pilote de dénombrer les appareils conduits. L'oubli du **DISTINCT** rend la requête fausse (on compterait alors le nombre de vols assurés).

Cette technique de paraphrasage ne peut être utilisée que si les deux ensembles dénombrés sont parfaitement comparables.

La deuxième forme d'expression des divisions est plus délicate à formuler. Elle se base sur une double négation de la requête et utilise la clause **NOT EXISTS**. Au lieu d'exprimer qu'un tuple t doit être associé à tous les tuples du diviseur pour appartenir au résultat, on recherche t tel qu'il n'existe aucun tuple du diviseur qui ne lui soit pas associé. Rechercher des tuples associés ou non, signifie concrètement effectuer des opérations de jointure.

Exemple : quels sont les numéros des pilotes qui conduisent tous les avions de la compagnie? Pour l'exprimer en SQL, cette requête est traduite par : "Existe-t-il un pilote tel qu'il n'existe aucun avion de la compagnie qui ne soit pas conduit par ce pilote ?".

SELECT NUMPIL
FROM PILOTE P
WHERE NOT EXISTS (**SELECT** *
FROM AVION A
WHERE NOT EXISTS
(**SELECT** *
FROM VOL
WHERE VOL.NUMAV = A.NUMAV **AND** VOL.NUMPIL = P.NUMPIL))

Détaillons l'exécution de cette requête. Pour chaque pilote examiné par le 1er bloc, les différents tuples de AVION sont balayés au niveau du 2ème bloc et pour chaque avion, les conditions de jointure du 3ème bloc sont évaluées. Supposons que les trois relations de la base sont réduites aux tuples présentés dans la figure suivante.

Pilote Avion Vol

NUMPIL	…
100 101 102	

NUMAV	…
10 21	

NUMVOL	NUMPIL	NUMAV
IT100	100	10
IT200	100	10
AF214	101	21
AF321	102	10
AF036	101	10

Figure 5 : une instance de la base

Considérons le pilote n° 100. Parcourons les tuples de la relation AVION. Pour l'avion n° 10, le 3ème bloc retourne un résultat (le vol IT100), **NOT EXISTS** est donc faux et l'avion n° 10 n'appartient pas au résultat du 2ème bloc. L'avion n° 21 n'étant jamais piloté par le pilote 100, le 3ème bloc ne rend aucun tuple, le **NOT EXISTS** associé est donc évalué à vrai. Le 2ème bloc rend donc un résultat non vide (l'avion n°21) et donc le **NOT EXISTS** du 1er bloc est faux. Le pilote n° 100 n'est donc pas retenu dans le résultat de la requête.

Pour le pilote 101 et l'avion 10, il existe un vol (AF214). Le 3ème bloc retourne un résultat, **NOT EXISTS** est donc faux. Pour le même pilote et l'avion n° 21, le 3ème bloc restitue un tuple et à nouveau **NOT EXISTS** est faux. Le 2ème bloc rend donc un résultat vide ce qui fait que le **NOT EXISTS** du 1er bloc est évalué à vrai. Le pilote 101 fait donc partie du résultat de la requête. Le processus décrit est à nouveau exécuté pour le pilote 102 et comme il ne pilote pas l'avion 21, le 2ème bloc retourne une valeur et la condition du 1er bloc élimine ce pilote du résultat.

II.6 Gestion des transactions

Comme tout SGBD, ORACLE est un système transactionnel, i.e. il gère les accès concurrents de multiples utilisateurs à des données fréquemment mises à jour et soumises à diverses contraintes. Une transaction est une séquence d'opérations manipulant les données. Exécutée sur une base cohérente, elle laisse la base dans un état cohérent : toutes les opérations sont réalisées ou aucune.

Les commandes **COMMIT** et **ROLLBACK** de SQL permettent de contrôler la validation et l'annulation des transactions :

- **COMMIT** : valide la transaction en cours. Les modifications de la base deviennent définitives ;
- **ROLLBACK** : annule la transaction en cours. La base est restaurée dans son état au début de la transaction.

Une modification directe de la base (**INSERT**, **UPDATE** ou **DELETE**) ne sera validée que par la commande **COMMIT**, ou lors de la sortie normale.

En cours de transaction, seul l'utilisateur ayant effectué les modifications les voit. Ce mécanisme est utilisé par les systèmes de gestion de bases de données pour assurer l'intégrité de la base en cas de fin anormale d'une tâche utilisateur : il y a automatiquement ROLLBACK des transactions non terminées.

ORACLE est un système transactionnel qui assure la cohérence des données en cas de mise à jour de la base, même si plusieurs utilisateurs lisent ou modifient les mêmes données simultanément.

ORACLE utilise un mécanisme de verrouillage pour empêcher deux utilisateurs d'effectuer des transactions incompatibles et régler les problèmes pouvant survenir.

ORACLE permet le verrouillage de certaines unités (table ou ligne) automatiquement ou sur demande de l'utilisateur. Les verrous sont libérés en fin de transaction.

Il est possible de se mettre en mode "validation automatique", dans ce cas, la validation est effectuée automatiquement après chaque commande SQL de mise à jour de données. Pour se mettre dans ce mode, il faut utiliser l'une des commandes suivantes :

SET AUTOCOMMIT IMMEDIATE ou **SET AUTOCOMMIT ON**

L'annulation de ce mode se fait avec la commande : **SET AUTOCOMMIT OFF**

III. SQL langage de contrôle des données

III.1 Création d'index

Pour accélérer les accès aux tuples, la création d'index peut être réalisée pour un ou plusieurs attributs fréquemment consultés. La commande à utiliser est la suivante :

> **CREATE** [**UNIQUE**] [**NOCOMPRESS**] **INDEX** <nom_index> **ON** <nom_table> (<nom_colonne>, [nom_colonne]…) ;

Lorsque le mot-clef **UNIQUE** est précisé, l'attribut (ou la combinaison) indexé doit avoir des valeurs uniques.

Une fois qu'ils ont été créés, les index sont automatiquement utilisés par le système et de manière transparente pour l'utilisateur.

Remarque : lors de la spécification de la contrainte **PRIMARY KEY** pour un ou plusieurs attributs ORACLE génère automatiquement un index primaire (**UNIQUE**) pour le(s) attribut(s) concerné(s).

Exemple :

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

III.2 Création et utilisation des vues

Une vue permet de consulter et manipuler des données d'une ou de plusieurs tables.

On considère qu'une vue est une table virtuelle car elle peut être utilisée de la même façon qu'une relation mais ses données (redondantes par rapport à la base originale) ne sont pas physiquement stockées. Plus précisément, une vue est définie sous forme d'une requête d'interrogation et c'est cette requête qui est conservée dans le dictionnaire de la base.

L'utilisation des vues permet de :

- restreindre l'accès à certaines colonnes ou certaines lignes d'une table pour certains utilisateurs (confidentialité) ;
- simplifier la tâche de l'utilisateur en le déchargeant de la formulation de requêtes complexes ;
- contrôler la mise à jour des données.

La commande de création d'une vue est la suivante :

CREATE VIEW *nom_vue [(colonne,…)]*

AS<*requête*>**[WITH CHECK OPTION]**

Si une liste de noms d'attributs est précisée après le nom de la vue, ces noms seront ceux des attributs de la vue. Ils correspondent, deux à deux, avec les attributs indiqués dans le ***SELECT*** *de la requête définissant la vue. Si la liste de noms n'apparaît pas, les attributs de la vue ont le même nom que ceux attributs indiqués dans le* ***SELECT*** *de la requête.*

Il est très important de savoir comment la vue à créer sera utilisée : consultation simplement et/ou mises à jour.

Exemple : pour éviter que certains utilisateurs aient accès aux salaires et prime des pilotes, la vue suivante est définie à leur intention et ils n'ont pas de droits sur la relation PILOTE.

CREATE VIEW RESTRICT_PIL

AS

SELECT NUMPIL, NOMPIL, PRENOM_PIL, ADRESSE

FROM PILOTE

Exemple : pour épargner aux utilisateurs la formulation d'une requête complexe, une vue est définie par les développeurs pour consulter la charge horaire des pilotes. Sa définition est la suivante :

CREATE VIEW CHARGE_HOR (NUMPIL, NOM, CHARGE)
AS
SELECT P.NUMPIL, NOMPIL, **SUM**(HEURE_ARR – HEURE_DEP)
FROM PILOTE P, VOL
WHERE P.NUMPIL = VOL.NUMPIL
GROUP BY P.NUMPIL, NOMPIL

Lorsque cette vue est créée, les utilisateurs peuvent la consulter simplement par :

SELECT * **FROM** CHARGE_HOR

Un utilisateur ne s'intéressant qu'aux pilotes parisiens dont la charge excède un seuil de 40 heures formulera la requête suivante.

SELECT *
FROM CHARGE_HOR C, PILOTE P
WHERE C.NUMPIL = P.NUMPIL **AND** CHARGE > 40
AND ADRESSE = 'PARIS'

Lorsque le système évalue une requête formulée sur une vue, il combine la requête de l'utilisateur et la requête de définition de la vue pour obtenir le résultat.

Lorsqu'une vue est utilisée pour effectuer des opérations de mise à jour, elle est soumise à des contraintes fortes. En effet pour que les mises à jour, à travers une vue, soient automatiquement répercutées sur la relation de base associée, il faut impérativement que :

- la vue ne comprenne pas de clause **GROUP BY**.
- la vue n'ait qu'une seule relation dans la clause **FROM**. Ceci implique que dans une vue multi-relation, les jointures soient exprimées de manière imbriquée.

Lorsque la requête de définition d'une vue comporte une projection sur un sous ensemble d'attributs d'une relation, les attributs non mentionnés prendront des valeurs nulles en cas d'insertion à travers la vue.

Exemple : définir une vue permettant de consulter les vols des pilotes habitant Bayonne et de les mettre à jour.

CREATE VIEW VOLPIL_BAYONNE
AS
SELECT *

FROM VOL
WHERE NUMPIL **IN**
(**SELECT** NUMPIL
FROM PILOTE
WHERE ADRESSE = 'BAYONNE')

La vue précédente permet la consultation uniquement des vols vérifiant la condition donnée sur le pilote associé. Il est également possible de mettre à jour la relation VOL à travers cette vue mais l'opération de mise à jour peut concerner n'importe quel vol (sans aucune condition).

Par exemple supposons que le pilote n° 100 habite Paris, l'insertion suivante sera réalisée dans la relation VOL à travers la vue, mais le tuple ne pourra pas être visible en consultant la vue.

INSERT INTO VOLPIL_BAYONNE
(NUMVOL,NUMPIL,NUMAV,VILLE_DEP)
VALUES ('IT256', 100, 14, 'PARIS')

Si la clause **WITH CHECK OPTION** est présente dans l'ordre de création d'une vue, la table associée peut être mise à jour, avec vérification des conditions présentes dans la requête définissant la vue. La vue joue alors le rôle d'un filtre entre l'utilisateur et la table de base, ce qui permet la vérification de toute condition et notamment des contraintes d'intégrité.

Exemple : définir une vue permettant de consulter et de les mettre à jour uniquement les vols des pilotes habitant Bayonne.

CREATE VIEW VOLPIL_BAYONNE
AS
SELECT *
FROM VOL
WHERE NUMPIL **IN**
(**SELECT** NUMPIL
FROM PILOTE
WHERE ADRESSE = 'BAYONNE')
WITH CHECK OPTION

L'ajout de la clause **WITH CHECK OPTION** à la requête précédente interdira toute opération de mise à jour sur les vols qui ne sont pas assurés par des pilotes habitant Bayonne et l'insertion d'un vol assuré par le pilote n° 100 échouera.

Exemple : définir une vue sur PILOTE, permettant la vérification de la contrainte de domaine suivante : le salaire d'un pilote est compris entre 3.000 et 5.000.

```
CREATE VIEW DPILOTE
AS
SELECT *
FROM PILOTE
WHERE SALAIRE BETWEEN 3000 AND 5000
WITH CHECK OPTION
```

L'insertion suivante va alors échouer.

```
INSERT INTO DPILOTE (NUMPIL,SALAIRE) VALUES(175,7000)
```

Exemple : définir une vue sur vol permettant de vérifier les contraintes d'intégrité référentielle en insertion et en modification.

```
CREATE VIEW IMVOL
AS
SELECT *
FROM VOL
WHERE NUMPIL IN (SELECT NUMPIL FROM PILOTE)
AND NUMAV IN (SELECT NUMAV FROM AVION)
WITH CHECK OPTION
```

De manière similaire à une relation de la base, une vue peut être

- consultée via une requête d'interrogation ;
- décrite structurellement grâce à la commande **DESCRIBE** ou en interrogeant la table système ALL_VIEWS ;
- détruite par l'ordre **DROP VIEW** <nom_vue>.

III.3 Protection des données : contrôles d'accès

Comme Oracle est un SGBD multiutilisateurs, l'usager doit être identifié avant de pouvoir utiliser des ressources. L'accès aux informations et à la base de données doit être contrôlé à des fins de sécurité et de cohérence. Les Rôles et les privilèges sont définis pour sécuriser l'accès aux données de la base. Ces concepts sont mis en œuvre pour protéger les données en accordant (ou retirant) des privilèges a un utilisateur ou un groupe d'utilisateurs. Dans cette section, nous étudierons les aspects du langage SQL liés au contrôle des données et des accès. :

- **La gestion des utilisateurs :** à qui on associe des espaces de stockage (tablespaces) dans lesquels se trouveront leurs objets (table, index, séquences …).
- **La gestion des privilèges :** qui permettent de donner des droits sur la base de données (privilèges systèmes) et sur les données de la base (privilèges objets).
- **La gestion des rôles :** qui regroupent des privilèges système ou objets affectés par la suite à un ou plusieurs utilisateurs.

La gestion des utilisateurs

Un utilisateur est identifié au niveau de la base par son nom et peut se connecter puis accéder aux objets de la base sous réserve d'avoir reçu un certain nombre de privilèges.

Un schéma est une collection nommée (du nom de l'utilisateur qui en est en propriétaire) d'objets (tables, vues, séquences, index, procédures, …).

Les types d'utilisateurs, leurs fonctions et leur nombre peuvent varier d'une base à une autre. Néanmoins, pour chaque base de données en activité, on peut classifier les utilisateurs de la manière suivante :

- Le DBA (DataBase Administrator). Il en existe au moins un. parmi ses taches : la gestion de l'espace disque et des espaces pour les données (tablespaces) et la gestion des utilisateurs et leurs objets.
- Les utilisateurs qui se connectent et interagissent avec la base.

III.3.1 Création et suppression d'utilisateurs

Pour pouvoir accéder à ORACLE, un utilisateur doit être identifié en tant que tel par le système. Pour ce faire, il doit avoir un nom, un mot de passe et un ensemble de privilèges (ou droits).
Seul l'utilisateur qui possède le privilège CREATE USER peut créer des utilisateurs.

- **Création d'un utilisateur:**

  CREATE USER <nom_id> **IDENTIFIED BY** <password>;

- **Modification de mot de passe :**

  ALTER USER <nom_id> **IDENTIFIED BY** <new_password>
  REPLACE <old_password>;

- **Suppression d'un utilisateur:**

DROP USER <nom_id> [CASCADE]

CASCADE suppression des objets de l'utilisateur (Nécessaire sinon de les supprimer avant).

Lorsqu'un utilisateur est créé avec l'instruction **CREATE USER**, il ne dispose encore d'aucun droit car aucun privilège ne lui a encore été assigné. Il ne peut même pas se connecter à la base. Il faut donc lui assigner les privilèges nécessaires. Il doit pouvoir se connecter, créer des tables, des vues, des séquences.

III.3.2 La gestion des privilèges

Un privilège est un droit d'exécuter une certaine instruction SQL (on parle de privilège système), ou un droit d'accéder à un certain objet d'un autre schéma (on parle de privilège objet).

Privilèges systèmes

Il existe une centaine de privilèges système. Par exemple la création d'utilisateurs (CREATE USER), la création et la suppression de tables (CREATE /DROP TABLE), la création d'espaces (CREATE TABLESPACE). Pour assigner ces privilèges de niveau système il faut utiliser l'instruction **GRANT** dont voici la syntaxe :

GRANT privilègesystème | ALL PRIVILEGES **TO** *utilisateur* | *public*

[**IDENTIFIED BY** mot_passe**]**

[WITH ADMIN OPTION] ;

ALL PRIVILEGES : tous les privilèges système.

PUBLIC : pour attribuer le privilège à tous les utilisateurs.

IDENTIFIED BY : n'est obligatoire que lors de la création d'un nouvel utilisateur.

WITH ADMIN OPTION : permet d'attribuer aux bénéficiaires le droit de transmettre les privilèges reçus à une tierce personne.

Liste des privilèges système d'Oracle

Nom du privilege	Type d'action autorisée
ANALYZE	
ANALYZE ANY	Analyser toutes les tables, clusters, ou indexs dans la base de données.

AUDIT	
AUDIT ANY	Auditer tous les objets dans la base de données.
AUDIT SYSTEM	Auditer les actions de type DBA
CLUSTER	
CREATE CLUSTER	créer un cluster.
CREATE ANY CLUSTER	créer un cluster dans tous les schémas.
ALTER ANY CLUSTER	Modifier tous les clusters dans la base de données.
DROP ANY CLUSTER	Supprimer tous les clusters dans la base de données.
DATABASE	
ALTER DATABASE	Modifier la structure physique de la base
DATABASE LINK	
CREATE DATABASE LINK	Créer des databases links privés.
INDEX	
CREATE ANY INDEX	créer un index dans tous les schemas sur toutes les tables.
ALTER ANY INDEX	Modidier tous les index dans la base de données.
DROP ANY INDEX	Supprimer tous les index dans la base de données.
PRIVILEGE	
GRANT ANY PRIVILEGE	Donner tous les privileges système
PROCEDURE	
CREATE PROCEDURE	Créer des procedures stockées, fonctions, et packages
CREATE ANY PROCEDURE	Créer des procedures stockées, fonctions, et packages dans tous les schemas. (suppose ALTER ANY TABLE, BACKUP ANY TABLE, DROP ANY TABLE, SELECT ANY TABLE, INSERT ANY TABLE, UPDATE ANY TABLE, DELETE ANY TABLE, ou GRANT ANY TABLE.)
ALTER ANY PROCEDURE	Compiler toutes les procedures stockées, fonction, ou packages dans tous les schemas.
DROP ANY PROCEDURE	Supprimer toutes les procedures, fonction, ou package stockés dans tous les schemas.
EXECUTE ANY PROCEDURE	Executer toutes les procedures ou fonctions dans tous les schemas.

PROFILE	
CREATE PROFILE	Créer des profils.
ALTER PROFILE	Modifier tous les profils dans la base de données.
DROP PROFILE	Supprimer tous les profils dans la base de données.
ALTER RESOURCE COST	Modifier la ressource 'cost' dans toutes les sessions.
PUBLIC DATABASE LINK	
CREATE PUBLIC DATABASE LINK	Créer des database links publics.
DROP PUBLIC DATABASE LINK	Supprimer database links publics.
PUBLIC SYNONYM	
CREATE PUBLIC SYNONYM	Créer des synonyms publics.
DROP PUBLIC SYNONYM	Supprimer des synonyms publics.
ROLE	
CREATE ROLE	Créer des roles.
ALTER ANY ROLE	Modifier tous les roles dans la base de données.
DROP ANY ROLE	Supprimer tous les roles dans la base de données.
GRANT ANY ROLE	Grant tous les roles dans la base de données.
ROLLBACK SEGMENT	
CREATE ROLLBACK SEGMENT	Créer des rollback segments.
ALTER ROLLBACK SEGMENT	Modifier des rollback segments.
DROP ROLLBACK SEGMENT	Supprimer des rollback segments.
SESSION	
CREATE SESSION	Se connecter !!!
ALTER SESSION	faire des ALTER SESSION.
RESTRICTED SESSION	Se connecter malgré un démarrage 'RESTRICT'. (OSOPER et OSDBA donnent ce privilege.)
SEQUENCE	

CREATE SEQUENCE	crée une sequence dans son schema.
CREATE ANY SEQUENCE	Créer toutes les sequences dans tous les schemas.
ALTER ANY SEQUENCE	Modifier toutes les sequence dans tous les schémas.
DROP ANY SEQUENCE	Supprimer toutes les sequence dans tous les schémas.
SELECT ANY SEQUENCE	Reference toutes les sequence dans tous les schémas.
SNAPSHOT	
CREATE SNAPSHOT	Créer des snapshots (clichés) dans son schema. (l'utilisateur doit aussi avoir le privilege CREATE TABLE.)
CREATE SNAPSHOT	Créer des snapshots dans tous les schémas. (CREATE ANY TABLE nécessaire.)
ALTER SNAPSHOT	Modifier tous les snapshots dans tous les schémas.
DROP ANY SNAPSHOT	Supprimer tous les snapshots dans tous les schémas.
SYNONYM	
CREATE SYNONYM	créer un synonym dans son schema.
CREATE SYNONYM	Créer tous les synonyms dans tous les schémas.
DROP ANY SYNONYM	Supprimer tous les synonyms dans tous les schémas.
SYSTEM	
ALTER SYSTEM	faire des ALTER SYSTEM .
TABLE	
CREATE TABLE	Créer des tables ou des indexs dans son propre schéma
CREATE ANY TABLE	Créer des tables dans tous les schémas.
ALTER ANY TABLE	Modifier toutes les table dans tous les schémas et compiler toutes les vues dans tous les schémas.
BACKUP ANY TABLE	Réaliser des exports incrémentaux.
DROP ANY TABLE	Supprimer ou vider toutes les table dans tous les schémas.
LOCK ANY TABLE	Verrouiller toutes les tables ou vues dans tous les schémas.
COMMENT ANY TABLE	Commenter toutes les tables, vues, ou colonnes dans son schema.

SELECT ANY TABLE	Interroger toutes les tables, vues, ou clichés dans tous les schémas.
INSERT ANY TABLE	Insert rows into toutes les table ou view dans tous les schémas.
UPDATE ANY TABLE	Update rows in toutes les table ou view dans tous les schémas.
DELETE ANY TABLE	Delete rows from toutes les table ou view dans tous les schémas.
TABLESPACE	
CREATE TABLE SPACE	Créer tablespaces; add files to the operating system via Oracle, regardless of the l'utilisateur's operating system privileges.
ALTER TABLESPACE	Modifier tablespaces; add files to the operating system via Oracle, regardless of the l'utilisateur's operating system privileges.
MANAGE TABLESPACE	Take toutes les tablespace offline, bring toutes les tablespace online, et begin et end backups of toutes les tablespace.
DROP TABLESPACE	Supprimer tablespaces.
UNLIMITED TABLESPACE	Use an unlimited amount of *toutes les*tablespace. This privilege overrides toutes les specific quotas assigned. If revoked, the grantee's schema objects remain but further tablespace allocation is denied unless allowed by specific tablespace quotas. *This system privilege can be granted only to l'utilisateurs et not to roles. In general, specific tablespace quotas are assigned instead of granting this system privilege.*
TRANSACTION	
FORCE TRANSACTION	Fouce the commit ou rollback of own in-doubt distributed transaction in the local database.
FORCE ANY TRANSACTION	Fouce the commit ou rollback of toutes les in-doubt distributed transaction in the local database.
TRIGGER	
CREATE TRIGGER	crée un trigger in own schema.
CREATE ANY TRIGGER	Créer toutes les trigger dans tous les schémas associated with toutes les table dans tous les schémas.

ALTER ANY TRIGGER	Enable, disable, ou compile toutes les trigger dans tous les schémas.
DROP ANY TRIGGER	Supprimer toutes les trigger dans tous les schémas.
USER	
CREATE ANY USER	Créer l'utilisateurs; assign quotas on *toutes les* tablespace, set default et tempouary tablespaces, et assign a profile as part of a CREATE USER statement.
BECOME ANY USER	Become another l'utilisateur. (Required by toutes les l'utilisateur perfouming a full database impout.)
ALTER USER	Modifier other l'utilisateurs: change toutes les l'utilisateur's passwoud ou authentication method, assign tablespace quotas, set default et tempouary tablespaces, assign profiles et default roles, in an ALTER USER statement. (Not required to alter own passwoud.)
DROP USER	Supprimer another l'utilisateur.
VIEW	
CREATE VIEW	crée un view in own schema.
CREATE ANY VIEW	crée un view dans tous les schémas. (Requires that l'utilisateur also have ALTER ANY TABLE, BACKUP ANY TABLE, DROP ANY TABLE, LOCK ANY TABLE, COMMENT ANY TABLE, SELECT ANY TABLE, INSERT ANY TABLE, UPDATE ANY TABLE, DELETE ANY TABLE, ou GRANT ANY TABLE.)
DROP ANY VIEW	Supprimer toutes les view dans tous les schémas.

Exemple : droit de création de table

GRANT CREATE TABLE TO nom_utilisateur ;

Puis les droits de création de vues

GRANT CREATE VIEW TO nom_utilisateur ;

Et il en va de même pour tous les autres privilèges qui lui sont assignés.

L'ensemble de ces privilèges peuvent être assignés au sein d'une même commande

GRANT CREATE SESSION, CREATE TABLE, CREATE VIEW TO nom_utilisateur ;

Exemple :

- **GRANT** CREATE SESSION , CREATE SEQUENCE TO Hicham;

Hicham peut se connecter à la base et il peut créer des séquences.

- **GRANT** CREATE TABLE TO Hicham WITH ADMIN OPTION;

Hicham peut créer des tables dans son schéma et peut retransmettre ce privilège à un tiers.

- **GRANT** CREATE SESSION , CREATE ANY TABLE, DROP ANY TABLE TO Ahmed;

Ahmed peut se connecter à la base, créer et détruire des tables dans tout schéma.

- **Retrait de privilège système**

Un utilisateur d'ORACLE peut être supprimé à tout moment ou se voir démuni de certains privilèges. La commande correspondante est :

REVOKE privilègesystème **| ALL PRIVILEGES**
FROM utilisateur | PUBLIC

ALL PRIVILEGES : si l'utilisateur ou le rôle ont tous les privilèges système.
PUBLIC : pour annuler le privilège à chaque utilisateur ayant reçu ce privilège par l'option public.

Exemple :

- **REVOKE** CREATE SESSION **FROM** Hicham, Ahmed;

Hicham et Ahmed ne peuvent plus se connecter à la base.

- **REVOKE** ALL PRIVILEGES **FROM** Ahmed;

Commande incorrect car Ahmed n'a pas reçu tous les privilèges système.

Privilèges objets :

Les privilèges objets sont relatifs aux données de la base et aux actions sur les objets (table, vue, séquence, procédure). Chaque type d'objet a différents privilèges associés comme l'indique le tableau suivant : Permettent les manipulations sur des objets spécifiques. Les privilèges SELECT, INSERT, UPDATE, DELETE.

Privilège	table	vue	séquence	programme PL/SQL
ALTER			×	
DELETE	×	×		
EXECUTE				×
INDEX	×			
INSERT	×	×		
REFERENCES	×			
SELECT	×	×	×	
UPDATE	×	×		

Attribution de privilèges objets :

L'instruction GRANT permet d'attribuer un (ou plusieurs) privilèges à un (ou plusieurs) objet à un (ou plusieurs) bénéficiaire. L'utilisateur qui exécute cette commande doit avoir lui-même le droit de transmettre ces privilèges sauf s'il s'agit de ses propres objets pour lesquels il possède automatiquement les privilèges avec l'option GRANT OPTION. La syntaxe est la suivante :

GRANT privilègeobjet | ALL PRIVILEGES [(colonne 1, colonne 2)]

ON SCHEMA.nomobjet

TO utilisateur | PUBLIC

[WITH GRANT OPTION]

Privilègeobjet: description du privilège objet (select, delete…).

Colonne : précise la colonne ou les colonnes sur lesquelles se porte le privilège.

ALL PRIVILEGES: donne tout les privilèges.

PUBLIC: pour attribuer le privilège à tous les utilisateurs.

WITH GRANT OPTION: permet de donner aux bénéficiaires le droit de retransmettre les privilèges reçus à une tierce personne.

Exemple : assigner à l'utilisateur le droit de sélectionner, insérer, modifier et supprimer des lignes dans la table ETUDIANTS de l'utilisateur SMI

GRANT SELECT, INSERT ,UPDATE ,DELETE ON SMI.ETUDIANTS **TO** nom_utilisateur ;

Une liste de colonnes peut être indiquée dans l'instruction afin de restreindre davantage les droits sur une table

GRANT UPDATE (note, moy) **ON** SMI.ETUDIANTS **TO** nom_utilisateur ;

L'utilisateur peut modifier la table SMI.ETUDIANTS mais uniquement les colonnes note et moy.

Exemple :

GRANT REFERENCES (brevet), UPDATE (nom,age), SELECT

ON Hicham.Pilote

TO Ahmed

Affectation des privilèges de lecture de la table Pilote dans le schéma de Hicham, de modification des colonnes nom et âge et de référence à la clé primaire brevet à l'utilisateur Ahmed.

Retrait de privilège objet :

Pour pouvoir révoquer un privilège objet, il faut détenir au préalable cette permission ou avoir reçu le privilège système ANY OBJECT PRIVILEGE. Il n'est pas possible d'annuler un privilège objet qui a été accordé avec l'option WITH GRANT OPTION.

REVOKE [**GRANT OPTION FOR**] Priv1,Priv2 …

ON objet

FROM utilisateur, utilisateur2… | PUBLIC
[**RESTRICT** | **CASCADE**]

GRANT OPTION FOR : retrait uniquement du droit de transfert.
CASCADE : retrait des droits accordés à des tiers (ADMIN option) par celui à qui on retire le privilège
RESTRICT : refus du retrait en cascade

Exemple:

REVOKE UPDATE(nom,age), SELECT ON Hicham.Pilote FROM Ahmed;
Ahmed ne peut plus modifier ni lire la table Pilote de l'utilisateur Hicham.

- **Principes généraux appliqués aux privilèges**

  - Un utilisateur possède automatiquement tous les privilèges sur un objet qui lui appartient.
  - Un utilisateur ne peut pas donner plus de privilèges qu'il n'en a reçus.
  - S'il n'a pas reçu le privilège avec l'option WITH GRANT OPTION, un utilisateur ne peut pas assigner à son tour ce même privilège.

III.3.3 rôles

Un rôle est un regroupement de privilèges (système ou objets). Un rôle est accordé à un ou plusieurs utilisateurs, voire à tous (PUBLIC). Ce mécanisme facilite la gestion des privilèges.

- **Créer des rôles et leur assigner des privilèges**

L'instruction **GRANT** permet d'assigner un ou plusieurs privilèges système ou objet. Cependant, lorsque la liste des privilèges est importante, cette manière de procéder s'avère rapidement fastidieuse et répétitive. Si l'on prend l'exemple d'un utilisateur travaillant au service comptabilité, il doit recevoir un certain nombre de privilèges sur un certain nombre d'objets. Un autre utilisateur du même service à toutes les chances de se voir assigner des privilèges identiques. C'est pourquoi il est souhaitable de pouvoir regrouper des privilèges identiques dans un même ensemble.

Un Rôle permet :

• Regroupement d'utilisateurs qui partagent les mêmes privilèges

• Association d'un rôle à chaque utilisateur

• Attribution de privilèges à chaque rôle

- **Création /suppression de rôles**

  CREATE ROLE nom_role;

  DROP ROLE nom_role;

Il n'est pas possible de donner le privilège de REFERENCES à un rôle.

- **Modification de rôle**

  ALTER ROLE nom_role ;

- **Association d'un rôle à un utilisateur**

  GRANT nom_role **TO** utilisateur ;

Exemple:

CREATE ROLE comptabilite ;

GRANT SELECT, INSERT, UPDATE, DELETE ON CPT.FACTURE TO comptabilite;
GRANT SELECT, INSERT, UPDATE, DELETE ON CPT.LIG_FAC TO comptabilite;
GRANT SELECT, INSERT, UPDATE, DELETE ON CPT.JOURNAL TO comptabilite;

Une fois le rôle créé, il peut être assigné à un utilisateur.

GRANT comptabilite TO nom_utilisateur ;

Exercice :

1. Créer un utilisateur service_client.
2. Attribuer les droits nécessaires à l'utilisateur pour pouvoir :
   - Créer les tables
   - Modifier les tables
   - Supprimer les tables

Partie II : Langage PL/SQL

Introduction :

PL/SQL est un langage structuré en blocs, constitués d'un ensemble d'instructions SQL. Il est préférable de travailler avec un bloc PL /SQL plutôt qu'avec une suite d'instructions SQL susceptibles d'encombrer le trafic réseau. En effet, un bloc PL/SQL donne lieu à un seul échange sur le réseau entre le client et le serveur. Tandis que chaque instruction SQL donne lieu à l'envoi d'un message du client vers le serveur suivi de la réponse du serveur vers le client.

I. Bloc PL/SQL :

Chaque bloc PL/SQL peut être constitué de 3 sections :

• Une section facultative de déclaration et initialisation de types, variables et constantes

• Une section obligatoire contenant les instructions d'exécution

• Une section facultative de gestion des erreurs

[DECLARE | PROCEDURE name IS | FUNCTION name RETURN datatype IS]
- - déclaration et initialisation
BEGIN
- - instructions exécutables
[EXCEPTION]
- - gestion des erreurs
END;

Exemple :

Un bloc PL/SQL peut être représenté de la façon suivante :

```
DECLARE
Chaine VARCHAR2(15) := 'Hello World' ;
BEGIN
DBMS_OUTPUT.PUT_LINE(Chaine ) ;
EXCEPTION
When OTHERS then
Null ;
END ;
```

Le mot clé **BEGIN** détermine le début de la section des instructions exécutables.

Le mot clé **END;** indique la fin de la section des instructions exécutables.

La section déclarative (facultative) d'un bloc débute par le mot clé **DECLARE.** Elle contient toutes les déclarations des variables qui seront utilisées localement par la section exécutable, ainsi que leur éventuelle initialisation.

Une variable *Chaine* est déclarée de type VARCHAR2(15) et initialisée avec la valeur 'Hello World'.

Dans la section exécutable, cette variable est transmise à la fonction DBMS_OUTPUT.PUT_LINE() pour être affichée à l'écran.

Toute variable doit être déclarée avant d'être utilisée dans la section exécutable.

La section de gestion des erreurs (facultative) débute par le mot clé **EXCEPTION.** Elle contient le code exécutable mis en place pour la gestion des erreurs. Lorsqu'une erreur intervient dans l'exécution, le programme est stoppé et le code erreur est transmis à cette section.

Les erreurs doivent être interceptées avec le mot clé **WHEN** suivi du code erreur ciblé. Ici, le code **OTHERS** définit toutes les erreurs non interceptées individuellement par les clauses **WHEN** précédentes.

I.1 La section déclarative

Dans cette section tous les types, variables et constantes nécessaires à l'exécution du bloc doivent être déclarés. Ces variables peuvent être de n'importe quel type SQL ou PL/SQL.

Leur initialisation, facultative, s'effectue avec :

- **l'opérateur :=**

  DECLARE

  LN_Nbre **NUMBER**(3) := 0 ;

  LD_**Date DATE** := **SYSDATE** ;

  LC_Nom **VARCHAR2**(10) := 'PL/SQL' ;

- **Le mot clé CONSTANT** : une constante est une variable dont l'initialisation est obligatoire et dont la valeur ne pourra pas être modifiée en cours d'exécution. Elle est déclarée avec le mot clé constant qui doit précéder le type.

  DECLARE

  LN_Pi **CONSTANT NUMBER** := 3. 1415926535 ;

- **Le mot-clé DEFAULT :** permet d'initialiser des variables sans utiliser l'opérateur d'affectation comme dans l'exemple suivant :

  LN_Pi NUMBER DEFAULT 3.1415926535;

- **La directive INTO d'une requête :** consiste à utiliser une commande SQL SELECT qui définit une valeur de la base de données comme suit :
  SELECT brevet
  INTO v_brevet
  FROM Pilote
  WHERE nom = 'Gratien Viel';

- **Déclaration de variable par référence aux données de la base**

Usage du type de donné par référence au schéma (ligne) ou à un attribut (colonne).

- Attribut **%TYPE :** est utilisé pour déclarer une variable en fonction d'une définition de colonne de base de données.

  Exemple:
  DECLARE
  v_brevet Pilote.brevet%TYPE;

v_brevet est une variable PLSQL de même type que la colonne *brevet* de la table *Pilote*.
Utile pour garantir la compatibilité des affectations.

- Attribut **%ROWTYPE :** est utilisé pour déclarer une variable correspondant à l'ensemble des colonnes d'une table ou d'une vue d'une base de données.

Exemple :
DECLARE
piloteRec Pilote%ROWTYPE;

Les noms et les types de données des champs de l'enregistrement sont issus des colonnes de la table *Pilote.*

Remarque : comme SQL, PL/SQL n'est pas sensible à la casse. Pour lui les expressions suivantes sont équivalentes :
NOM_VARIABLE NUMBER ;

Nom_Variable Number ;
nom_variable number ;

I. 2 La section exécution

Délimitée par les mots clé **BEGIN** et **END;** elle contient les instructions d'exécution du bloc PL/SQL, les instructions de contrôle et d'itération, l'appel des procédures et fonctions, l'utilisation des fonctions natives, les ordres SQL, etc. Chaque instruction doit être suivi du terminateur d'instruction **';'.** PL/SQL ne permet pas les commandes SQL langage de définition comme la création de table, cependant, il permet tous les autres types de commandes SQL (*insert, delete, update, commit* ...).
La commande ***select*** (sauf si imbriqué) est **toujours** utilisé avec ***into***, pour affecter les attributs retrouvés aux variables PL/SQL.

II. Types des données

Un aspect important d'un langage est la manière de définir les variables. Une fois que les variables sont définies, PL/SQL permet de les utiliser dans des commandes SQL ou dans d'autres commandes du langage. La définition de constantes au sein de PL/SQL suit les mêmes règles. Il faut définir les variables et les constantes avant de les référencer dans une autre construction.
Tout type de donnée de PL/SQL ou SQL est un type valide dans une définition de variable. Les types de données les plus utilisés sont VARCHAR2, DATE, NUMBER (types de SQL), BOOLEAN et BINARY_INTEGER, TABLE, RECORD (types de PL/SQL).

II.1 Types de données scalaires

PL/SQL supporte une vaste gamme de types de données scalaires pour définir des variables et des constantes. A la différence des types de données composites, les types de données scalaires n'ont pas de composantes accessibles. Ces types de données tombent dans l'une des catégories suivantes :

- Booléen
- Date/heure
- Caractère
- Nombre

Chaque catégorie va maintenant être étudiée de plus prées.

- **Booléen**

Le type de données BOOLEAN, qui ne prend pas de paramètres, est utilisé pour stocker une valeur binaire, TRUE ou FALSE. Ce type de donnée peut aussi stocker la non-valeur NULL.

- **Date/Heure**

Le type de données DATE, qui ne prend aucun paramètre, est utilisé pour stocker des valeurs de dates. Ces valeurs de date incluent l'heure lorsqu'elles sont stockées dans une colonne de la base de données. Les valeurs par défaut pour le type de données DATE sont les suivantes :
Date : premier jour du mois courant
Heure : minuit

- **Caractère**

Le type de données caractère inclut CHAR, VARCHAR2, LONG, RAW et LONG RAW. CHAR est destinée aux données de type caractère de longueur fixe et VARCHAR2 stocke des données de type caractère de longueur variable.
LONG stocke des chaines de longueur variable ; RAW et LONG RAW stockent des données binaires ou des chaines d'octets. Les types de données CHAR, VARCHAR2 et RAW prennent un paramètre optionnel pour spécifier la longueur : type(max_len).

- **Nombre**

Il y a deux types de données dans la catégorie des nombres : BINARY_INTEGER et NUMBER. BINARY_INTEGER stocke des entiers signés sur l'étendue de -2^{31} à $2^{31} - 1$. L'utilisation la plus courante de ce type de donnée est celle d'un index pour des tables PL/SQL. Le stockage de nombres de taille fixe ou flottants de n'importe quelle taille est possible avec le type de donnée NUMBER. Pour des nombres flottants, la précision et l'échelle peuvent être spécifiées avec le format suivant :

```
NUMBER(10,2)
```

Une variable déclarée de cette manière a un maximum de 10 chiffres et l'arrondi se fait à deux décimales.

II.2 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.

- **Tableaux**

Les variables de type TABLE permettent de définir et de manipuler des tableaux dynamiques (car définis sans dimension initiale). Un tableau est composé d'une clé primaire (de type BINARY_INTEGER) et d'une colonne (de type scalaire, TYPE, ROWTYPE ou RECORD) pour stocker chaque élément. La syntaxe générale pour déclarer un type de tableau et une variable tableau est la suivante :

```
TYPE nomTypeTableau IS TABLE OF {typeScalaire | table.colonne%TYPE | table%ROWTYPE}

[INDEX BY BINARY_INTEGER];

nomTableau nomTypeTableau ;
```

Exemple:

```
DECLARE
TYPE brevets_tytab is TABLE OF VARCHAR2(6)
        INDEX BY BINARY_INTEGER ;
TYPE nomPilotes_tytab IS TABLE OF Pilote.nom%TYPE
        INDEX BY BINARY_INTEGER;
TYPE pilotes_tytab IS TABLE OF Pilote%ROWTYPE
        INDEX BY BINARY_INTEGER;
tab_brevets  brevets_tytab ;
tab_nomPilotes  nomPilotes_tytab ;
tab_pilotes   pilotes_tytab ;
```

Constructions de tableaux : Après leur déclaration, les tableaux sont disponibles pour le stockage d'information. Pour stocker de l'information dans les tableaux définis dans l'exemple précédent, il suffit de référencer le tableau avec une valeur numérique. Cela est illustré dans l'exemple suivant :

```
BEGIN
   Tab_brevets(-1) :='PL-1' ;
   Tab_brevets(-2) :='PL-2' ;
   tab_nomPilotes  (7800) :='bilal' ;
   tab_Pilotes(0).brevet :='Pl-0' ;
END ;
```

- **Enregistrements**

Bien que la directive %ROWTYPE permet de déclarer une structure composée de colonnes de tables, elle ne convient pas à des données personnalisées. Le type de données RECORD définit des données personnalisées (l'équivalent de *struct* en C). La syntaxe générale pour déclarer un RECORD est la suivante :

```
TYPE nomRecord IS RECORD
   ( nomChamp typeDonnée [NOT NULL]  { := |  DEFAULT} expression]
    [, nomChamp type Donnée…]…);
```

Exemple:

```
DECLARE
   TYPE  avionAirbus_rec IS RECORD
      (nserie CHAR(10), nomAvion CHAR(20), usine CHAR(10) :='Blagnac',
      nbHVol NUMBER(7,2)) ;/*Définition du type RECORD contenant quatre
                                champs ;initialisation du champ usine */
r_unA320  avionAirbus_rec ;       --déclaration de deux variables de type RECORD
```

```
r_FGLFS   avionAirbus_rec ;
BEGIN
        r_unA320.nserie :='A1' ;    --Initialisation des champs d'un RECORD
        r_unA320.nomAvion :='A320-200' ;
        r_unA320.nbHVol :=2500 ;
        r_FGLFS :=r_unA320 ;     --Affectation d'un RECORD
        …
```

Les types RECORD ne peuvent pas être stockés dans une table. En revanche, il est possible qu'un champ d'un RECORD soit lui-même un RECORD, ou soit déclaré avec les directives %TYPE ou %ROWTYPE.

III. Curseurs

Un curseur est une zone de mémoire utilisée par Oracle pour récupérer les résultats de requêtes SQL qui retourne plusieurs enregistrements. Il existe trois types de curseur :

1. **Curseurs implicites** : créés et gérés en interne par le serveur Oracle afin de traiter les instructions du MLD de SQL (INSERT, UPDATE et DELETE). Ce curseur porte le nom de SQL et il est exploitable après avoir exécuté l'instruction. La commande qui suit le LMD remplace l'ancien curseur par un nouveau. Les attributs de curseurs implicites permettent de connaitre un certain nombre d'informations qui ont été renvoyées après l'instruction du LMD et qui peuvent être utiles au programmeur.

SQL%ROWCOUNT : nombre de lignes affectées par la dernière instruction LMD.

SQL%FOUND : Booléen valant TRUE si la dernière instruction LMD affecte au moins un enregistrement.

SQL%NOTFOUND : Booléen valant TRUE si la dernière instruction LMD n'affecte aucun enregistrement.

Exemple:

```
DECLARE
g_pilotesAFDetruits NUMBER ;
BEGIN
```

```
DELETE FROM Pilote WHERE compa='AF';
g_pilotesAFDetruits :=SQL%ROWCOUNT ;

DBMS_OUTPUT.PUT_LINE(g_pilotesAFDetruits) ;

END ;
```

2. **Curseurs explicites** : déclarés explicitement par le programmeur, ils sont associés à une seule requête SELECT. Ils se déclarent comme suit :

```
CURSOR nom_ curseur IS SELECT col1, col2, ... FROM nom_table WHERE ...;
```

Exemple:

```
CURSOR zone IS SELECT brevet, nbhVol, comp FROM Pilote WHERE comp='AF';
```

Pour exploiter les résultats d'un curseur, on utilise les instructions suivantes:

- OPEN nomCurseur : Ouverture du curseur (chargement des lignes).
- FETCH nomCurseur INTO listeVariables | nomRECORD : Positionnement sur la ligne suivante et chargement de l'enregistrement courant dans une ou plusieurs variables.
- CLOSE nomCurseur : Ferme le curseur. L'exception INVALID_CURSOR se déclenche si des accès au curseur sont opérés après sa fermeture.

Les curseurs explicites peuvent avoir les attributs suivants :

nomCurseur%NOTFOUND : Retourne TRUE si le dernier FETCH n'a retourné aucun enregistrement.

nomCurseur%FOUND : Retourne TRUE si le dernier FETCH a retourné un enregistrement.

nomCurseur%ISOPEN : Retourne TRUE si le curseur est ouvert, FALSE sinon.

nomCurseur%ROWCOUNT : Retourne le nombre d'enregistrements trouvés jusqu'à présent.

- **Parcours d'un curseur :** suivant le traitement à effectuer sur le curseur à parcourir, on peut utiliser une structure répétitive *tant que* (while), *répéter* (loop), *pour* (for) et à l'aide d'une condition généralement *nomCurseur%NOTFOUND* ou *nomCurseur%FOUND*.

NB : avant la première extraction, *nomCurseur%NOTFOUND* renvoie toujours NULL. Si l'instruction FETCH ne parvient jamais à s'exécuter correctement, la boucle *répéter* devient infinie. Il est conseillé de programmer la sortie d'une structure *répéter* à l'aide de

la condition composée : EXIT WHEN nomCurseur%NOTFOUND OR nomCurseur%NOTFOUND IS NULL.

Exemple : répéter (loop)

```
DECLARE
  CURSOR curseur1 IS SELECT brevet, nbHVol, comp FROM Pilote WHERE
  comp=’AF’;
  var1 Pilote.brevet%TYPE;
  var2 Pilote.nbHVol%TYPE;
  var3 Pilote.comp%TYPE;
  totalHeures NUMBER:=0;
BEGIN
  OPEN curseur1;
  LOOP
  FETCH curseur1 INTO var1, var2, var3;
  EXIT WHEN curseur1%NOTFOUND;
  totalHeures := totalHeures + var2 ;
  END LOOP;
  CLOSE Curseur1;
END;
```

Exemple : tant que (while)

```
DECLARE
  CURSOR curseur1 IS SELECT brevet, nbHVol, comp FROM Pilote WHERE
  comp=’AF’;
  var1 Pilote.brevet%TYPE;
  var2 Pilote.nbHVol%TYPE;
  var3 Pilote.comp%TYPE;
  totalHeures NUMBER:=0;
BEGIN
  OPEN curseur1;
  FETCH curseur1 INTO var1, var2, var3;
  WHILE (curseur1%FOUND) LOOP
     totalHeures := totalHeures + var2 ;
     FETCH curseur1 INTO var1, var2, var3;
```

```
  END LOOP;
  CLOSE Curseur1;
END;
```

3. **Curseurs dynamiques :** Un curseur dynamique est un curseur qui n'est pas associé à une seule requête SQL. Dans ce cas, lors de sa déclaration, il faudra utiliser le mot réservé **REF.**

```
TYPE nomTypeCurDynamiq is REF CURSOR [RETURN typeRetourSQL];
nomCurDynamique nomTypeCurDynamiq;
```

- Un REF CURSOR est vu comme un pointeur sur une zone mémoire dans le serveur (contenant le résultat d'une requête).
- Il peut retourner des valeurs ou non.
- L'ouverture d'un curseur dynamique est commandée par l'instruction OPEN FOR requête.
- La lecture du curseur s'opère toujours avec l'instruction FETCH.

Exemple : Curseur non typé

```
DECLARE
    TYPE ref_zone IS REF CURSOR ;
    zone ref_zone ;
    var1 Pilote.brevet%TYPE ;
    var2 Pilote.nom%TYPE ;
    var3 Pilote.nbHVol%TYPE ;

BEGIN
    OPEN zone FOR SELECT brevet, nom, FROM Pilote WHERE NOT (comp= 'AF ');
    FETCH zone INTO var1, var2;
    WHILE (zone%FOUND) LOOP
        DBMS_OUTPUT.PUT_LINE('nom: ' || var2 || ' (' || var1 || '). ');
        FETCH zone INTO var1, var2;
    END LOOP;
    CLOSE zone;
END;
```

Exemple : Curseur typé

```
DECLARE
    TYPE ref_zone IS REF CURSOR RETURN Pilote%ROWTYPE ;
    zone ref_zone ;
    enreg zone%ROWTYPE ;

BEGIN
    OPEN zone FOR SELECT * FROM Pilote WHERE NOT (comp= 'AF ');
    FETCH zone INTO enreg;
    WHILE (zone%FOUND) LOOP
        DBMS_OUTPUT.PUT_LINE('nom: ' || enreg.nom || ' ('  || enreg.comp || '). ');
        FETCH zone INTO enreg;
    END LOOP;
    CLOSE zone;
END;
```

IV Structures de contrôle

Tout langage procédural a des structures de contrôle qui permettent de traiter l'information d'une manière logique en contrôlant le flot des informations. Les structures disponibles au sein de PL/SQL incluent :

- Les structures conditionnelles *si* et *cas* (IF…et CASE) ;
- Les structures répétitives *tant que*, *répéter* et *pour* (WHILE, LOOP, FOR).

IV.1 Structures conditionnelles

PL/SQL propose deux structures pour programmer une action conditionnelles : la structure IF et la structure CASE.

- **Structure IF** : suivant les tests à programmer, on peut distinguer trois formes de structures IF : IF-THEN, IF-THEN-ELSE et IF-THEN-ELSIF. La syntaxe est la suivante :

1. IF-THEN

```
IF condition THEN
   Instructions ;
```

```
        END IF;
```

2. IF-THEN-ELSE

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

3. IF-THEN-ELSIF

```
        IF condition1 THEN
          instructions;
        ELSIF condition2
          instructions;
        ELSE
           instructions;
        END IF;
```

Exemple:

```
DECLARE
    v_telephone CHAR(14) NOT NULL:='06-76-85-14-89';

BEGIN
    IF SUBSTR(v_telephone,1,2)='06' THEN
      DBMS_OUTPUT.PUT_LINE(' C '' est un portable') ;
    ELSE
      DBMS_OUTPUT.PUT_LINE(' C '' est un fixe') ;
    END IF ;
END ;
```

- **Structure CASE :** comme l'instruction IF, la structure CASE permet d'exécuter une séquence d'instructions en fonction de différentes conditions.la structure CASE est utile lorsqu'il faut évaluer une même expression et proposer plusieurs traitements pour diverses conditions.

  En fonction de la nature de l'expression et des conditions, une des deux écritures suivantes peut être utilisée :

1)

```
CASE variable
   WHEN expr1 THEN instructions1;
   WHEN expr2 THEN instructions2;
   …
   WHEN exprN THEN instructionsN;
      [ELSE instructionsN+1];
   END CASE;
```

2)

```
CASE
   WHEN condition1 THEN instructions1;
   WHEN condition 2 THEN instructions2;
   …
   WHEN condition N THEN instructionsN;
      [ELSE instructionsN+1];
   END CASE;
```

Exemple 1:

```
DECLARE
      grade CHAR(1) ;
      appraisal VARCHAR2(20) ;
BIGIN
      Appraisal :=
            CASE grade
                  WHEN ’A’ THEN ‘Excellent’;
                   WHEN ‘B’ THEN ‘Very good’;
                   WHEN ‘C’ THEN ‘Good’;
                    ELSE ‘No such grade’
            END CASE ;
       DBMS_OUTPUT.PUT_LINE(‘grade:’ || grade || ‘appraisal:’ ||
      appraisal);
 END;
```

Exemple 2:

```
DECLARE
```

```
        grade CHAR(1) ;
        appraisal VARCHAR2(20) ;
    BIGIN
        Appraisal :=
            CASE
                WHEN grade='A' THEN 'Excellent';
                WHEN grade IN ('B','C') THEN 'Good';
                ELSE 'No such grade'
            END case;
        DBMS_OUTPUT.PUT_LINE('grade:' || grade || 'appraisal:' ||
        appraisal);
    END;
```

IV.2 Structures répétitives

Les trois structures répétitives *tant que*, *répéter* et *pour utilisent* l'instruction *LOOP… END LOOP*.

- **Structure tant que :** avant chaque itération (et notamment avant la première), la condition est évaluée. Si elle est vraie, la séquence d'instructions est exécutée, puis la condition est réévaluée pour un éventuel nouveau passage dans la boucle. Ce processus continu jusqu'à ce que la condition soit fausse pour passer en séquence après le END LOOP. Quand la condition n'est jamais fausse, on dit que le programme boucle. La syntaxe est la suivante :

```
WHILE condition LOOP
        Instructions ;
END LOOP ;
```

Exemple :

```
DECLARE
   v_somme  NUMBER(4) :=0 ;
   v_entier    NUMBER(3) :=1 ;
BEGIN
  WHILE (v_entier  <=100) LOOP
   v_somme := v_somme +  v_entier ;
```

```
    v_entier := v_entier +1 ;
  END LOOP ;
 DBMS_OUTPUT.PUT_LINE('somme=' || v_somme);
END;
```

- **Structure répéter :** se programme à l'aide de la syntaxe LOOP EXIT suivante :

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

La particularité de cette structure est que la première itération est effectuée quelles que soient les conditions initiales. La condition n'est évaluée qu'en fin de boucle.

- Si aucune condition n'est spécifié (WHEN condition absent), la sortie de la boucle est immédiate dès la fin des instructions.
- Si la condition est fausse, la séquence d'instructions est de nouveau exécutée. Ce processus continue jusqu'à ce que la condition soit vraie pour passer en séquence après le END LOOP.
- Quand la condition n'est jamais fausse, on dit que le programme boucle.

Exemple :

```
DECLARE
    v_somme  NUMBER(4) :=0 ;
    v_entier    NUMBER(3) :=1 ;
BEGIN
  LOOP
    v_somme := v_somme +  v_entier ;
    v_entier := v_entier +1 ;
   EXIT WHEN  v_entier >100;
  END LOOP ;
 DBMS_OUTPUT.PUT_LINE('somme=' || v_somme);
END;
```

Cette structure doit être utilisée quand il n'est pas nécessaire de tester la condition avec les données initiales avant d'exécuter les instructions contenues dans la boucle.

- **Structure pour :** célèbre pour les parcours de vecteurs, tableaux et matrices en tout genre, la structure *pour* se caractérise par la connaissance a priori du nombre d'itérations que le programmeur souhaite faire effectuer pour son algorithme. La syntaxe est la suivante :

```
FOR compteur IN [REVERSE]  valeurInf..valeurSup LOOP
        Instructions ;
END LOOP ;
```

A la première itération le compteur reçoit automatiquement la valeur initiale (valeurInf). Après chaque itération le compteur est incrémenté (ou décrémenté si l'option REVERSE a été choisie). La sortie de la boucle est automatique après l'itération correspondant à la valeur finale du compteur (valeurSup). La déclaration de la variable compteur n'est pas obligatoire.

Exemple :

```
DECLARE
    v_somme  NUMBER(4) :=0 ;
BEGIN
  FOR v_entier IN 1..100 LOOP
   v_somme := v_somme +  v_entier ;
  END LOOP ;
DBMS_OUTPUT.PUT_LINE('somme='  ||  v_somme);
END;
```

V Exceptions

Afin d'éviter qu'un programme s'arrête à la première erreur (requête ne retournant aucune ligne, valeur incorrecte à écrire dans la base, conflit de clé primaire, division par zéro…), il est indispensable de prévoir tous les cas potentiels d'erreurs et d'associer à chacun de ces cas la programmation d'une exception PL/SQL. Cette exception correspond à une valeur d'erreur et est associée à un identificateur. Une fois levée, l'exception termine le corps principal des instructions et renvoie au bloc EXCEPTION du programme en question. Deux mécanismes peuvent déclencher une exception :

- Une erreur Oracle se produit, l'exception associée est déclenchée automatiquement (exemple du SELECT ne ramenant aucune ligne, ce qui déclenche l'exception ORA-01403 d'identificateur NO_DATA_FOUND).

- Le programmeur désir dérouter volontairement (par l'intermédiaire de l'instruction RAISE) son programme dans le bloc des exceptions sous certaines conditions. L'exception est ici manuellement déclenchée et peut appartenir à l'utilisateur (dans l'exemple suivant PILOTE_TROP_JEUNE) ou être prédéfinie au niveau d'ORACLE (division par zéro d'identificateur ZERO_DIVIDE qui sera automatiquement déclenchée.

```
BEGIN
…..
IF (…) THEN RAISE PILOTE TROP JEUNE;
SELECT … INTO ….FROM…;
…
EXCEPTION
WHEN NO DATA FOUND THEN
    Instructions –A
WHEN ZERO DIVIDE THEN
    Instructions –B
WHEN PILOTE TROP JEUNE THEN
    Instructions –C
WHEN OTHERS THEN
    Instructions –D
END;
```

Si aucune erreur ne se produit, le bloc est ignoré et le traitement se termine.

La syntaxe générale d'un bloc d'exception est la suivante. Il est possible de grouper plusieurs exceptions pour programmer le même traitement. La dernière entrée (OTHERS) doit être toujours placée en fin du bloc d'erreurs.

```
EXCEPTION
    WHEN exception1 [OR exception2 …]  THEN
        Instructions;
    WHEN exception3 [OR exception4 …]  THEN
        Instructions;
    WHEN OTHERS [OR exception2 …]  THEN
        Instructions;
```

Si une anomalie se produit, le bloc EXCEPTION s'exécute.

- Si le programme prend en compte l'erreur dans une entrée WHEN…, les instructions de cette entrée sont exécutées et le programme se termine.

- Si l'exception n'est pas prise en compte dans le bloc EXCEPTION, il existe une section OTHERS où des instructions s'exécutent.

V.1 Exceptions définies par le système

Des exceptions internes à PL/SQL sont levées automatiquement en cas d'erreur. Dans le précédent exemple, NO_DATA_FOUND est une exception définie par le système. La table suivante est une liste complète des exceptions internes.

Nom de l'exception	Erreur Oracle
CURSOR_ALREADY_OPEN	ORA-06511
DUP_VAL_ON_INDEX	ORA-00001
INVALID_CURSOR	ORA-01001
INVALID_NUMBER	ORA-01722
LOGIN_DENIED	ORA-01017
NO_DATA_FOUND	ORA-01403
NOT_LOGGED_ON	ORA-01012
PROGRAM_ERROR	ORA-06501
STORAGE_ERROR	ORA-06500
TIMEOUT_ON_RESOURCE	ORA-00051
TOO_MANY_ROWS	ORA-01422
TRANSACTION_BACKED_OUT	ORA-00061
VALUE_ERROR	ORA-06502
ZERO_DIVIDE	ORA-01476

V.2 Exceptions définies par l'utilisateur

PL/SQL permet à l'utilisateur de définir des d'exceptions dans la zone des déclarations ou des spécifications de sous-programmes. Cela se fait en donnant un nom à l'exception.

```
nomException EXCEPTION;
```

- **Déclenchement**

Une exception utilisateur ne sera pas levée de la même manière qu'une exception interne. Le programme doit explicitement dérouter le traitement vers le bloc des exceptions par la

directive RAISE. L'instruction RAISE permet également de déclencher des exceptions prédéfinies. Dans l'exemple suivant, programmons deux exceptions :

- erreur_piloteTropJeune qui va interdire l'insertion des pilotes ayant moins de 200 heures de vol.
- erreur_piloteTropExperimente qui va interdire l'insertion des pilotes ayant plus de 20000 heures de vol.

Exemple :

```
CREATE PROCEDURE saisiePilote
                (p_brevet IN VARCHAR2,p_nom IN VARCHAR2,
                 p_nbHVol IN NUMBER,p_comp IN VARCHAR2) IS
        erreur_piloteTropJeune          EXCEPTION ;
        erreur_piloteTropExperimente    EXCEPTION ;
BEGIN
  INSERT INTO Pilote (brevet, nom, nbHVol, comp)
        VALUES (p_brevet, p_nom, p_nbHVol, p_comp);
IF p_nbHVol < 200 THEN RAISE erreur_ piloteTropJeune ;
END IF ;
IF p_nbHVol > 20000 THEN RAISE erreur_ piloteTropExperimente ;
END IF ;
COMMIT;
EXCEPTION
  WHEN erreur_ piloteTropJeune THEN
        ROLLBACK ;
        DBMS_OUTPUT.PUT_LINE('désolé, le pilote manque d' 'expérience') ;
WHEN erreur_ piloteTropExperimente THEN
        ROLLBACK ;
        DBMS_OUTPUT.PUT_LINE('désolé, le pilote a trop d' 'expérience') ;
WHEN OTHERS THEN
        ROLLBACK ;
        DBMS_OUTPUT.PUT_LINE('Erreur d' 'oracle' || SQLERRM || '(' ||
        SQLCODE || ')') ;
END ;
```