

## Cours Base de données II

Professeur : Khalid HOUSNI

# Plan de cours

- **Ch. I. Principes de fonctionnement des (SGBD)**
  - Introduction aux bases de données
  - Modèle de base de données
  - Système de gestion de base de données (SGBD)
- **Ch. II. Développement d'une base de données (tables, contraintes, relations, vues)**
  - Création de tables, contraintes et relations
  - Création et gestion des vues
- **Ch. III. Procédures stockées et Déclencheurs(Triggers)**
  - Gestion des procédures et fonctions
  - Métadonnées des procédures et fonctions
  - Extensions au standard SQL (Délimiteur, contrôle flux, boucles, Curseurs)
  - Structure d'un déclencheur
  - Types d'événements déclencheurs
  - Propriétés d'un déclencheur
  - Validation des données
- **Ch. IV Pages web pilotées par une base de données**
  - Ouverture d'une connexion vers une base de données
  - Optimisation des requêtes (Partitionnement des tables et indexes, indexation des opérations en ligne et en parallèle, Maintenance et configuration des indexes).
  - Stockage de données capturées par des formulaires
  - Envoi de requêtes dynamiques à une base de données
  - Générer une page web affichant les résultats d'une requête

## **Chapitre I : Principes de fonctionnement des SGBD**

- Introduction aux bases de données
- Modèle de base de données
- Système de gestion de base de données (SGBD)

# Introduction aux bases de données

- Un système d'information est construit autour de volumes de données de plus en plus important. Ces données doivent être stockées sur des supports physiques. Les données sont stockées et organisées dans des **bases de données – BD (databases – DB)**.
- Un utilisateur doit pouvoir les retrouver. Il faut pouvoir les **interroger** par des requêtes.
- Les données évoluent, il faut donc pouvoir les manipuler : **ajouter, modifier, supprimer** des données.
- Le logiciel de base qui permet de manipuler ces données est appelé un **système de gestion de bases de données – SGBD (database management system – DBMS)**.

*Tout système d'information est construit autour de bases de données*

# Introduction aux bases de données

- **Base de données**- Un ensemble organisé d'informations avec un objectif commun.
- Peu importe le support utilisé pour rassembler et stocker les données (**papier, fichiers, etc.**), dès lors que des données sont rassemblées et stockées d'une manière organisée dans un but spécifique, on parle de base de données.
- **Enjeux**
  - les bases de données de demain
  - devront être capables de gérer **plusieurs dizaines de téra-octets de données**, géographiquement distribuées à l'échelle d'Internet, par plusieurs dizaines de milliers d'utilisateurs dans un contexte d'exploitation changeant.

# Introduction aux bases de données

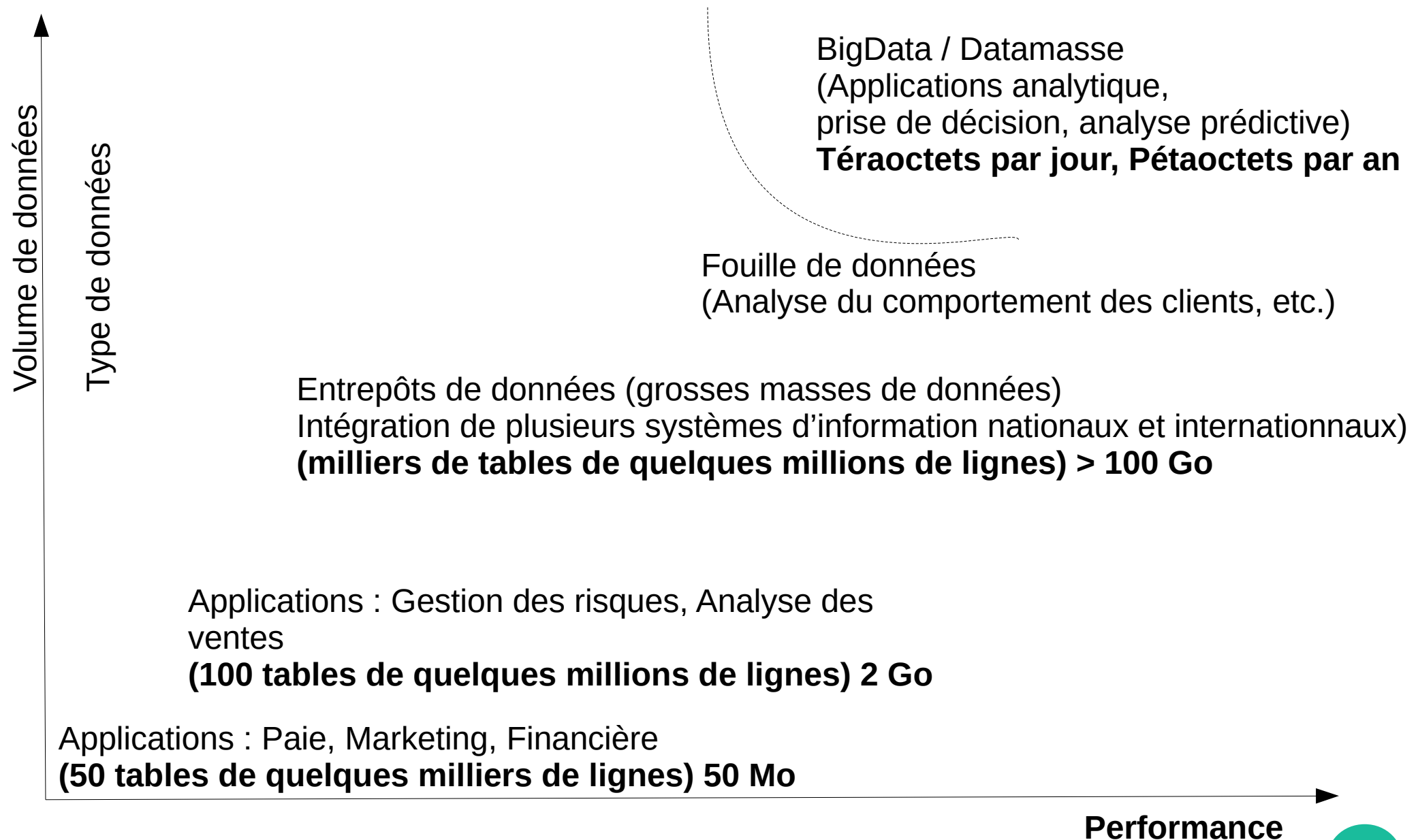
- **Une base de données informatique** est un ensemble de données qui ont été stockées sur un support informatique, et organisées et structurées de manière à pouvoir facilement consulter et modifier leur contenu.
- l'utilisation directe de fichiers soulève de très gros problèmes :
  - 1) **Lourdeur d'accès aux données.** En pratique, pour chaque accès, même le plus simples, il faudrait écrire un programme.
  - 2) **Manque de sécurité.** Si tout programmeur peut accéder directement aux fichiers, il est impossible de garantir la sécurité et l'intégrité des données.
  - 3) **Pas de contrôle de concurrence.** Dans un environnement où plusieurs utilisateurs accèdent aux mêmes fichiers, des problèmes de concurrence d'accès se posent.

# Concepts de base

- **Un Système de Gestion de Base de Données (SGBD)** est un logiciel (ou un ensemble de logiciels) permettant de manipuler les données d'une base de données.
- **SGBDR** : Le R de SGBDR signifie "relationnel". Un SGBDR est un **SGBD qui implémente la théorie relationnelle**.
- **Le langage SQL** :
  - Le SQL (Structured Query Language) est un langage informatique qui permet d'interagir avec des bases de données relationnelles.
  - Il a été créé dans les années **1970** et c'est devenu standard en 1986 (pour la norme ANSI - 1987 en ce qui concerne la norme ISO). Il est encore régulièrement amélioré.

# Historique

## Applications BD, ED, FD, ...



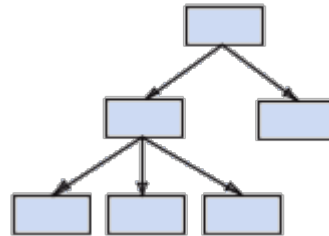


# Modèles de base de données

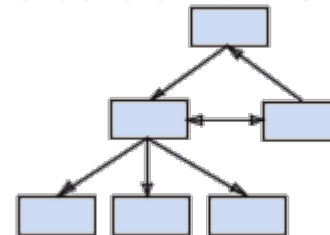
- Les bases de données sont apparues à la fin des années 60, à une époque où la nécessité d'un système de gestion de l'information souple se faisait ressentir.
- Il existe plusieurs modèles de SGBD, différenciés selon la représentation des données qu'elle contient

# Modèles de base de données

- **le modèle hiérarchique** : les données sont classées hiérarchiquement, selon une arborescence descendante. Ce modèle utilise des pointeurs entre les différents enregistrements. Il s'agit du premier modèle de SGBD

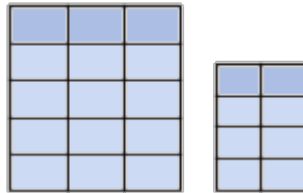


- Limitation : problème au niveau des relations  $N^*N$
- **le modèle réseau** : comme le modèle hiérarchique ce modèle utilise des pointeurs vers des enregistrements. Toutefois la structure n'est plus forcément arborescente dans le sens descendant

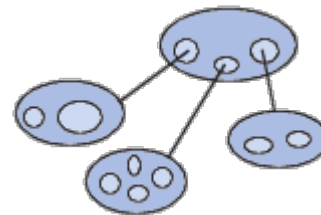


# Modèles de base de données

- **le modèle relationnel** : les données sont enregistrées dans des tableaux à deux dimensions (lignes et colonnes). La manipulation de ces données se fait selon la théorie mathématique des relations

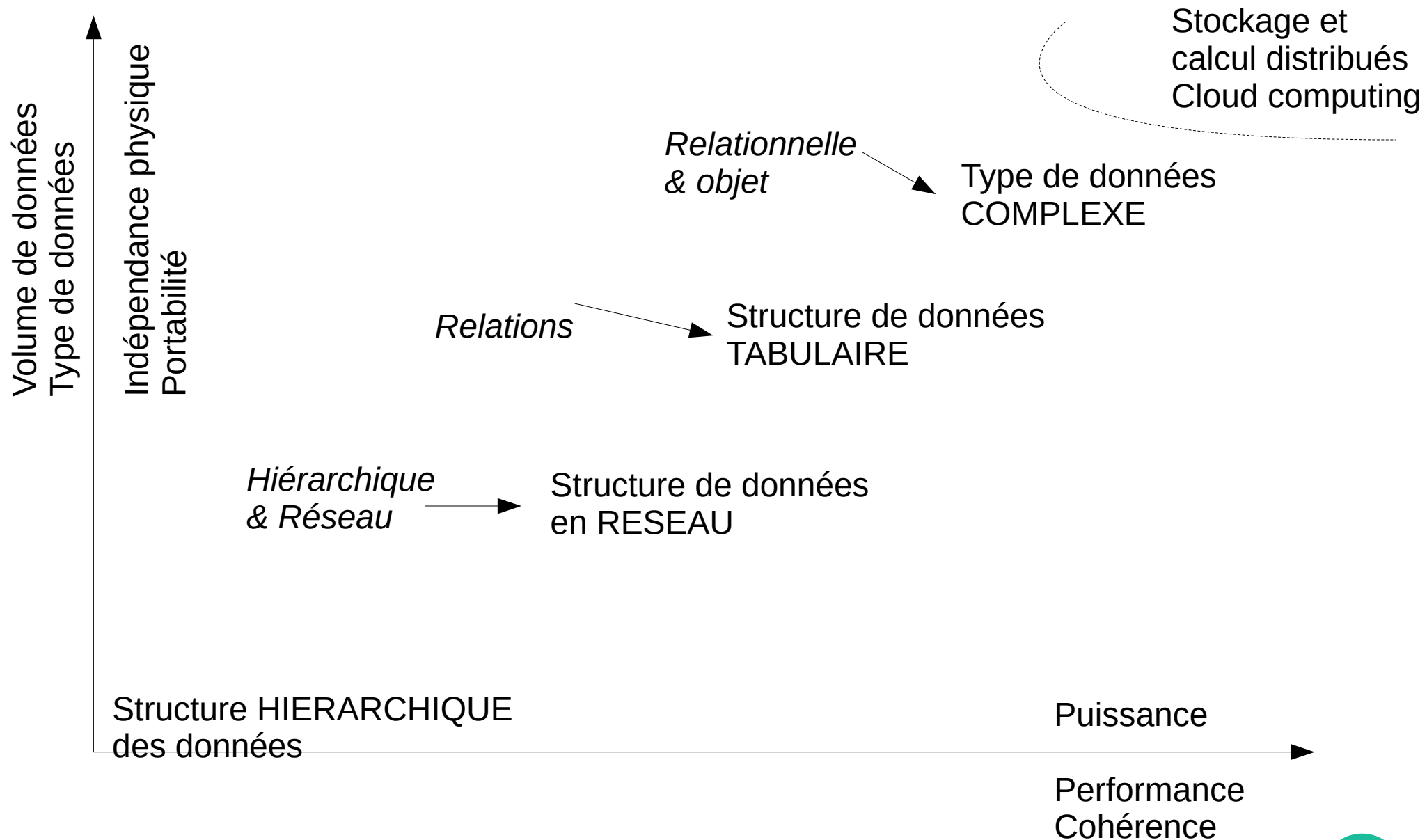


- **le modèle objet** : les données sont stockées sous forme d'objets, c'est-à-dire de structures appelées classes présentant des données membres. Les champs sont des instances de ces classes



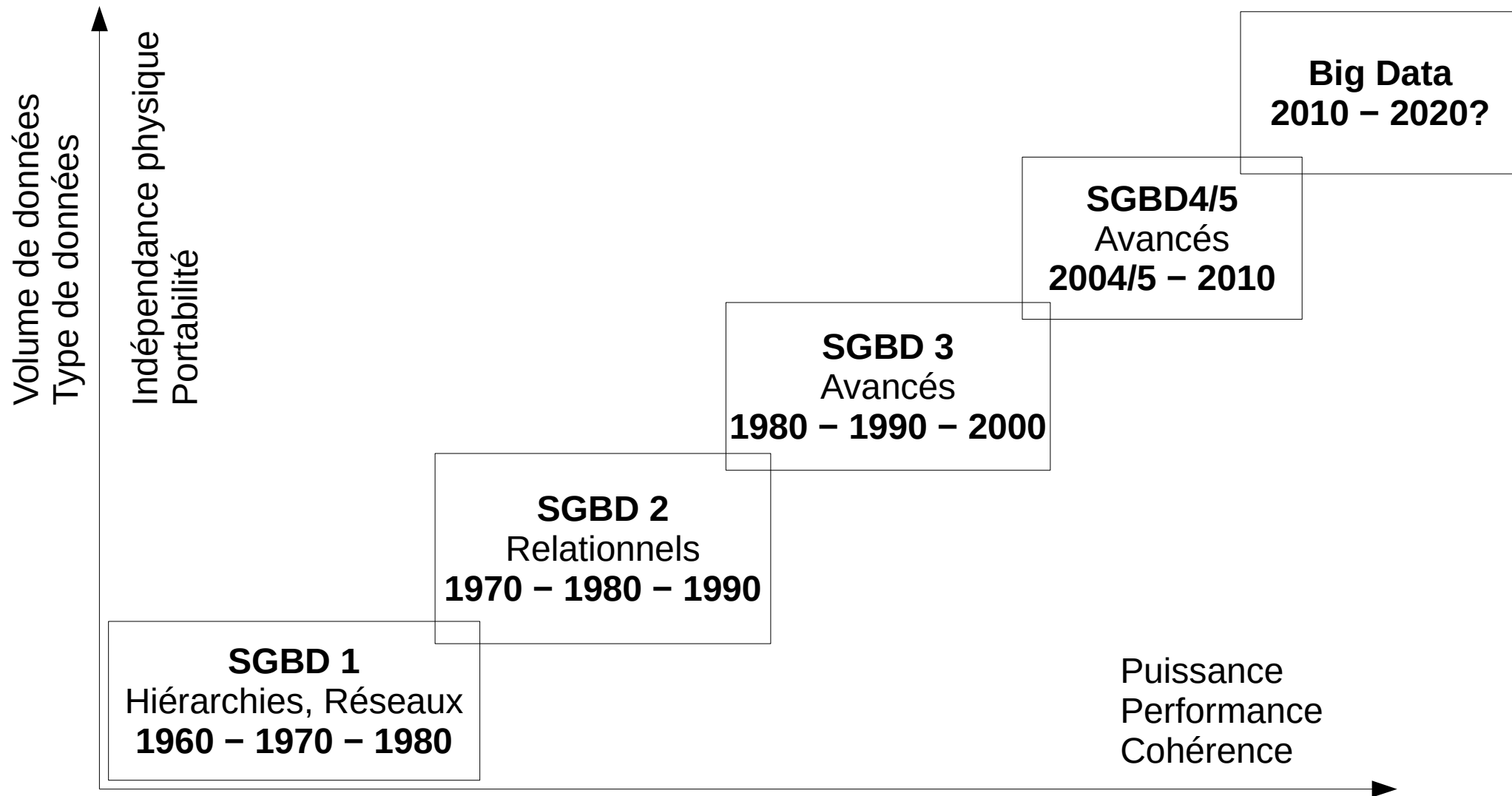
# Historique

## Structure et type de données



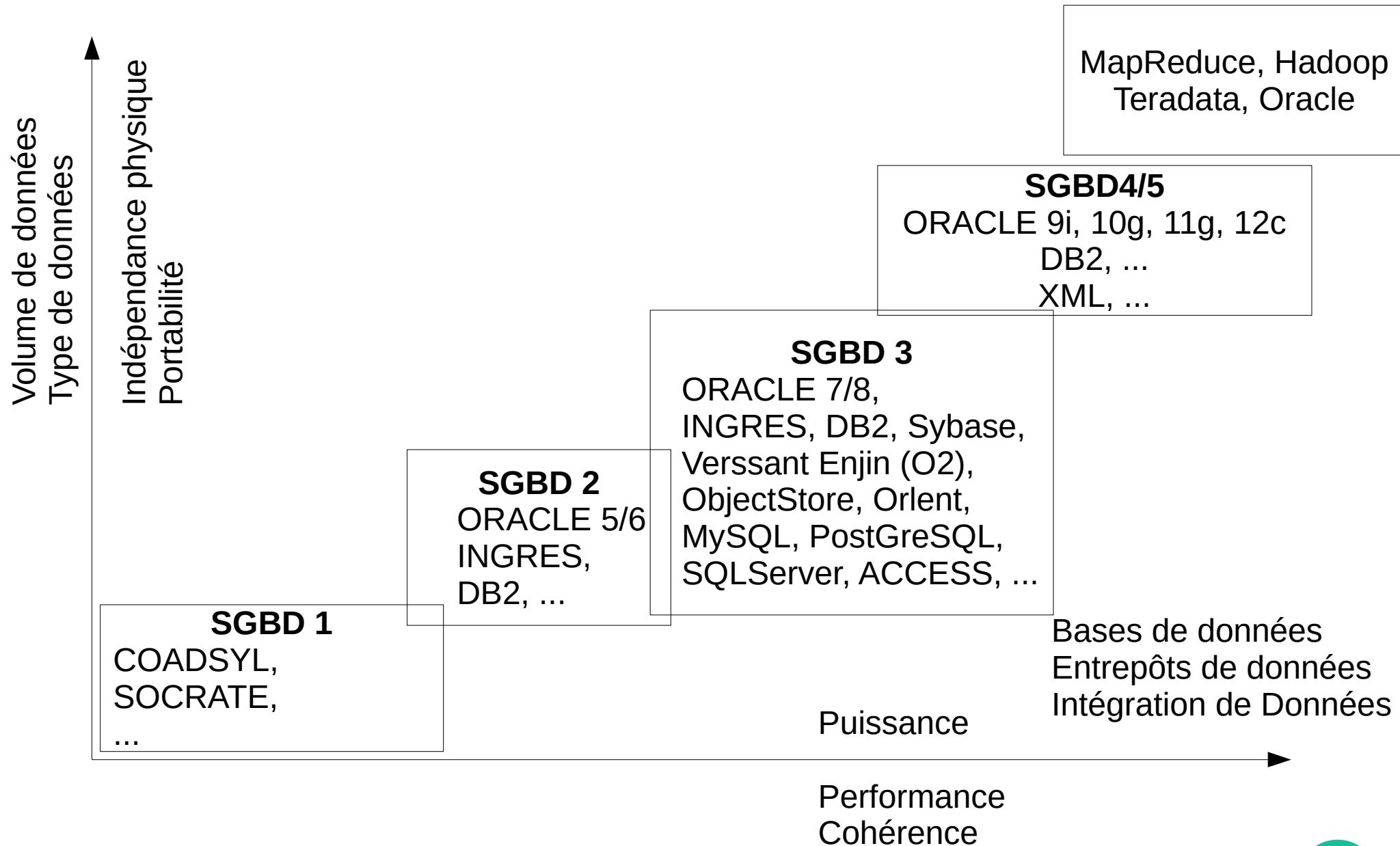
# Historique

## Génération de SGBD



# Historique

## Génération de SGBD



# Objectifs des SGBD

- Des objectifs principaux ont été fixés aux SGBD dès l'origine de ceux-ci et ce, afin de résoudre les problèmes causés par le système de fichier classique.
  - **Indépendance physique** : La façon dont les données sont définies doit être indépendante des structures de stockage utilisées.
  - **Indépendance logique** : Un même ensemble de données peut être vu différemment par des utilisateurs différents. Toutes ces visions personnelles des données doivent être intégrées dans une vision globale.
  - **Accès aux données** : L'accès aux données se fait par l'intermédiaire d'un Langage de Manipulation de Données (LMD).
  - **Administration centralisée des données** : Toutes les données doivent être centralisées dans un réservoir unique commun à toutes les applications.
  - **Non redondance des données**

# Objectifs des SGBD

- **Cohérence des données** : Les données sont soumises à un certain nombre de contraintes d'intégrité qui définissent un état cohérent de la base. Les contraintes d'intégrité sont décrites dans le Langage de Description de Données (LDD).
- **Partage des données**
- **Sécurité des données** : Les données doivent pouvoir être protégées contre les accès non autorisés.
- **Résistance aux pannes**



# SGBD «fichier» vs «client/serveur»

- Il existe à ce jour, deux type courant d'implémentation physique des SGBD relationnels :
  - Ceux qui utilisent un service de fichiers associés à un protocole de réseau afin d'accéder aux données : SGBD « fichier ». ces SGBD ne proposent en général pas le contrôle des transactions, et peu fréquemment le DDL et DCL.
    - **Avantage** : **Simplicité** du fonctionnement, **coût peu élevé** voir gratuit, **format des fichiers ouverts**, **administration quasi inexistante**.
    - **Inconvénient** : faible capacité de stockage, encombrement du réseau, faible nombre d'utilisateurs, droits d'accès sont gérés par le système d'exploitation ...
    - **Exemples** : Access, SQLite, MySQL

# SGBD «fichier» vs «client/serveur»

- Ceux qui utilisent une application centralisée dite serveur de données : SGBD client/serveur. Ce service consiste à faire tourner sur un serveur physique, un moteur qui assure une relation indépendance entre les données et les demandes de traitement venant des différentes applications.
  - **Avantage : grande capacité de stockage**, gestion de la **concurrency** dans un SI à grand nombre d'utilisateurs, indépendance vis à vis de l'OS, gestion des **transaction** ...
  - **Inconvénient : lourdeur dans le cas de solution « monoposte »**, **coût élevé** des licences, nécessite de machines puissantes
  - **Exemples** : InterBase, Oracle, PostgreSQL, SQL Server, Sybase , ..



- MySQL
  - MySQL est un Système de Gestion de Bases de Données Relationnelles.
  - MySQL utilise le langage SQL.
  - MySQL est un logiciel Open Source. Une version gratuite de MySQL est par conséquent disponible. À noter qu'une version commerciale payante existe également.
  - Le développement de MySQL commence en 1994 par David Axmark et Michael Widenius. EN 1995, la société MySQL AB. C'est la même année que sort la première version officielle de MySQL.
  - En 2008, MySQL AB est rachetée par la société **Sun Microsystems**, qui est elle-même rachetée par **Oracle Corporation en 2010**.

- Oracle DATABASE

- Edité par Oracle Corporation est un SGBDR payant.
- Son coût élevé fait qu'il est principalement utilisé par des entreprises.
- Oracle gère très bien de grands volumes de données. Pour plusieurs centaines de Go de données Oracle sera bien plus performant.
- Par ailleurs, Oracle dispose d'un langage procédural très puissant (du moins plus puissant que le langage procédural de MySQL) : le **PL/SQL**.

- PostgreSQL
  - Comme MySQL, **PostgreSQL est un logiciel Open Source.**
  - La première version Windows n'est apparue qu'à la sortie de la version 8.0 du logiciel, en 2005.
  - **PostgreSQL a longtemps été plus performant que MySQL,** mais ces différences tendent à diminuer. MySQL semble être aujourd'hui équivalent à PostgreSQL en terme de performances sauf pour quelques opérations telles que l'insertion de données et la création d'index.
  - Le langage procédural utilisé par PostgreSQL s'appelle le **PL/pgSQL.**

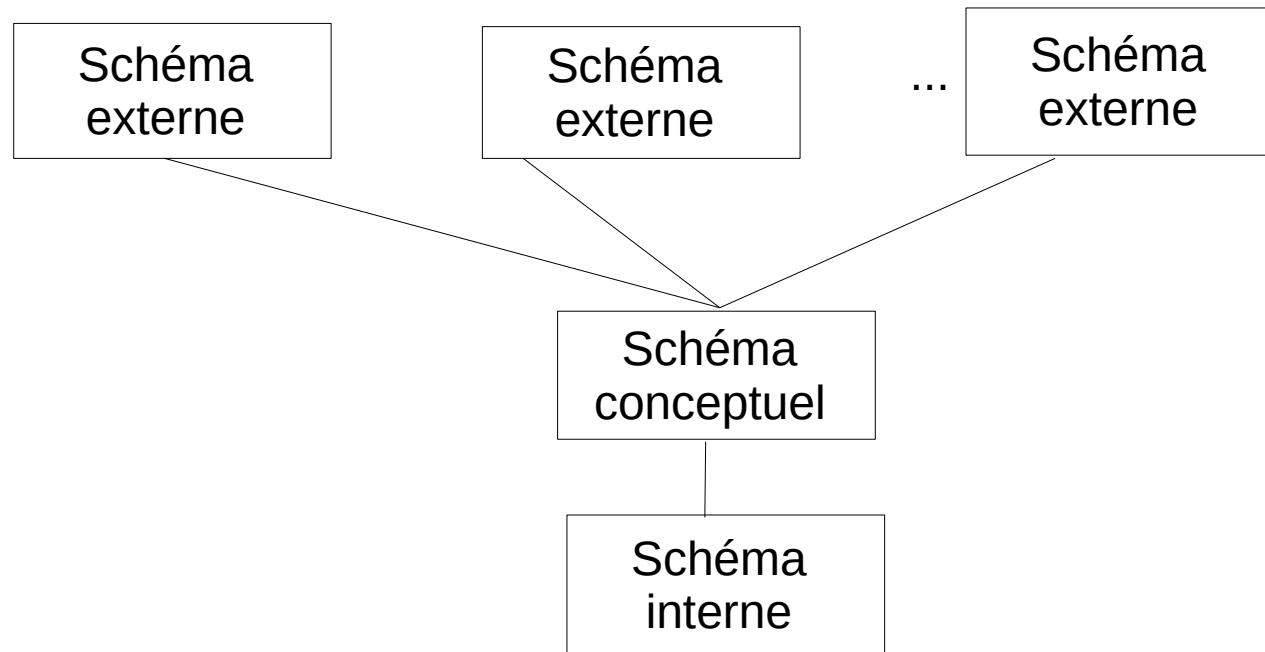


- MS Access
  - MS Access ou Microsoft Access est un logiciel édité par Microsoft
  - **c'est un logiciel payant qui ne fonctionne que sous Windows.**
  - **Il n'est pas du tout adapté pour gérer un grand volume** de données et a beaucoup moins de fonctionnalités que les autres SGBDR.
  - **Son avantage principal** est l'**interface graphique** intuitive qui vient avec le logiciel.

- SQLite

- La particularité de SQLite est de ne pas utiliser le schéma client-serveur utilisé par la majorité des SGBDR.
- SQLite stocke toutes les données dans de simples fichiers. Par conséquent, il ne faut pas installer de serveur de base de données, ce qui n'est pas toujours possible (certains hébergeurs web ne le permettent pas).
- Pour de très petits volumes de données, SQLite est très performant.
- Le fait que les informations soient simplement stockées dans des fichiers rend le système difficile à sécuriser (autant au niveau des accès, qu'au niveau de la gestion de plusieurs utilisateurs utilisant la base simultanément).

# Architecture à trois niveaux de description





Dans la suite de ce cours nous nous intéressons aux bases de données relationnelles et au SGBD Oracle version expresse

-

## **Ch. II. Développement d'une base de données (tables, contraintes, relations, vues)**

- Création de tables, contraintes et relations
- Création et gestion des vues

# Concepts de base

**Domaine :** ensemble de valeurs caractérisée par un nom

**Relation :** sous-ensemble du produit cartésien d'une liste de domaines caractérisée par un nom

La relation est une table à deux dimensions. Les colonnes sont les attributs et lignes sont les tuples.

**Attribut :** colonne au niveau d'une relation caractérisé par un nom. Un attribut possède un nom, un type et un domaine.

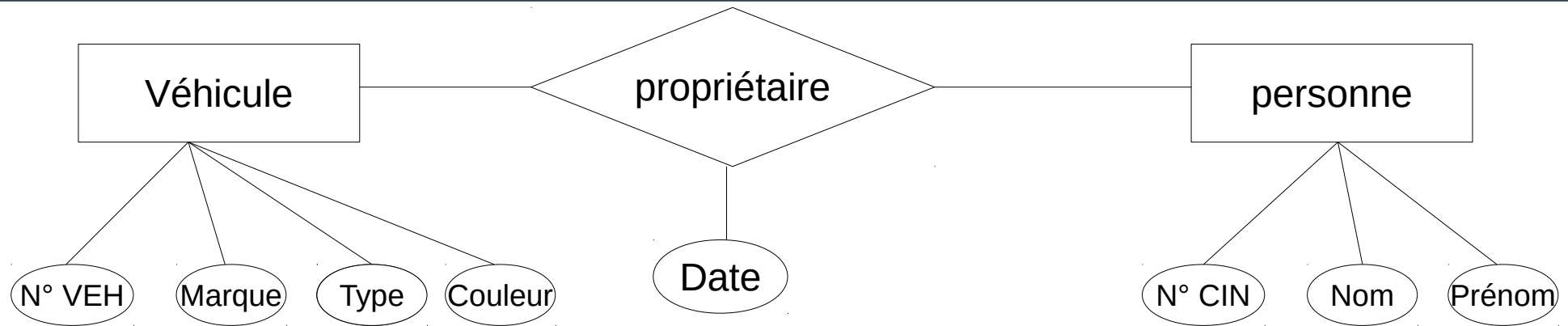
**Tuple :** Ligne d'une relation appelée aussi enregistrement.

**Schéma de la relation :** nom de la relation suivi de la liste des attributs et de la définition de leurs domaines

**Clé :** Ensemble minimal d'attributs dont la connaissance des valeurs permet d'identifier un tuple unique de la relation considérée.

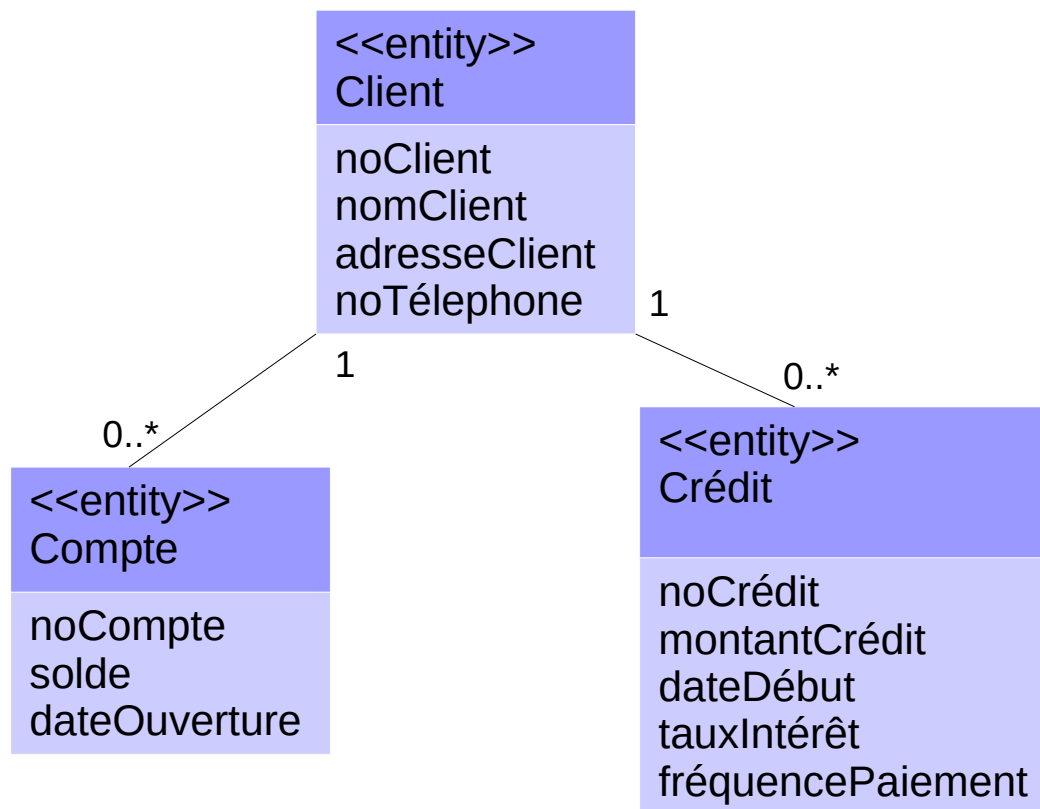
**Valeur NULL :** valeur conventionnelle dans une relation pour représenter une information inconnue

# Entité-Relation



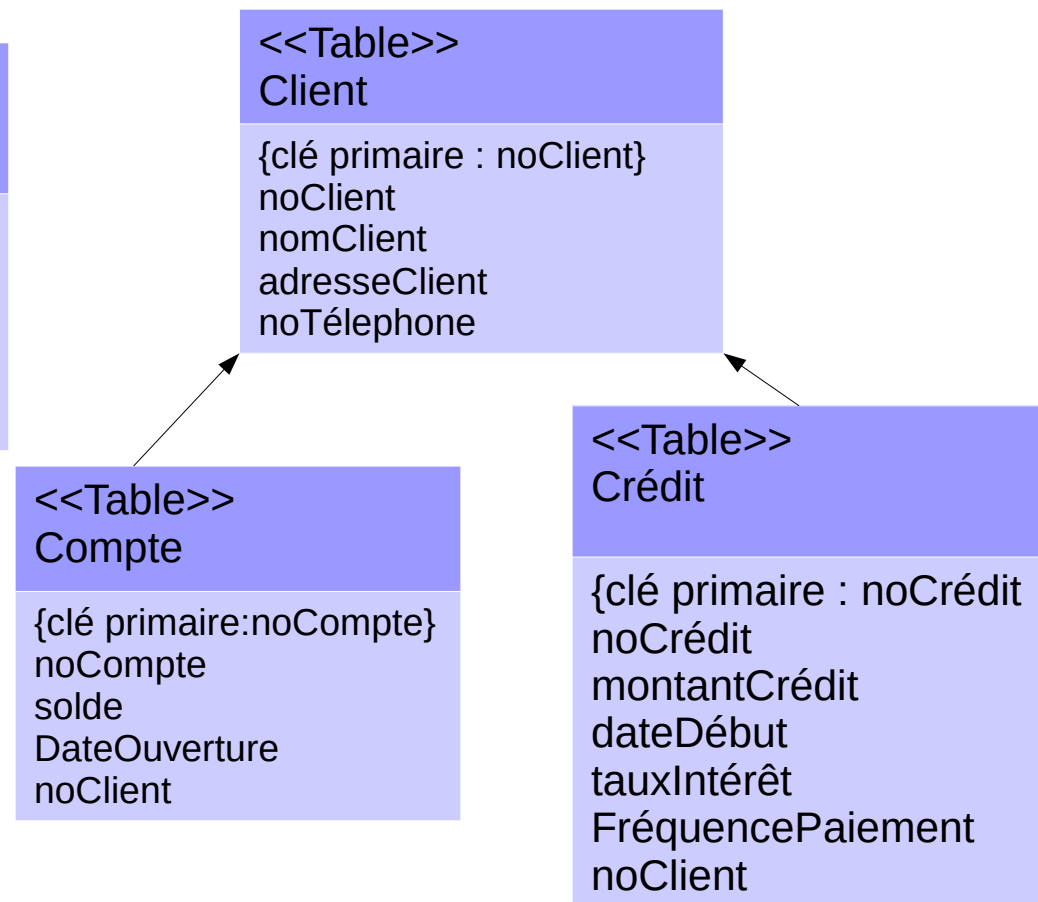
- **Entité** : définit comme un objet pouvant être identifié distinctement.  
**Ex** : un véhicule, une personne, un client, un livre, un compte bancaire, une commande, etc.
  - **Entités faibles** : son existence dépend de l'existence d'une autre entité.
  - **Ex** : une entité LIGNE DE COMMANDE n'existe que si l'entité COMMANDE correspondante est présente.
  - **Entités régulières** : son existence ne dépend pas de l'existence d'une autre entité.
- **Les relations** : représentent les liens existants entre les entités.  
**Ex** : un véhicule est relié à une personne.
  - **La dimension ou degré** de la relation est le nombre d'entités impliquées dans une relation.

# Conception du schéma d'une base de données

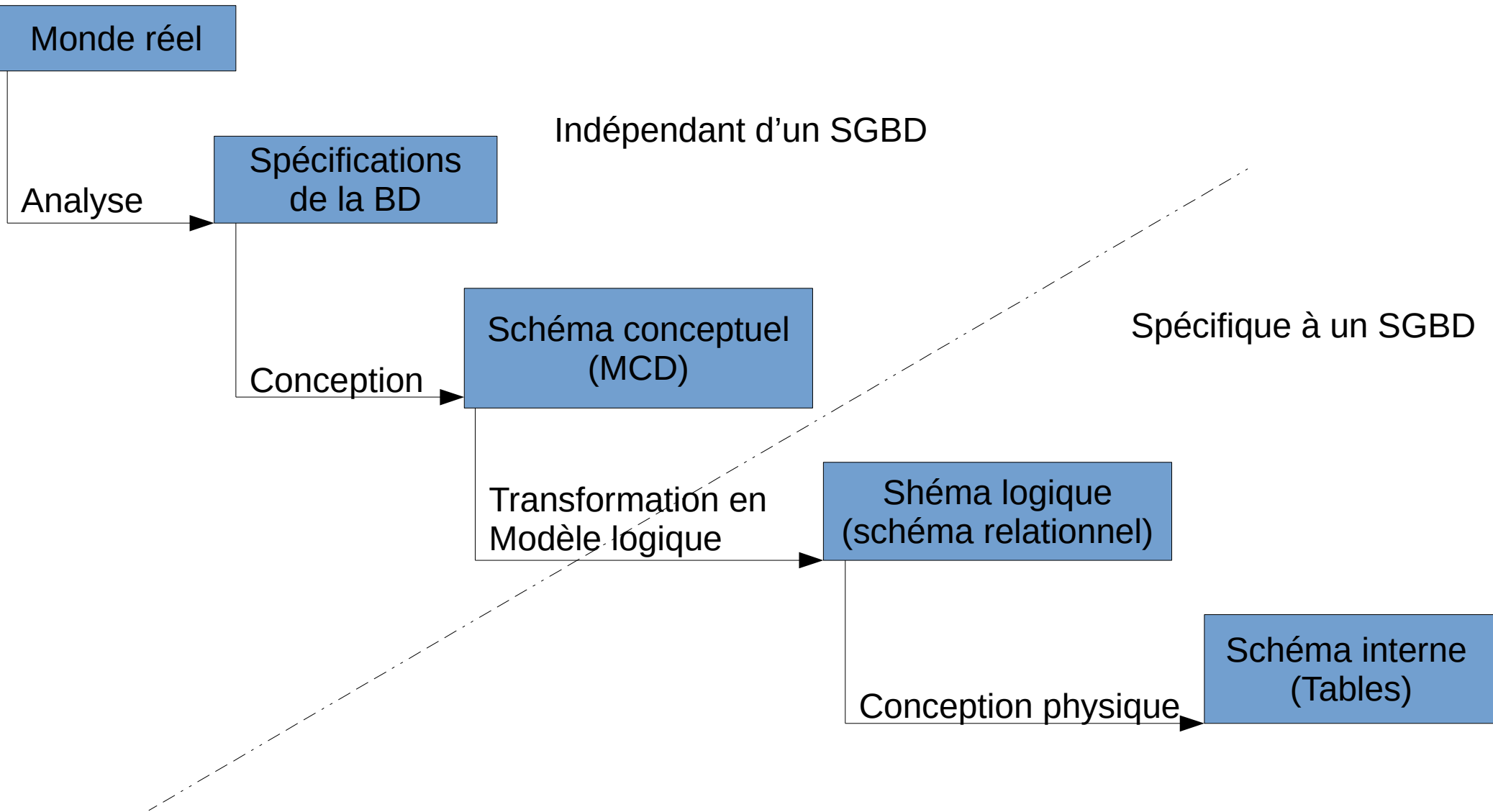


Modèle conceptuel

Schéma relationnel



# Processus de conception d'une BD



# La théorie de la normalisation

La théorie de la normalisation est l'un des points forts du modèle relationnel. Elle permet de définir formellement la qualité des tables au regard du problème posé par la redondance des données.

formes normales caractérisant les tables relationnelles :

**Première forme normale (1FN)** : si tous les attributs sont en dépendance fonctionnelle de la clé. Soit encore : si elle ne contient que des attributs atomiques.

**Deuxième forme normale (2FN)** : si elle est en 1FN et, si de plus, il n'existe pas de dépendance fonctionnelle entre une partie d'une clé et une colonne non clé de la table.

**Troisième forme normale (3FN)** : si elle est en 2FN et si, de plus, aucune dépendance fonctionnelle entre les colonnes non clé.

# La théorie de la normalisation

## Exemples

- **LIGNEARTICLE(numcommande, numproduit, quantité, prixproduit)**  
dont la clé est **numcommande + numproduit**
  - n'est pas en 2FN car **prixproduit** dépend fonctionnellement de **numproduit**
- **PRODUIT(numproduit, ..., numfournisseur, nomfournisseur, adressefournisseur, ...)** de clé **numproduit**
  - n'est pas en 3FN car **nomfournisseur** et **adressefournisseur** dépendent fonctionnellement de **numfournisseur**.
- Une solution est d'éclater la relation en deux relations : la relation **PRODUIT(numproduit, ..., numfournisseur)** et la relation **FOURNISSEUR(numfournisseur, nomfournisseur, adresse fournisseur)**.



# Langage SQL

- *Structured Query Language*
- Norme établie pour SGBD relationnel
- Les subdivisions du SQL
  - **LDD** (langage de définition des données): c'est la partie du SQL qui permet de créer des bases de données, des tables, des vues, des domaines, des index, des contraintes, ...  
**ALTER, CREATE, DROP**
  - **LMD** (langage de manipulation de données) : c'est la partie du SQL qui s'occupe de traiter les données : mise à jour et interrogation (requêtes) **INSERT, UPDATE, DELETE, SELECT**
  - **LCD** (langage de contrôle de données):c'est la partie du SQL qui s'occupe de gérer les droit d'accès au tables **GRANT, REVOKE**
  - **LCT** (langage de contrôle de transactions) : c'est la partie du SQL chargé de contrôler la bonne exécution des transactions **SET TRANSACTION, COMMIT, ROLLBACK**
  - **SQL intégré (Embedded SQL)** : il s'agit d'éléments procéduraux que l'on intègre à un langage hôte (C++, Java ...) **SET, DECLARE CURSOR, OPEN, FETCH**

# Langage SQL

- Il n'y a aucune différence entre un SQL disponible pour un SGBDR de type fichier et un autre SGBD C/S.
- Dans ce document, les exemples d'illustration utilisent la base de données dont le schéma relationnel est
  - avions(No\_AV, NOM\_AV, CAP, LOC)
  - pilotes(No\_PIL, NOM\_PIL, VILLE)
  - vols(No\_VOL, No\_AV, No\_PIL, V\_d, V\_a, H\_d, H\_a)

# SQL comme LDD : Exemple

- **avions(no\_AV, NOM\_AV, CAP, LOC)**
  - no\_AV NUMBER(4)
  - NOM\_AV VARCHAR2(20)
  - CAP\_NUMBER(4)
  - LOC VARCHAR2(15)
- **pilotes(no\_PIL, NOM\_PIL, VILLE)**
  - no\_PIL NUMBER(4)
  - NOM\_PIL VARCHAR2(20)
  - VILLE VARCHAR2(15)
- **vols(no\_VOL, no\_AV, no\_PIL, V\_d, V\_a, H\_d, H\_a)**
  - no\_VOL VARCHAR2(5)
  - no\_AV NUMBER(4)
  - no\_PIL NUMBER(4)
  - V\_d VARCHAR2(15)                      - H\_d DATE
  - V\_a VARCHAR2(15)                      - H\_a DATE

# Types de données en SQL2

- **Types pour les chaînes de caractères**
  - CHAR(taille)
    - chaînes de caractères de longueur fixe
    - codage en longueur fixe : remplissage de blancs
    - taille comprise entre 1 et 2000 octets
  - VARCHAR(taille max)
    - chaînes de caractères de longueur variable
    - taille comprise entre 1 et 4000 octets
  - constantes
    - chaînes de caractères entre guillemets
  - NCHAR(taille), NVARCHAR(taille max) sont codés sur le jeu UNICODE

# Types de données en SQL2

- **Types numériques**

- types numériques pour les entiers :
  - **SMALLINT** pour 2 octets
  - **INTEGER** pour 4 octets
- types numériques pour les décimaux à virgule flottante :
  - **REAL**
  - **DOUBLE PRECISION**
  - **FLOAT**
- types numériques pour les décimaux à virgule fixe :
  - **DECIMAL(nb chiffres, nb décimales)**
  - **NUMERIC(nb chiffres, nb décimales)**
- constantes
  - exemples : 43.8, -13, 5.3E-6

# Types de données en SQL2

- **Types temporels**
  - DATE
    - pour les dates
  - TIME
    - pour les heures, minutes et secondes
  - TIMESTAMP
    - pour un moment précis : date et heure, minutes et secondes (précision jusqu' à la microseconde)
  - constantes
    - exemples : '1/05/2007' ou '1 MAY 2007'
- **Remarque :** pour oracle le type DATE remplace DATE et TIME de SQL2

# Manipulation des dates

- Exemple de manipulation des dates

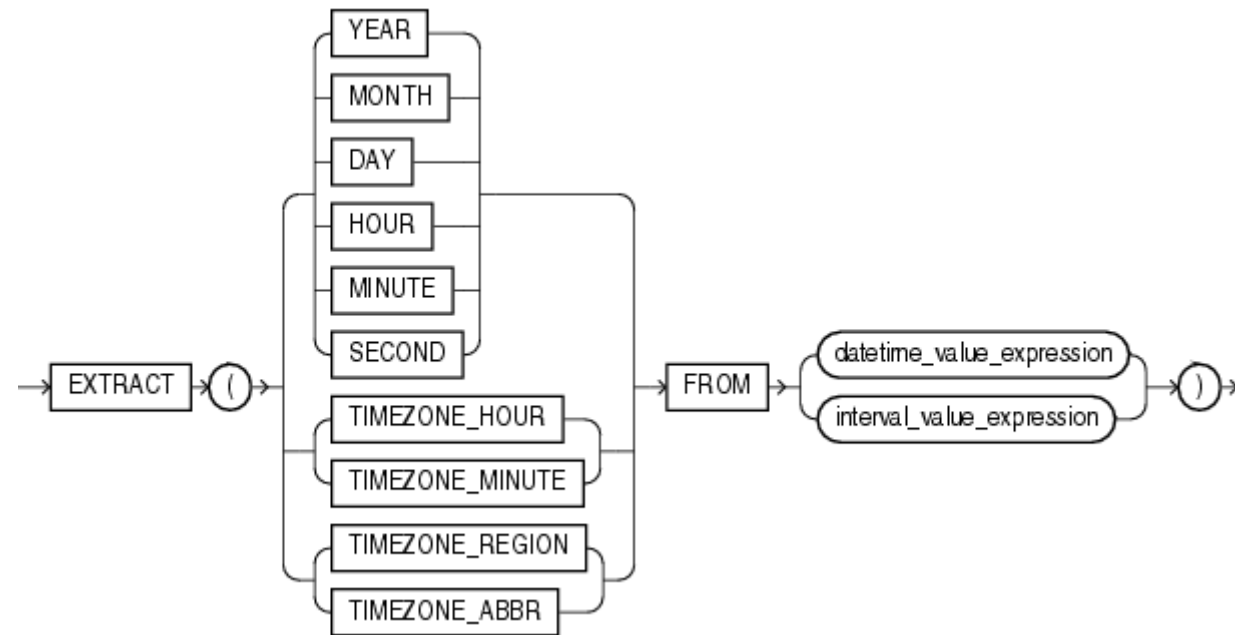
SELECT

```
TO_CHAR(SYSDATE, 'DD') JOUR,  
TO_CHAR(SYSDATE, 'MM') MOIS,  
TO_CHAR(SYSDATE, 'YYYY') ANNEE,  
TO_CHAR(SYSDATE, 'HH24') HEURE,  
TO_CHAR(SYSDATE, 'MI') MINUTE,  
TO_CHAR(SYSDATE, 'SS') SECONDE  
FROM DUAL;
```

- EXTRACT

Retourne l'année, le mois, le jour, l'heure, la minute ou la seconde d'une date :

```
SELECT EXTRACT(YEAR FROM SYSDATE),  
EXTRACT(MONTH FROM SYSDATE),  
EXTRACT(DAY FROM SYSDATE)  
FROM DUAL;
```



```
SELECT SUM(extract(hour from h_d)) NBR_H  
FROM vols where no_pil=1;
```

# Types de données en SQL2

- **Type booléen**
  - **BIT**
    - pour enregistrer la valeur d'un bit
    - **exemples** : BIT(1), BIT(4)
  - **pas supporté par ORACLE**



# Nouveaux type de données SQL3

- **BOOLEAN**
- **CHARACTER LARGE OBJECT[(n [m])]** (ou **CLOB**) est un type littéral permettant de stocker des caractères de grande dimension dans un jeu à deux octets par caractères (ASCII). n la longueur et m l'unité (K pour kilo, M méga ou G pour giga octects.
- **NATIONAL CHARACTER LARGE OBJECT[(n [m])]** (ou **NCLOB**) pour un jeu UNICODE.
- **BINARY LARGE OBJECT[(n [m])]** (ou **BLOB**) une chaine de bits de grande dimenssion dont la longueur est fixe.

- création de table

**CREATE TABLE** nom de table (liste de définition de colonne, [liste de contrainte de table]) ;

définition de colonne : :=

nom de colonne

(nom de domaine ou type)

[liste de contrainte de colonne]

[**DEFAULT** valeur par défaut]

- création de table : contrainte de colonne

contrainte de colonne ::=

[**CONSTRAINT** nom]

type de contrainte de colonne

type de contrainte de colonne ::=

**PRIMARY KEY** ou

**NOT NULL** ou

**UNIQUE** ou

**CHECK**(condition sur valeur) ou

**REFERENCES** nom de table(nom de colonne)

- création de table : contrainte de table

contrainte de table ::=

[**CONSTRAINT** nom]

type de contrainte de table

type de contrainte de table ::=

**PRIMARY KEY** (liste de nom de colonne) ou

**NOT NULL** (liste de nom de colonne) ou

**UNIQUE** (liste de nom de colonne) ou

**CHECK** (condition sur ligne) ou

**FOREIGN KEY** liste de nom de colonne **REFERENCES**

nom de de table (liste de nom de colonne)

- Exemple : Création de la table avions à partir du schéma :  
avions(no\_AV, nom\_AV, CAP, LOC)

```
CREATE TABLE avions (  
    no_AV NUMBER(4)  
    CONSTRAINT Cle_P_avions PRIMARY KEY,  
    NOM_AV VARCHAR2(20),  
    CAP NUMBER(4)  
    CONSTRAINT Dom_CAP_avions CHECK (CAP >= 4),  
    LOC VARCHAR2(15) );
```

# SQL comme LDD : Exemple : Création de la table vols

- Création de la table vols à partir du schéma :  
vols(no\_VOL, no\_AV, no\_PIL, V\_d, V\_a, H\_d, H\_a)
- Contraintes de colonnes ?
- Contraintes de table ?

# SQL comme LDD : Exemple : Création de la table vols

- vols(no VOL, no AV, no PIL, V d, V a, H d, H a)

**CREATE TABLE** vols (

no\_VOL **VARCHAR2**(5)

**CONSTRAINT** Cle\_P\_vols **PRIMARY KEY**,

no\_AV **NUMBER**(4)

**CONSTRAINT** Ref\_no\_AV\_vols **REFERENCES** avions,

no\_PIL **NUMBER**(4)

**CONSTRAINT** Ref\_no\_PIL\_vols **REFERENCES** pilotes,

V\_d **VARCHAR2**(15) **NOT NULL**,

V\_a **VARCHAR2**(15) **NOT NULL**,

H\_d **DATE**,

H\_a **DATE**,

**CONSTRAINT** C1\_vols **CHECK** (v\_d <> v\_a),

**CONSTRAINT** C2\_vols **CHECK** (h\_d < h\_a) );

- suppression de table

**DROP TABLE** nom ;

Quand une table est supprimée, ORACLE :

efface tous les index qui y sont attachés quelque soit le propriétaire

efface tous les privilèges qui y sont attachés

**MAIS les vues et les synonymes se référant à cette table ne sont pas supprimés**



- modification de table

**ALTER TABLE** nom\_de\_table modification\_de\_table ;

modification\_de\_table ::=

**ADD COLUMN** définition de colonne

**ADD CONSTRAINT** contrainte de table

**DROP COLUMN** nom de colonne

**DROP CONSTRAINT** nom de contrainte

# SQL comme LDD

- Exemple : Ajout d'une colonne à la table vols de schéma :

vols(no\_VOL, no\_AV, no\_PIL, V\_d, V\_a, H\_d, H\_a)

**ALTER TABLE** vols **ADD COLUMN** COUT\_VOL **NUMBER**(8)

le schéma devient :

vols(no\_VOL, no\_AV, no\_PIL, V\_d, V\_a, H\_d, H\_a, COUT\_VOL)

## • MODIFIER DES COLONNES MULTIPLES

- Pour MODIFIER DES COLONNES MULTIPLES dans une table existante, la syntaxe Oracle ALTER TABLE est la suivante:

**ALTER TABLE** table\_name

**MODIFY** (column\_1 column\_type,

column\_2 column\_type,

...

column\_n column\_type);

- Exemple de modification du type d'un champ sous oracle

alter table vols

Modify (

H\_D TIMESTAMP ,

H\_A TIMESTAMP );

- insertion de lignes dans une table

**INSERT INTO** nom de table [liste de colonnes] **VALUES** liste de valeurs ;

ou

**INSERT INTO** nom de table [liste de colonnes] requête ;

# SQL comme LDD

- ajouter un avion dans la table avions en respectant l'ordre des colonnes

```
INSERT INTO avions VALUES (100, 'Airbus', 200, 'Agadir') ;
```

- ajouter un avion dans la table avions sans connaître l'ordre

```
INSERT INTO avions (no_AV, CAP, LOC, NOM_AV)
```

```
VALUES (101, 200, 'Agadir', 'Airbus') ;
```

- ajouter un avion dans la table avions dont la localisation est INDEFINI

```
INSERT INTO avions (no_AV, NOM_AV, CAP)
```

```
VALUES (102, 'Airbus', 200) ;
```

ou

```
INSERT INTO avions
```

```
VALUES (102, 'Airbus', 200, NULL) ;
```

```
insert into vols values(1,1,1, 'Agadir', 'Rabat',  
To_DATE('1983-04-25 8:30:00AM', 'YYYY-MM-DD HH:MI:SSAM'),  
To_DATE('1983-04-25 10:30:00AM', 'YYYY-MM-DD HH:MI:SSAM'));
```

- suppression de lignes d'une table

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

Exemples :

- vider la table avions

**DELETE FROM** avions ;

- supprimer de la table avions tous les avions dont la capacité est inférieur à 100

**DELETE FROM** avions

**WHERE** CAP < 100 ;

- modification de lignes dans une table

**UPDATE** nom\_de\_table **SET** liste\_expression\_colonne  
[**WHERE** condition] ;

expression\_colonne ::=

nom\_de\_colonne = expression ou

nom\_de\_colonne = requête

- Exemple :
- modifier la capacité de l'avion numéro 100

**UPDATE** avions **SET** CAP = 300

**WHERE** no AV = 100 ;



# SQL comme Langage de Requêtes

- interrogation

requête ::= **SELECT** [**DISTINCT**] projection

**FROM** liste de (nom de table [**AS** nom]) | (requête **AS** nom)

**WHERE** condition

[**GROUP BY** liste de nom de colonne]

[**HAVING** condition]

[**ORDER BY** liste de ((nom de colonne | rang de colonne) (**ASC** | **DESC**))];

requête ::= requête ( **UNION** | **INTERSECT** | **EXCEPT** ) requête

requête ::= (requête)

# SQL comme Langage de Requêtes

- projection ::=

\* | nom de table | liste de (terme de projection[[AS] nom])

terme de projection ::=

expression | agrégation

expression ::=

valeur | nom de colonne | expression arithmétique | ...

agrégation ::=

COUNT(\*)

opérateur d'agrégation([DISTINCT] expression)

opérateur d'agrégation ::=

COUNT | SUM | AVG | MAX | MIN

# SQL comme Langage de Requêtes

- comparaison ::=

expression (= | <> | > | < | <= | >= | ) expression |

expression (= | <>) (| SOME | ALL ) requête

expression (> | < | <= | >=) (| SOME | ALL )

requête\_mono\_colonne

- appartenance à un intervalle ::=

expression BETWEEN expression AND expression

- appartenance à un ensemble ::=

expression (IN | NOTIN) (requête) |

(liste de expression) (IN | NOTIN) (requête)

- ensemble de valeurs ::= (liste de valeur) | requête mono colonne

- existence ::= EXISTS (requête)

# SQL comme Langage de Requêtes

- Sélection de lignes

- **toutes les lignes et toutes les colonnes**

`SELECT * FROM` nom de table ;

- pour connaître toutes les caractéristiques des avions stockés dans la table

`SELECT * FROM` avions ;

- **toutes les lignes mais seulement certaines colonnes**

`SELECT` liste de nom de colonne `FROM` nom de table;

- pour connaître les numéros de tous les vols

`SELECT` no\_VOL `FROM` vols ;

# SQL comme Langage de Requêtes

- Sélection de lignes

- **suppression des lignes dupliquées**

`SELECT DISTINCT` liste de nom de colonne `FROM`  
nom de table ;

- pour connaître les numéros des pilotes qui conduisent au moins un avion

`SELECT DISTINCT` no\_PIL `FROM` vols ;

- **colonnes calculées**

`SELECT` expression [`AS` alias] `FROM` nom de table ;

- afficher une augmentation de 5% du coût de chaque vol

`SELECT` no\_VOL, '5%' "%",  
COUT\_VOL\*0.05 AUGM, COUT\_VOL\*1.05 "Nouveau coût"  
`FROM` vols ;

# SQL comme Langage de Requêtes

- sur les chaînes de caractères

- afficher les trajets assurés par les vols sous la forme :

Ville de départ -- > Ville d'arrivée

```
SELECT no_VOL, V_d || ' -- > ' || V_a TRAJET  
FROM vols ;
```

- sur les dates

- afficher les dates de départ et d'arrivée de chaque vol en décalant les dates

```
SELECT no_VOL, D_d + 1/24 D_d, D_a + 1/24 D_a  
FROM vols ;
```

- pour connaître la durée en heures de tous les vols

```
SELECT no_VOL, 24 *(D_a -D_d) durée  
FROM vols ;
```

# SQL comme Langage de Requêtes

- Recherche par comparaison

**SELECT** liste de nom de colonne **FROM** nom de table

**WHERE** expression ;

- pour connaître tous les avions qui ont une capacité > 200 places

**SELECT** no\_AV

**FROM** avions

**WHERE** CAP > 200 ;

- pour connaître tous les pilotes qui effectuent des vols qui durent plus d'une heure

**SELECT DISTINCT** no\_PIL

**FROM** vols

**WHERE** (24 \*(D\_a - D\_d) ) > 1 ;

# SQL comme Langage de Requêtes

- Recherche par ressemblance

`SELECT` liste de nom\_de\_colonne `FROM` nom\_de\_table

`WHERE` expression [`NOT`] `LIKE` motif [caractères spéciaux] ;

- caractère spéciaux :

- `%` : remplace 0, 1 ou plusieurs caractères
- `_` remplace 1 caractère

- caractère d'échappement :

- permet de traiter les caractères spéciaux comme de simples caractères
- il n'y a pas de caractère d'échappement prédéfini



- Recherche par ressemblance : exemples  
pour connaître la capacité de tous les avions Boeing

```
SELECT no_AV, NOM_AV, CAP  
FROM avions  
WHERE NOM_AV LIKE 'Boeing%';
```

# SQL comme Langage de Requêtes

- Recherche avec condition conjonctive

**SELECT** liste de nom\_de\_colonne **FROM** nom\_de\_table

**WHERE** condition **AND** condition ;

- pour connaître tous les avions qui sont à Agadir et dont la capacité est de 300 places

**SELECT** no\_AV

**FROM** avions

**WHERE** LOC = 'Agadir' **AND** CAP = 300 ;

# SQL comme Langage de Requêtes

- Recherche avec condition disjonctive

**SELECT** liste de nom\_de\_colonne **FROM** nom\_de\_table

**WHERE** condition **OR** condition ;

- pour connaître tous les vols qui utilisent les avions 100 ou 101

**SELECT** no VOL

**FROM** vols

**WHERE** no\_AV = 100 **OR** no\_AV = 101 ;

# SQL comme Langage de Requêtes

- Recherche avec condition négative

**SELECT** liste de nom\_de\_colonne **FROM** nom\_de\_table

**WHERE NOT** condition ;

- pour connaître tous les pilotes qui n'habitent pas à Rabat

**SELECT** no\_PIL

**FROM** pilotes

**WHERE NOT** VILLE = 'Rabat' ;

# SQL comme Langage de Requêtes

- Recherche avec un intervalle

**SELECT** liste de nom\_de\_colonne **FROM** nom\_de\_table

**WHERE** expression **BETWEEN** expression **AND** expression ;

- pour connaître tous les avions qui ont une capacité entre 200 et 300 places

**SELECT** no AV

**FROM** avions

**WHERE** CAP **BETWEEN** 200 **AND** 300 ;

# SQL comme Langage de Requêtes

- Recherche avec une liste

**SELECT** liste de nom\_de\_colonne **FROM** nom\_de\_table

**WHERE** expression [**NOT**] **IN** liste de expression ;

- pour connaître tous les pilotes qui habitent soit à Rabat soit à Casa

**SELECT** no\_PIL

**FROM** pilotes

**WHERE** VILLE **IN** ('Rabat', 'Casa') ;

# SQL comme Langage de Requêtes

- Recherche avec une liste

**SELECT** liste de nom\_de\_colonne **FROM** nom de table

**WHERE** expression (<> | > | < | <= | >= | ) **ALL**

liste de expression ;

**SELECT** liste de nom de colonne **FROM** nom de table

**WHERE** expression (<> | > | < | <= | >= | ) **SOME**

liste de expression ;

- pour connaître tous les vols dont le départ est à plus de 5 jours et dont la durée est moins de 5 heures

**SELECT** no\_VOL, D\_d, 24\*(D\_a - D\_d) Durée

**FROM** vols

**WHERE** D\_d > ALL (sysdate +5, D\_a -5/24) ;

# SQL comme Langage de Requêtes

- Traitement de l'absence de valeur
  - **sur les expressions numériques**

un calcul numérique ou de dates exprimé avec les opérateurs +, -, \*, / n'a pas de valeur **lorsqu'au moins une des composantes n'a pas de valeur**

```
SELECT no_AV, 2*CAP/3 AS CAP_RED  
FROM avions  
WHERE no_AV = 320 ;
```



# SQL comme Langage de Requêtes

- Traitement de l'absence de valeur
  - **sur les chaînes de caractères**
    - un calcul de chaînes exprimé avec l'opérateur || n'a pas de valeur **lorsque toutes ses composantes n'ont pas de valeur**
    - la chaîne vide et l'absence de valeur sont confondues

```
SELECT no_CL, NOM_RUE_CL || " " || VILLE_CL
```

```
AS ADR_CL
```

```
ou no_CL, NOM_RUE_CL || NULL || VILLE_CL AS ADR_CL
```

```
FROM clients
```

```
WHERE no_CL = 1035 ;
```

# Traitement de l'absence de valeur

- Traitement de l'absence de valeur
  - **sur les comparaisons**
    - toute comparaison exprimée avec les opérateurs =, <>, >, <, <=, >=, LIKE qui comporte une expression qui n'a pas de valeur prend la valeur logique INDEFINIE
    - les comparaisons ignorent les lignes où il y a absence de valeur  
`SELECT * FROM pilotes`  
`WHERE NAISS_PIL <> 1960 AND VILLE <> 'Agadir' ;`
    - comparaisons indéfinies :  
`SELECT *`  
`FROM avions`  
`WHERE NULL = NULL OR '' = '' OR '' LIKE '%'`  
`OR 'A' LIKE '' OR 'A' NOT LIKE '' ;`

# SQL comme Langage de Requêtes

- équivalences disjonctives
  - $\text{expr NOT BETWEEN expr}_1 \text{ AND expr}_2$   
 $\Leftrightarrow \text{expr} < \text{expr}_1 \text{ OR expr} > \text{expr}_2$
  - $\text{expr IN (expr}_1 \cdots \text{expr}_N)$   
 $\Leftrightarrow \text{expr} = \text{expr}_1 \text{ OR} \cdots \text{OR expr} = \text{expr}_N$
  - $\text{expr op ANY (expr}_1 \cdots \text{expr}_N)$   
 $\Leftrightarrow \text{expr op expr}_1 \text{ OR} \cdots \text{OR expr op expr}_N$

# SQL comme Langage de Requêtes

- Les expressions suivantes :
  - $\text{expr NOT BETWEEN expr}_1 \text{ AND expr}_2$
  - $\text{expr IN (expr}_1 \cdots \text{expr}_N)$
  - $\text{expr op ANY (expr}_1 \cdots \text{expr}_N)$

sont vraies ssi  $\text{expr}$  a une valeur et si au moins une des expressions  $\text{expr}_1$ ,  $\text{expr}_N$  a une valeur qui satisfait les comparaisons

**SELECT** NUM\_PIL **FROM** pilotes

**WHERE** VILLE **IN** ('Rabat', 'Fes', ' ');

# SQL comme Langage de Requêtes

- équivalences conjonctives
  - $\text{expr BETWEEN expr}_1 \text{ AND expr}_2$   
 $\Leftrightarrow \text{expr} \geq \text{expr}_1 \text{ OR expr} \leq \text{expr}_2$
  - $\text{expr NOT IN (expr}_1 \cdots \text{expr}_N)$   
 $\Leftrightarrow \text{expr} \neq \text{expr}_1 \text{ OR} \cdots \text{OR expr} \neq \text{expr}_N$
  - $\text{expr op ALL (expr}_1 \cdots \text{expr}_N)$   
 $\Leftrightarrow \text{expr op expr}_1 \text{ AND} \cdots \text{AND expr op expr}_N$

# SQL comme Langage de Requêtes

- Les expressions suivantes :
  - $\text{expr BETWEEN expr}_1 \text{ AND expr}_2$
  - $\text{expr NOT IN (expr}_1 \cdots \text{expr}_N)$
  - $\text{expr op ALL (expr}_1 \cdots \text{expr}_N)$

sont vraies ssi  $\text{expr}$  a une valeur et si au toutes les expressions  $\text{expr}_1$ ,  $\text{expr}_N$  a une valeur qui satisfait les comparaisons

**SELECT** NUM\_PIL **FROM** pilotes

**WHERE** VILLE **NOT IN** ('Marseille', 'Nice', '');" ;

# SQL comme Langage de Requêtes

- Traitement de l'absence de valeur
  - **recherche de l'absence de valeur**

**SELECT** liste de nom\_de\_colonne **FROM** nom de table

**WHERE** expression **IS** [**NOT**] **NULL** ;

- pour connaître tous les vols auxquels on n'a pas encore affecté d'avions

**SELECT** no\_VOL

**FROM** vols

**WHERE** no\_AV **IS** NULL ;

# SQL comme Langage de Requêtes

- Traitement de l'absence de valeur
  - **Donner une valeur à l'absence de valeur**

**NVL** (expr 1 , expr 2 ) = expr 1 si elle définie, expr 2 sinon

expr 1 et expr 2 doivent être de même type

- pour qu'une capacité d'avion inconnue soit considérée comme 0

```
SELECT no_VOL, NVL(CAP,0)  
FROM avions ;
```



# SQL comme Langage de Requêtes

- Ordonner les réponses

**SELECT** liste de nom de colonne

**FROM** nom de table

[**WHERE** expression]

**ORDER BY** { expression | position } [**ASC** | **DESC**]

[{ expression | position } [**ASC** | **DESC**]] ;

# SQL comme Langage de Requêtes

- Ordonner les réponses
  - pour connaître les horaires des vols triés par ordre croissant des dates et heures de départ

```
SELECT no_VOL, D_d, D_a
```

```
FROM vols
```

```
ORDER BY D_d ;
```

# SQL comme Langage de Requêtes

- Les fonctions de groupe
  - les fonctions de groupe calculent les résultats à partir d'une collection de valeurs.

**COUNT** (\*|[DISTINCT | ALL|expression]) comptage des lignes

**COUNT** ([DISTINCT | ALL|expression]) comptage des valeurs

**MAX** ([DISTINCT | ALL|expression]) maximum des valeurs

**MIN** ([DISTINCT | ALL|expression]) minimum des valeurs

**SUM** ([DISTINCT | ALL|expression]) somme des valeurs

**AVG** ([DISTINCT | ALL|expression]) moyenne des valeurs

**STDDEV** ([DISTINCT | ALL|expression]) écart-type des valeurs

**VARIANCE** ([DISTINCT | ALL|expression]) variance des valeurs

# SQL comme Langage de Requêtes

- Les fonctions de groupe

- pour connaître le nombre d'avions

```
SELECT COUNT(*) NBR_AV
```

```
FROM avions ;
```

- pour connaître le nombre d'heures de vols du pilote 4020

```
SELECT SUM(24 *(D_a - D_d)) NBR_H
```

```
FROM vols
```

```
WHERE no_PIL = 4020 ;
```

# SQL comme Langage de Requêtes

- Les regroupements de lignes

- les fonctions de groupe calculent les résultats à partir d'une collection de valeurs.

**SELECT** liste d'expressions1

**FROM** nom de table

**GROUP BY** liste d'expressions2 ;

- les expressions de liste d'expressions1 doivent être des expressions formées uniquement :
  - d'expressions de liste d'expressions2
  - de fonctions de groupe
  - de constantes littérales

# SQL comme Langage de Requêtes

- Les regroupements de lignes
  - pour connaître le nombre d'avions affectés à chaque ville d'affectation d'un avion

```
SELECT LOC, COUNT(*) NBR_AV
```

```
FROM avions
```

```
GROUP BY LOC ;
```

# SQL comme Langage de Requêtes

- Les regroupements de lignes

- pour connaître le nombre de vols qui ont la même durée

```
SELECT 24*(D_a - D_d) DUR_VOL, COUNT(*) NBR_VOL  
FROM vols  
GROUP BY D_a - D_d ;
```

# SQL comme Langage de Requêtes

- Les regroupements de lignes

- regroupement de lignes sélectionnées

**SELECT** liste d'expressions1

**FROM** nom de table

**WHERE** condition

**GROUP BY** liste d'expressions2 ;

- pour connaître le nombre d'avions différents utilisés par chaque pilote assurant un vol

**SELECT** LOC, no\_PIL, **COUNT**(**DISTINCT** no\_AV) NBR\_AV

**FROM** vols

**WHERE** no\_PIL **IS NOT NULL**

**GROUP BY** no\_PIL ;



# SQL comme Langage de Requêtes

- Conditions sur l'ensemble des lignes
  - pour savoir si le pilote 4010 assure tous les vols avec un avion différent à chaque fois

```
SELECT 'OUI' REP
```

```
FROM vols
```

```
WHERE no_PIL = 4010
```

```
HAVING COUNT(*) = COUNT(DISTINCT no_AV) ;
```

# SQL comme Langage de Requêtes

- Conditions sur l'ensemble des lignes

**SELECT** liste d'expressions

**FROM** nom de table

[ **WHERE** condition ]

**GROUP BY** liste d'expressions2

**HAVING** condition sur lignes ;

- les expressions de liste d'expressions et condition sur lignes doivent être formées uniquement :
  - d'expressions de liste d'expressions2
  - de fonctions de groupe
  - de constantes littérales

# SQL comme Langage de Requêtes

- Conditions sur l'ensemble des lignes
  - pour connaître les pilotes qui conduisent au moins deux avions différents

```
SELECT no_PIL  
FROM vols  
WHERE no_PIL IS NOT NULL  
GROUP BY no_PIL  
HAVING COUNT(DISTINCT no_AV) >= 2 ;
```

# SQL comme Langage de Requêtes

- Opérateurs ensemblistes

SELECT liste d'expressions1

FROM nom de table

[ WHERE condition ]

[ GROUP BY liste d'expressions2]

UNION | UNION ALL | INTERSECT | MINUS

SELECT liste d'expressions3

FROM nom de table

[ WHERE condition ]

[ GROUP BY liste d'expressions4] ;

# SQL comme Langage de Requêtes

- Opérateurs ensemblistes

- pour connaître les villes qui sont soit des villes de départ soit des villes d'arrivées d'un vol

```
SELECT V_d VILL  
  
FROM vols  
  
WHERE V_d IS NOT NULL  
  
UNION  
  
SELECT V_a  
  
FROM vols  
  
WHERE V_a IS NOT NULL ;
```

# SQL comme Langage de Requêtes

- Opérateurs ensemblistes

- pour connaître le nombre de vols assurés par chaque pilote

```
SELECT no_PIL, COUNT(*) NBR_VOL
```

```
FROM vols
```

```
WHERE no_PIL IS NOT NULL
```

```
GROUP BY no_PIL
```

```
UNION ALL
```

```
(SELECT no_PIL, 0
```

```
FROM pilotes
```

```
MINUS
```

```
SELECT no_PIL, 0
```

```
FROM vols);
```

# SQL comme Langage de Requêtes

- Produit cartésien

**SELECT** liste d'expressions

**FROM** liste de(nom de table [ alias ])

[ **WHERE** condition ] ;

- pour connaître le coût de chaque classe du vol V900 lorsqu'on les applique au vol V100

**SELECT** Classe, COEF\_PRIX \* COUT\_VOL COUT

**FROM** defclasses D, vols V

**WHERE** D.no\_VOL = 'V900' AND V.no\_VOL = 'V100' :

# SQL comme Langage de Requêtes

- Opérateur de jointure naturelle

**SELECT** liste d'expressions

**FROM** liste de(nom de table [ alias ])

**WHERE** expr comp expr [ **AND** | **OR** expr comp expr ] ;

ou

**SELECT** liste d'expressions

**FROM**

nom de table [ alias ]

**INNER JOIN**

nom de table [ alias ]

**ON** expr comp expr [ **AND** | **OR** expr comp expr ] ;



# SQL comme Langage de Requêtes

- Opérateur de jointure naturelle
  - pour connaître le nombre de places de chaque vol qui a été affecté à un avion

```
SELECT no_VOL, CAP
```

```
FROM vols, avions
```

```
WHERE vols.no_AV = avions.no_AV ;
```

# SQL comme Langage de Requêtes

table vols		table avions	
no_VOL	no_AV	no_AV	CAP
V101	560	101	350
V141	101	240	NULL
V169	101	560	250
V631	NULL		
V801	240		

equi-jointure sur no\_AV

vols.no_VOL	vols.no_AV
V101	560
V141	101
V169	101
V801	240

avions.no_AV	avions.CAP
560	250
101	350
101	350
240	NULL

# SQL comme Langage de Requêtes

vols.no_VOL	vols.no_AV
V101	560
V141	101
V169	101
V801	240

avions.no_AV	avions.CAP
560	250
101	350
101	350
240	NULL

projection sur no\_VOL, CAP

vols.no_VOL	avions.CAP
V101	250
V141	350
V169	350
V801	NULL

# SQL comme Langage de Requêtes

**SELECT** liste d'expressions

**FROM** nom de table1, nom de table2

**WHERE** expr table1 comp nom table2.col(+)

[ **AND** expr table1 comp nom table2.col(+)]

ou

**SELECT** liste d'expressions

**FROM**

nom de table1 [ alias ]

**LEFT JOIN | RIGHT JOIN**

nom de table2 [ alias ]

**ON** expr comp expr [ **AND | OR** expr comp expr ] ;

# SQL comme Langage de Requêtes

- Opérateur de semi-jointure externe

pour connaître le nombre de places de chaque vol (même lorsqu'aucun avion n'est affecté au vol)

```
SELECT no_VOL, CAP
```

```
FROM vols V LEFT JOIN avions A
```

```
ON V.no_AV = A.no_AV ;
```

# SQL comme Langage de Requêtes

table vols		table avions	
no_VOL	no_AV	no_AV	CAP
V101	560	101	350
V141	101	240	NULL
V169	101	560	250
V631	NULL		
V801	240		

equi-jointure externe

vols.no_VOL	vols.no_AV
V101	560
V141	101
V169	101
V801	240
V631	NULL

avions.no_AV	avions.CAP
560	250
101	350
101	350
240	NULL
NULL	NULL

# SQL comme Langage de Requêtes

vols.no_VOL	vols.no_AV
V101	560
V141	101
V169	101
V801	240
V631	NULL

avions.no_AV	avions.CAP
560	250
101	350
101	350
240	NULL
NULL	NULL

projection sur no\_VOL, CAP

vols.no_VOL	avions.CAP
V101	250
V141	350
V169	350
V801	NULL
V631	NULL

# Fonctions diverses pour requêtes SQL

- Sous-requêtes
  - imbrication de sous-requêtes dans la clause WHERE

**SELECT** projection

**FROM** nom de table

**WHERE** condition

(**SELECT** projection

**FROM** nom de table

**WHERE** condition,  $\dots$  ) ;



# Fonctions diverses pour requêtes SQL

- Sous-requêtes : donnant une seule ligne
  - pour connaître les vols qui utilisent le même avion que celui utilisé par le vol V101

```
SELECT no_VOL
```

```
FROM vols
```

```
WHERE no_Av = (SELECT no_Av
```

```
FROM vols
```

```
WHERE no_VOL = 'V101' ) ;
```

# Fonctions diverses pour requêtes SQL

- Sous-requêtes donnant au plus une ligne
  - pour connaître le vols qui assure le même trajet que celui du vol V101 mais 2 jours plus tard

```
SELECT no_VOL
```

```
FROM vols
```

```
WHERE (V_d, V_a, D_d, D_a) =
```

```
    (SELECT (V_d, V_a, D_d+2, D_a+2)
```

```
        FROM vols
```

```
        WHERE no VOL = 'V101' ) ;
```

# Fonctions diverses pour requêtes SQL

- Sous-requêtes donnant 0, 1 ou plusieurs lignes
  - pour connaître les vols qui sont assurés par un pilote qui habite Rabat

```
SELECT no_VOL
```

```
FROM vols
```

```
WHERE no_PIL IN
```

```
    (SELECT no_PIL
```

```
      FROM pilotes
```

```
      WHERE VILLE = 'Rabat' ) ;
```

# Fonctions diverses pour requêtes SQL

- Sous-requêtes donnant 0, 1 ou plusieurs lignes
  - pour connaître les pilotes qui n'assurent aucun vol

```
SELECT no_PIL
```

```
FROM pilotes
```

```
WHERE no_PIL NOT IN
```

```
    (SELECT no_PIL
```

```
      FROM vols
```

```
      WHERE no_PIL IS NOT NULL ) ;
```

# Fonctions diverses pour requêtes SQL

- Sous-requêtes d'existence
  - pour connaître les avions qui sont conduits par au moins un pilote de Marseille

```
SELECT DISTINCT no_AV
```

```
FROM vols
```

```
WHERE EXISTS
```

```
    (SELECT *
```

```
      FROM pilotes
```

```
      WHERE no_PIL = vols.no_PIL
```

```
      AND VILLE = 'Agadir') ;
```

- une vue est une table virtuelle résultat d'une requête
- rôle d'une vue
  - réduire la complexité syntaxique des requêtes
  - définir les schémas externes.
  - définir des contraintes d'intégrité.
  - définir un niveau additionnel de sécurité en restreignant l'accès à un sous ensemble de lignes et/ ou de colonnes.

création d'une vue de schéma externe

**CREATE [OR REPLACE ] [ FORCE | NO FORCE ]**

**VIEW** nom de table [(liste de nom de colonne)]

**AS** requête **[WITH CHECK OPTION | WITH READ ONLY ] ;**

# Les vues

- **FORCE VIEW** permet de créer des vues lorsque la table ou les tables utilisées ne sont pas disponibles dans votre environnement, alors la Vue sera INVALIDE mais existante.
- Avec **NOFORCE** (valeur par défaut), si les tables n'existent pas, la vue n'est pas créée.
- Les vues créées avec l'option **WITH READ ONLY** peuvent être interrogées mais aucune opérations DML (UPDATE, DELETE, INSERT) peut être effectuées sur la Vue.
- L'option **WITH CHECK OPTION** ou **WITH CHECK OPTION CONSTRAINT** crée une contrainte de vérification sur la vue à partir de la clause WHERE.
- Les vues créées avec l'option **WITH CHECK OPTION CONSTRAINT** empêche toutes mises à jour de la Vue si les conditions de la clause WHERE ne sont pas respectées. (Visible dans USER\_CONSTRAINTS en type V).
- L'option **OR REPLACE** permet de changer la définition d'une vue sans faire un **DROP/CREATE**. L'avantage de l'option **OR REPLACE**, c'est la non suppression des privilèges accordés à la vue, cependant les objets dépendant de la vue deviennent invalides et doivent être compilés.



# Les vues

- création de la vue correspondant aux vols qui partent de Rabat

```
CREATE VIEW vols_d_Rabat  
AS SELECT no_VOL, V_a, H_d, H_a  
FROM vols  
WHERE V_d = 'Rabat';
```

- interroger une vue

- interrogation de la vue correspondant aux vols qui partent de Paris

```
SELECT * FROM vols_d_Rabat;
```

- supprimer une vue

```
DROP VIEW nom_de_vue;
```

- suppression de la vue correspondant aux vols qui partent de Rabat

```
DROP VIEW vols_d_Rabat;
```

# Les vues – Exemples

- pour connaître les pilotes qui assurent le plus grand nombre de vols

```
CREATE VIEW nbre_de_vols_par_pil
AS SELECT no_PIL, count(*) AS NBR_VOLS
FROM vols V
WHERE no_PIL IS NOT NULL
GROUP BY no_PIL ;

SELECT no_PIL FROM nbre_de_vols_par_pil
WHERE NBR_VOLS =
    (SELECT max( NBR_VOLS)
     FROM nbre_de_vols_par_pil) ;
```

# Les vues : mise à jour

- opérations sur les vues

INSERT

UPDATE

DELETE

- **restrictions** : Ces instructions ne s'appliquent pas aux vues qui contiennent :
  - une jointure
  - un opérateur ensembliste : UNION, INTERSECT, MINUS
  - une clause GROUP BY, CONNECT BY, ORDER BY ou START WITH
  - la clause DISTINCT, une expression ou une pseudo-colonne dans la liste de sélection des colonnes.

# Requêtes avec vues

- création de la vue pour la personne qui définit les vols

```
CREATE VIEW def_vols  
AS SELECT no_VOL, V_d, D_d, V_a, D_a  
FROM vols  
WHERE no_VOL IS NULL AND no_PIL IS NULL ;
```

- définir un nouveau vol

```
INSERT INTO def_vols VALUES  
( '999', 'Agadir', to_date('01/05/07 10 :30', 'DD/MM/RR HH  
:MI'), 'Rabat', to_date('01/05/07 10 :30', 'DD/MM/RR HH :MI')) ;
```

# Requêtes avec vues

- supprimer un vol non affecté

```
DELETE FROM def_vols  
WHERE no_VOL = 'V998' ;
```

- modifier un vol non affecté

```
UPDATE def_vols  
SET D_d= D_d + 1 / 24, D_a= D_a + 1 / 24 WHERE  
no_VOL = 'V998' ;
```

- connaître les vols non affectés

```
SELECT * FROM def_vols ;
```

# Requêtes avec vues

- création de la vue pour la personne qui affecte un avion et un pilote à un vol

```
CREATE VIEW affect_vols  
AS SELECT no_VOL, no_AV, no_PIL  
FROM vols ;
```

- affecter un avion et un pilote à un nouveau vol

```
UPDATE affect_vols  
SET no_AV = 101, no_PIL = 5050  
WHERE no_VOL = 'V999'  
AND no_AV IS NUL AND no_PIL IS NULL ;
```

- affecter un nouvel avion à un vol

```
UPDATE affect_vols  
SET no_AV = 202  
WHERE no_VOL = 'V999' ;
```

# Requêtes avec vues

- permuter l'affectation des pilotes de 2 vols

```
UPDATE affect_vols A1
```

```
SET no_PIL =
```

```
( SELECT no_PIL
```

```
FROM affect_vols A2
```

```
WHERE
```

```
(A1.no_VOL = 'V100' AND A2.no_VOL = 'V200')
```

```
OR
```

```
(A1.no_VOL = 'V200' AND A2.no_VOL = 'V100')
```

```
)
```

```
WHERE no_VOL = 'V100' OR no_VOL = 'V200' ;
```

# Les vues : contrôle de mise à jour

- vérification des contraintes de domaine (**interdiction des valeurs inconnues**)

```
CREATE VIEW a_avions  
AS SELECT * FROM avions  
WHERE  
    no_AV > 0  
    AND CAP > 1  
    AND NOM_AV IN ('Airbus', 'Boeing', 'Caravelle')  
WITH CHECK OPTION ;
```



# Requêtes avec vues

- Vérification des contraintes de domaine (**autorisation des valeurs inconnues**)

```
CREATE VIEW aa_avions
AS SELECT * FROM avions
WHERE
    no_AV > 0
    AND (CAP IS NULL OR CAP > 1)
    AND (NOM_AV IS NULL OR IN ('Airbus', 'Boeing',
    'Caravelle'))
WITH CHECK OPTION ;
```

# Requêtes avec vues

- Contraintes de référence

- 1) valider l'insertion dans la table référençant
- 2) valider la suppression dans la table référencée

**règle d'adéquation : les insertions et les suppressions se font toujours au travers des vues**

# Requêtes avec vues

- Exemple : expression de clés étrangères de la relation vols
  - validation des insertions dans vols

```
CREATE VIEW a_avions
```

```
AS SELECT * FROM vols
```

```
WHERE
```

```
no_AV > 0
```

```
AND (no_PIL IS NULL OR no_PIL IN (SELECT no_PIL FROM  
pilotes))
```

```
AND (NOM_AV IS NULL OR IN (SELECT NOM_AV FROM avions))
```

```
WITH CHECK OPTION ;
```

# Requêtes avec vues

- Exemple : expression de clés étrangères de la relation vols
  - validation des suppressions dans avions et dans pilotes

```
CREATE VIEW d_avions
```

```
AS SELECT * FROM avions A
```

```
WHERE NOT EXISTS ( SELECT * FROM vols V WHERE  
                    A.no_AV = V.no_AV) ;
```

```
CREATE VIEW d_pilotes
```

```
AS SELECT * FROM pilotes P
```

```
WHERE NOT EXISTS ( SELECT * FROM vols V WHERE  
                    P.no_PIL = V.no_PIL) ;
```

# Requêtes avec vues

- Expression de contraintes générales
- Exemple : empêcher l'affectation d'un même avion à deux vols différents dont les tranches horaires se chevauchent

```
CREATE VIEW a_vols
AS SELECT * FROM vols V1
WHERE NOT EXISTS (
    SELECT * FROM vols V2
    WHERE V1.no AV = V2.no AV
    AND NVL(V2.D_a, V2.D_d) >= NVL(V1.D_d, V1.D_a)
    AND NVL(V1.D_a, V1.D_d) >= NVL(V2.D_d, V2.D_a)
)
WITH CHECK OPTION ;
```

# SQL : langage de contrôle de données (LCD)

- Sécurité des données

- confidentialité

- gestion des rôles et des utilisateurs

- attribution de privilèges aux rôles et aux utilisateurs

- définition de filtres (protection de données confidentielles, contrôle d'intégrité)

- pérennité

- gestion des transactions

- intégrité

- gestion des transactions

# SQL : langage de contrôle de données (LCD)

- **transaction** : séquence d'opérations manipulant des données vérifient les propriétés suivantes :
  - atomicité
  - cohérence
  - indépendance
  - permanence
- **contrôle des transactions** :
  - **COMMIT** : valide la transaction en cours
  - **ROLLBACK** : annule la transaction en cours

# SQL : langage de contrôle de données (LCD)

- On distingue typiquement six types de commandes SQL de contrôle de données :
  - **GRANT** : autorisation d'un utilisateur à effectuer une action ;
  - **DENY** : interdiction à un utilisateur d'effectuer une action ;
  - **REVOKE** : annulation d'une commande de contrôle de données précédente ;
  - **COMMIT** : validation d'une transaction en cours ;
  - **ROLLBACK** : annulation d'une transaction en cours ;
  - **LOCK** : verrouillage sur une structure de données.



# Gestion des utilisateurs et des privilèges

- Rôles et privilèges sont définis pour sécuriser l'accès aux données de la base
- Ces concepts sont mis en oeuvre pour protéger les données en accordant (ou retirant) des privilèges à un utilisateur ou un groupe d'utilisateurs
- Un rôle est un regroupement de privilèges. Une fois créé il peut être assigné à un utilisateur ou à un autre rôle
- Les privilèges sont de deux types
  - ***Les privilèges de niveau système***  
Qui permettent la création, modification, suppression, exécution de groupes d'objets
  - ***Les privilèges de niveau objet***  
Qui permettent les manipulations sur des objets spécifiques

# Gestion des utilisateurs et des privilèges

- création de rôle

`CREATE ROLE` nom-de-rôle [`IDENTIFIED BY` mot-de passe ] ;

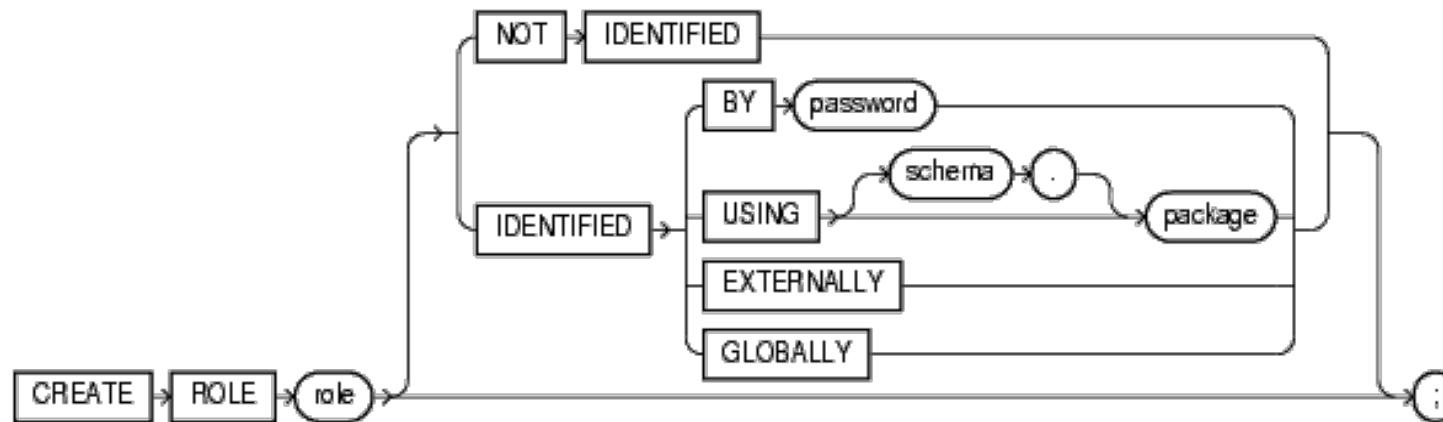
- ajout, modification, suppression de mot de passe

`ALTER ROLE` nom-de-rôle [`IDENTIFIED BY` mot-de passe ] ;

- suppression de rôle

`DROP ROLE` nom-de-rôle ;

## Créer des rôles et leur assigner des privilèges



- **role** représente le nom du rôle
- **NOT IDENTIFIED** (défaut) indique qu'aucun mot de passe n'est nécessaire pour activer le rôle
- **IDENTIFIED BY password** indique qu'un mot de passe est nécessaire pour activer le rôle
- **IDENTIFIED USING package** indique qu'un package va être utilisé pour fixer les droits de l'utilisateur
- **IDENTIFIED EXTERNALLY** indique que l'autorisation provient d'une source externe (S.E.)
- **IDENTIFIED GLOBALLY** pour un user **GLOBAL** géré par exemple par Enterprise Directory Service

# Exemples

- **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 ;**

# Gestion des utilisateurs et des privilèges

- création d'utilisateur

`CREATE USER` nom-d'utilisateur [`IDENTIFIED BY` mot-de passe ] ;

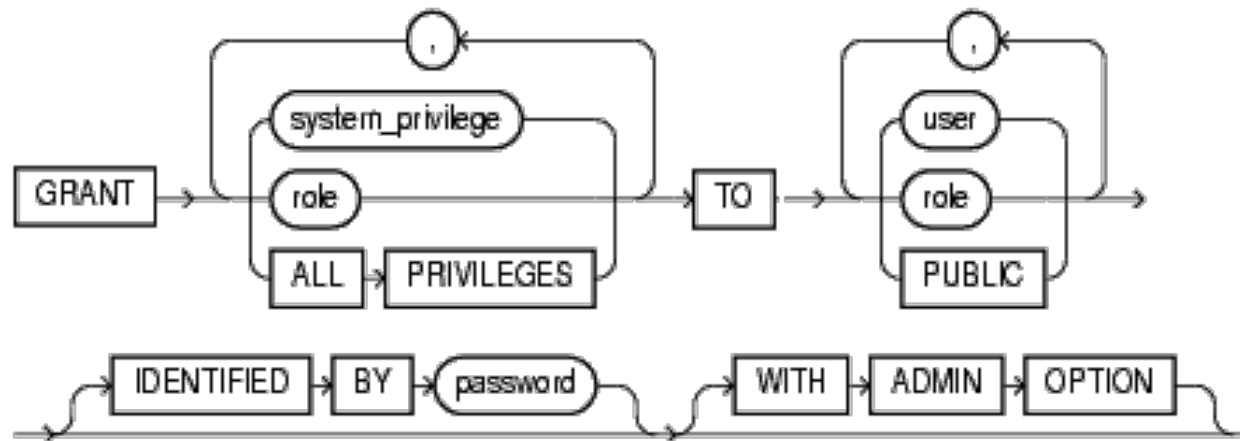
- ajout, modification, suppression de mot de passe

`ALTER USER` nom-d'utilisateur [`IDENTIFIED BY` mot-de passe ] ;

- suppression de rôle

`DROP USER` nom-d'utilisateur ;

# Assigner des privilèges système à un utilisateur



- **system\_privilege** représente un privilège système
- **role** représente un rôle préalablement créé
- **ALL PRIVILEGES** représente tous les privilèges système (à l'exception de SELECT ANY DICTIONARY)
- **user** représente le nom de l'utilisateur qui doit bénéficier du privilège
- **PUBLIC** assigne le privilège à tous les utilisateurs
- **WITH ADMIN OPTION** assigne à l'utilisateur le droit d'assigner, de retirer, de modifier et de supprimer à son tour les privilèges du rôle reçus

# Assigner des privilèges système à un utilisateur

- attribution de privilèges

GRANT systeme-privileges | ALL [privileges ]

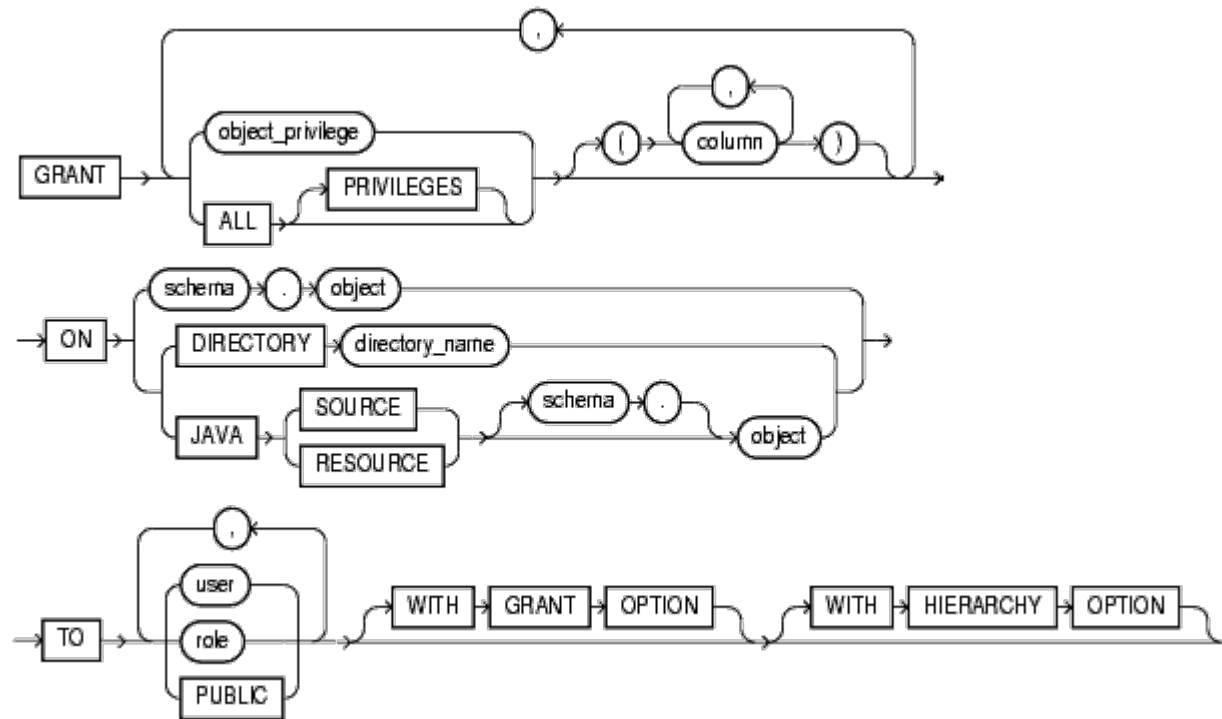
TO liste-roles-utilisateurs | PUBLIC

[WITH ADMIN OPTION ] ;

- systeme-privileges :

- CREATE ROLE
- CREATE SEQUENCE
- CREATE SESSION
- CREATE SYNONYM
- CREATE PUBLIC SYNONYM
- CREATE TABLE
- CREATE USER
- CREATE VIEW

# Assigner des privilèges objet à un utilisateur





# Assigner des privilèges objet à un utilisateur

- `object_privilege` représente un privilège objet
- `role` représente un rôle préalablement créé
- `ALL PRIVILEGES` représente tous les privilèges assignés à l'exécuteur de l'instruction
- `column` représente le nom de colonne d'une table
- `schema` représente le nom d'un schéma
- `object` représente le nom d'un objet du schéma
- `directory_name` représente le nom d'une directory
- `JAVA SOURCE` représente le nom d'une source Java
- `JAVA RESOURCE` représente le nom d'une ressource Java
- `WITH GRANT OPTION` assigne à l'utilisateur le droit d'assigner à son tour le privilège reçu à un autre utilisateur
- (`WITH GRANT OPTION` s'applique à un utilisateur ou à `PUBLIC`, mais pas à un rôle)
- `WITH HIERARCHY OPTION` assigne les privilèges aux sous-objets

# Assigner des privilèges objet à un utilisateur

- attribution de privilèges sur des objets oracle

GRANT liste-droits

ON nom-composant

TO liste-roles-utilisateurs

[WITH GRANT OPTION ] ;

- liste-droits :

- SELECT

- INSERT

- UPDATE

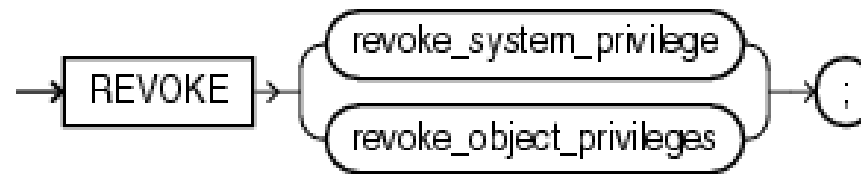
- DELETE

- ALTER

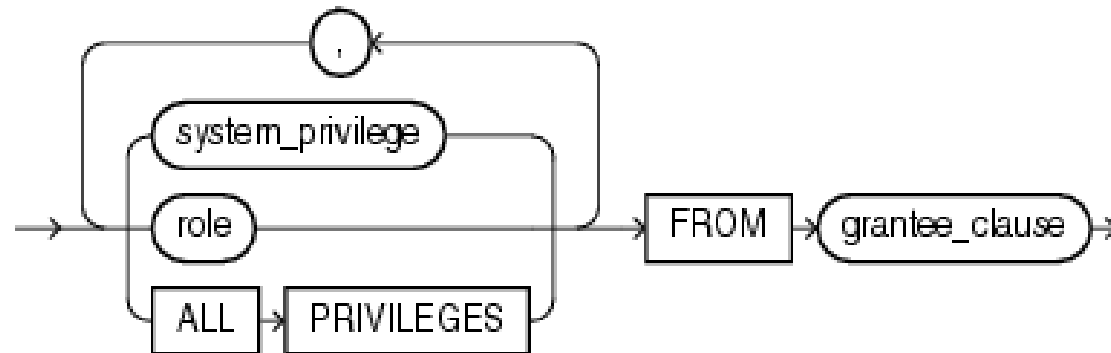
- REFERENCES

- ALL [PRIVILEGES ]

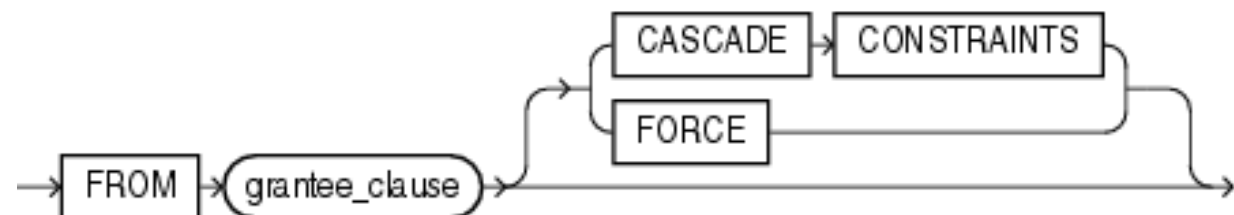
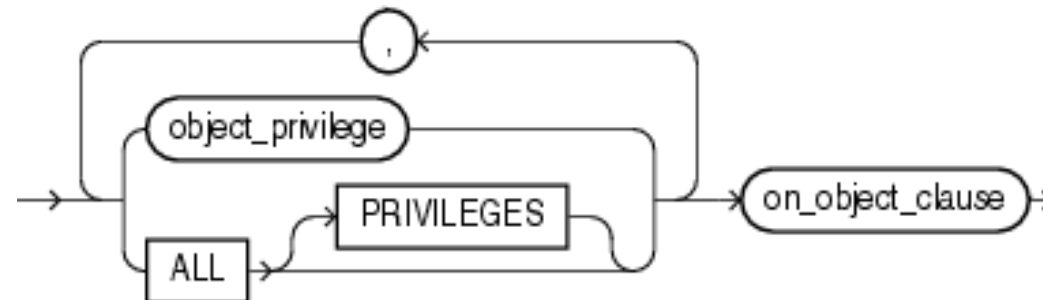
# Annulation des privilèges



revoke\_system\_privileges::=

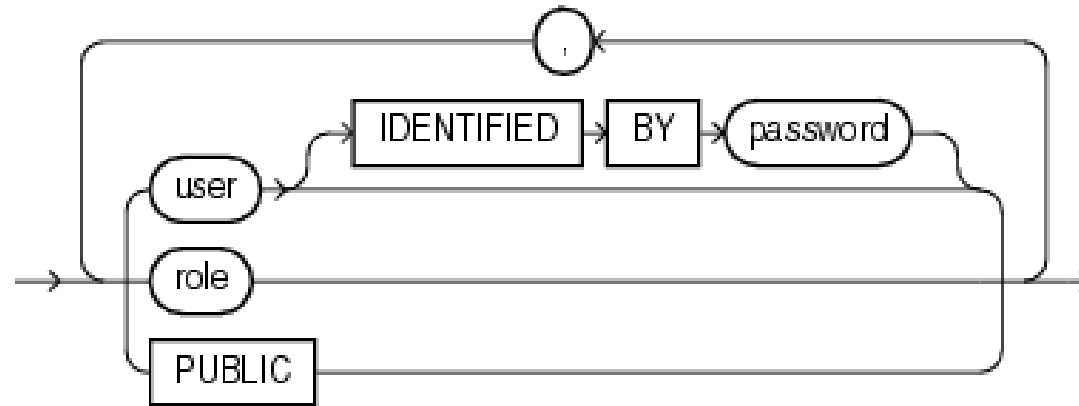


revoke\_object\_privileges::=

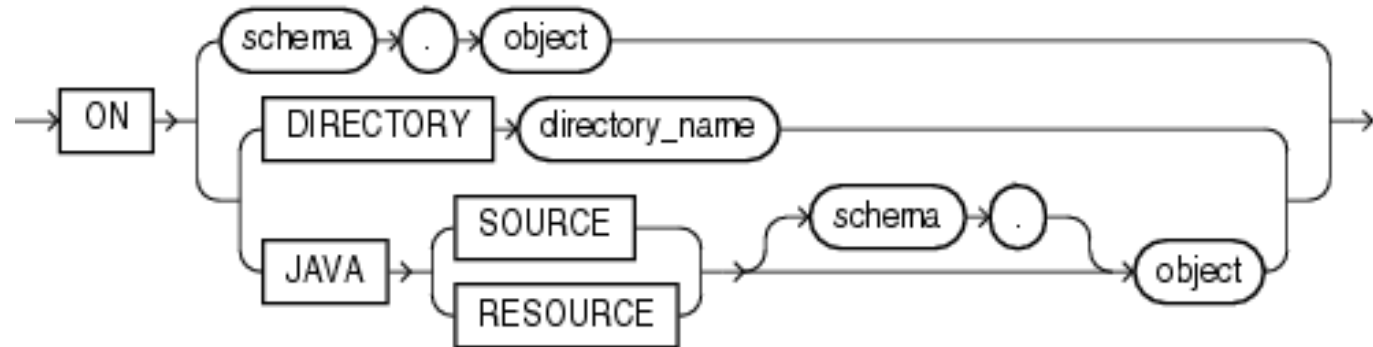


# Annulation des privilèges

grantee\_clause::=



on\_object\_clause::=



# Différence entre GRANT, DENY & REVOKE

- **Grant** est utilisé pour autoriser spécifiquement l'accès à un objet.
- **Deny** est utilisé pour empêcher spécifiquement l'accès à un objet.
- **Revoke** est utilisé pour supprimer l'accès spécifiquement accordé ou refusé à un objet.
- La différence entre **GRANT, DENY et REVOKE** peut également être indiquée dans les mots ci-dessous
- Lorsque l'autorisation est accordée, l'utilisateur ou le rôle reçoit l'autorisation d'effectuer un action, tel que la création d'une table.
- L'instruction DENY refuse l'autorisation sur un objet et empêche le principal d'obtenir l'autorisation GRANT en fonction de l'appartenance à un groupe ou à un rôle.
- L'instruction DENY refuse l'autorisation sur un objet et empêche le principal d'obtenir l'autorisation GRANT en fonction de l'appartenance à un groupe ou à un rôle.
- L'instruction REVOKE supprime une autorisation précédemment accordée ou refusée.
- Les autorisations refusées à une portée plus élevée dans un modèle de sécurité

## Remarque

- DENY n'existe pas en SQL. C'est une invention de Sybase / SQL Server...
- en SQL il n'existe que GRANT et REVOKE.

```
SELECT * FROM test
```

Le schéma courant est user1

Lorsque le schéma de l'objet n'est pas précisé (ici l'objet TEST), Oracle effectue la recherche de cet objet dans l'ordre suivant :

- Recherche d'un objet appartenant au schéma courant user1
- Recherche dans les synonymes du schéma courant
- Dans les synonymes PUBLIC

- Pour plus de détaille sur la création, activation, modification, suppression des rôles veuillez voir le lien suivant :  
<https://oracle.developpez.com/guide/administration/adminrole/>



## Ch. III. Procédures stockées et Déclencheurs(Triggers)

- Gestion des procédures et fonctions
- Métadonnées des procédures et fonctions
- Extensions au standard SQL (Délimiteur, contrôle flux, boucles, Curseurs)
- Structure d'un déclencheur
- Types d'événements déclencheurs
- Propriétés d'un déclencheur
- Validation des données

# Introduction

- Comme en programmation : code SQL souvent répété.
- Nécessité d'exécuter plusieurs requêtes pour une seule tâche.
- Solution :
  - Créer (comme en programmation):
    - Des fonctions
    - Des procédures
- Conséquences:
  - Allègement maxi des traitements sur le client
  - Structuration plus propre du code SQL

# Introduction

- Une procédure stockée est un groupe d'instructions qui est compilé une fois pour toutes et qui peut être exécuté plusieurs fois. Lorsqu'elle est exécutée, les performances sont améliorées car ses instructions ne doivent pas être recompilées.
- Un déclencheur est un type spécifique de procédure stockée qui n'est pas appelé directement par un utilisateur. Lorsque le déclencheur est créé, il est défini de façon à se déclencher lorsqu'un certain type de modification de données est effectué dans une table ou une colonne spécifique.
- Elles sont compilées puis stockées sur le serveur de BD.
- Implique une vitesse exécution accrue.
- Dans ce cours nous nous intéressons au langage PL/SQL

# Introduction

- Le PL/SQL (Procedural Language extensions to SQL) est un langage procédural d'ORACLE.
- L'intérêt du PL/SQL est de pouvoir dans un même traitement combiner la puissance des instructions SQL et la souplesse d'un langage procédural.
- Le fonctionnement de PL/SQL est basé sur l'interprétation d'un bloc de commandes. Ce mode de fonctionnement permet d'obtenir des gains de transmissions et des gains de performances.
- Dans l'environnement SQL, les ordres du langage sont transmis et exécutés les uns à la suite des autres.
- Dans l'environnement PL/SQL, les ordres SQL et PL/SQL sont regroupés en blocs; un bloc ne demande qu'un seul transfert exécution de l'ensemble des commandes contenues dans le bloc.

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

# Structure d'un bloc PL/SQL

## DECLARE

Déclarartion des variables locales au bloc, constantes, execptions, curseurs, modules locaux

## BEGIN

Commandes exécutable: Instructions SQL et PL/SQL. Possiblité de blocs imbriqués.

## EXCEPTION

La récupération et traitement des erreurs

END;

/

- **Sous SQL\*PLUS, il faut taper une dernière ligne contenant « / » pour compiler un bloc pl/sql**

# Structure d'un bloc PL/SQL

- Un bloc PL/SQL peut contenir:
  - Toute instruction du LMD (`SELECT`, `INSERT`, `UPDATE`, `DELETE`)
  - Les commandes de gestion des transactions ( `COMMIT`, `ROLLBACK`, `SAVEPOINT` )
- Les sections `DECLARE` et `EXCEPTION` sont optionnelles.
- **Chaque instruction se termine par ;**
- Les blocs peuvent être imbriqués
  - Les sous blocs ont la même structure que les blocs.
  - Une variable est visible dans le bloc où elle est déclarée et dans tous ses sous-blocs.
  - Si une variable est déclarée dans un premier bloc et aussi dans un sous bloc, la variable du bloc supérieur n'est plus visible dans le sous-bloc.

# Affichage

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

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
c varchar2(15) := 'Hello World !';
```

```
BEGIN DBMS_OUTPUT.PUT_LINE(c);
```

```
END; /
```

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
c varchar2(15) := 'Hello World !';
```

```
BEGIN
```

```
    DBMS_OUTPUT.PUT(c);
```

```
    DBMS_OUTPUT.NEW_LINE ;
```

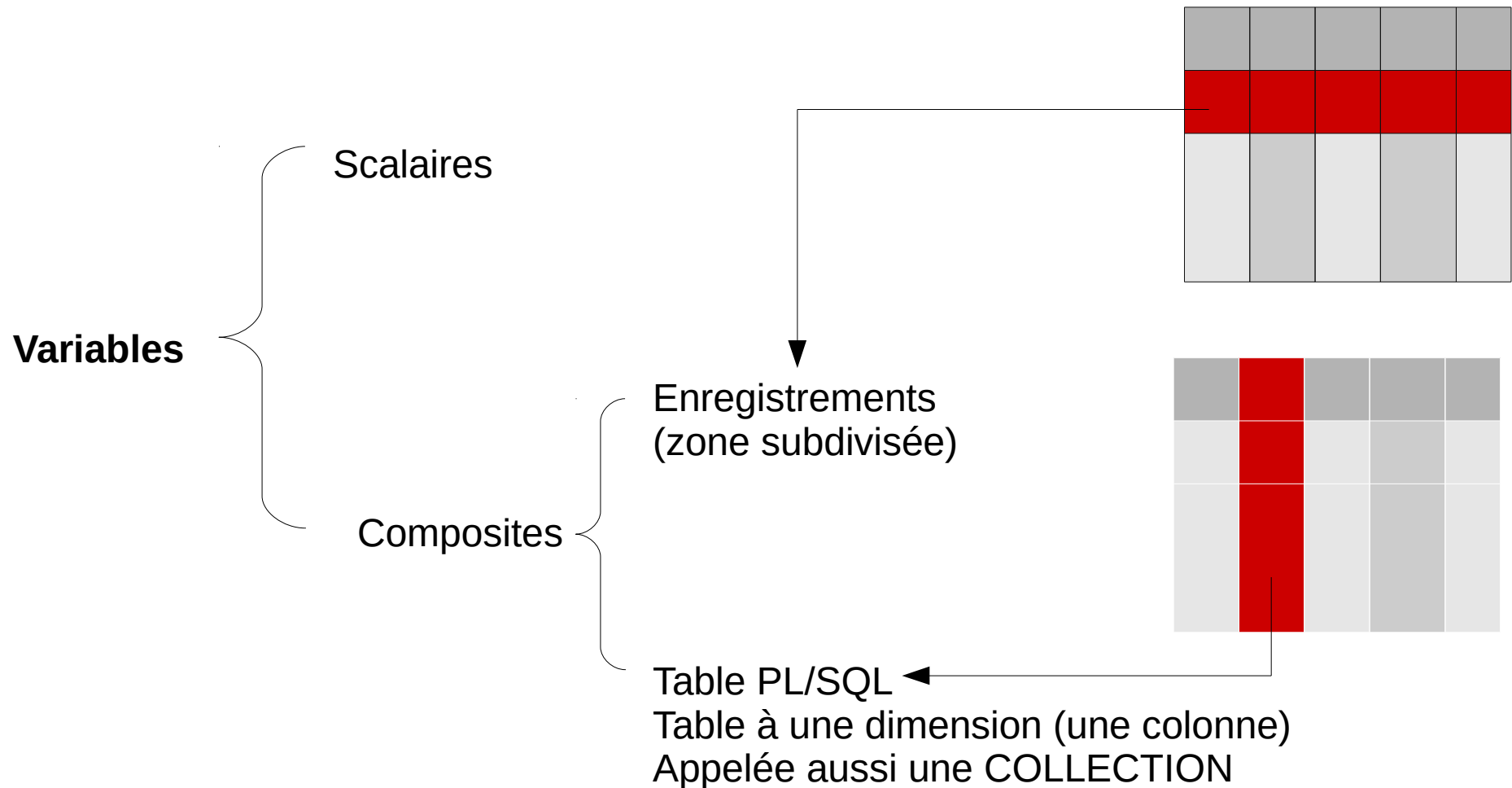
```
END; /
```



# Structure d'un bloc PL/SQL

- Variables
  - Identificateurs Oracle :
    - 30 caractères au plus
    - commence par une lettre
    - peut contenir lettres, chiffres, \_, \$ et #
  - Pas sensible à la casse
  - Doivent être déclarées avant d'être utilisées
- Commentaires
  - -- Pour un commentaire sur une ligne
  - /\* Pour commentaire sur plusieurs lignes \*/

# DÉCLARATION DES VARIABLES



# Structure d'un bloc – types de variables

- Types de variables
  - Les types habituels correspondants aux types SQL2 ou Oracle : `integer`, `varchar`,...
  - Types composites adaptés à la récupération des colonnes et lignes des tables SQL : `%TYPE`, `%ROWTYPE`
  - Type référence : `REF`

# Déclaration des variables scalaires

- Déclaration par copie du type de donnée:
  - PL/ SQL permet de déclarer un type de variable par copie. La variable fait référence au même type qu'une colonne ou même type qu'une autre variable .
  - **Syntaxe:**  
Nom-de-variable    nom\_table.nom-colonne%type ;  
  
**ou**  
Nom-de-variable1    Nom-de-variable2%type ;
  - **Exemple:**  
DECLARE  
  
    Nom\_client    client.nom%type ;  
  
    X number (10,3) ;  
  
    Y X%type ;

# Structure d'un bloc – L'attribut %ROWTYPE

- L'attribut **% ROWTYPE** permet de déclarer une variable de même type que l'enregistrement

Syntaxe: Nom\_de\_variable nom\_table **%ROWTYPE** ;

Exemple:

**DECLARE** enrg\_empl empl**%ROWTYPE** ;

- Les éléments de la structure sont identifiés par: nom\_variable . nomcolonne

# Utilisation d'une variable de type Record

- Un record permet de définir des types composites

- Syntaxe :

```
TYPE nom_record IS RECORD
```

```
( nom_ch 1 type ,
```

```
  nom_ch2 type,
```

```
.....);
```

- Déclaration d'une variable de ce type: `nom_variable nom_record`

```
DECLARE
```

```
    TYPE enrg_empl IS RECORD
```

```
        ( nom employe. nomempl%TYPE,
```

```
          salaire employe . sal %TYPE);
```

```
    e enrg_empl ;
```

# Déclaration des variables scalaires

- La partie déclarative dans un bloc PL/SQL, peut comporter 3 types de déclarations:
  - Déclarations des variables, constantes
  - Déclaration des curseurs
  - Déclaration des exceptions

## 1. Variables ou constantes

- Syntaxe:

**Nom-de-variable TYPE [CONSTANT] [NOT NULL] [:= | DEFAULT initial\_value] ;**

Expression: peut être une constante ou un calcul faisant éventuellement référence à une variable précédemment déclarée

### Exemple:

**DECLARE**

Nom\_duclient **char** (30);

X **number** **DEFAULT** 10 ;

PI **constant** **number**(7,5):=3.14159;

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

# Déclaration des variables scalaires

- Valorisation des variables: 3 possibilités:

- Par l'opérateur d'affectation **:=**
- Par la clause **Select....Into**
- Par le traitement d'un **curseur** dans la section **BEGIN**.

- Par l'opérateur d'affectation **:=**

- Nom\_variable**:=** expression
- **Exemple :**

```
X := 0;
```

```
Y := ( X+5) * Y ;
```

- Par la clause **Select....Into**

- **Exemple:**

```
DECLARE
```

```
Vref NUMBER(5,0);
```

```
Vprix produit.PU%type ;
```

```
BEGIN
```

```
SELECT REF, PU INTO Vref, Vprix
```

```
FROM Produit WHERE REF= 1 ;
```

```
dbms_output.put_line('NUMPROD='||Vref||' PU='||vprix);
```

```
END;
```



# Les collections VARRAY – définition & allocation

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

**TYPE** nom\_type **IS** {**VARRAY** | **VARYING ARRAY**} (size\_limit) **OF** typeElements [**NOT NULL**];

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

- Exemple

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

- Allocation d'un tableau

**DECLARE**

**TYPE** numberTab **IS VARRAY** (10) **OF** **NUMBER**; -- définition de type numberTab

tab numberTab ; -- déclaration de tableau tab

**BEGIN**

tab := numberTab ( ) ; -- allocation de l'espace mémoire avec constructeur de type

/\* utilisation du tableau \*/

**END**;

/

# Les collections VARRAY - Dimensionnement

**DECLARE**

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

```
tab numberTab ;
```

**BEGIN**

```
tab := numberTab ( ) ;
```

```
tab.EXTEND( 4 ) ;
```

```
/* utilisation du tableau */
```

**END;**

/

- Dans cet exemple, tab.EXTEND(4) permet par la suite d'utiliser les éléments du tableau t(1), t(2), t(3) et t(4). **Il n'est pas possible "d'étendre" un tableau à une taille supérieure à celle spécifiée lors de la création du type tableau associé.**
- On peut aussi utiliser le constructeur avec des valeurs d'initialisations.

```
tab := numberTab (1,2,3,4) ;
```

# Les collections TABLE

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

**TYPE** nom\_type **IS TABLE OF** type\_des\_valeurs **INDEX BY** type\_des\_clés

- **nom** est un nom de variable habituel.
- **type\_des\_valeurs** est un type de variable Oracle : NUMBER ou VARCHAR(longueur), par exemple. Ce sont les valeurs que l'on va récupérer de la base : on peut donc utiliser aussi la syntaxe colonne %TYPE.
- **type\_des\_clés** est **BINARY\_INTEGER** ou **VARCHAR(10)** par exemple. Ce sont les clés du tableau associatif, elles doivent être uniques !
- Les types admis doivent être numériques (**BINARY\_INTEGER** ou **PLS\_INTEGER**) ou alphabétique (**VARCHAR(longueur)**, **STRING**, **LONG** ou **VARCHAR2(longueur à préciser obligatoirement)**).
- Si on veut mettre autre chose (dates), il faudra convertir.

# Les collections TABLE

DECLARE

-- Type Tableau de chaines de 20 caractères maxi

TYPE nom\_type\_tableau is TABLE OF VARCHAR(20) INDEX BY BINARY\_INTEGER;

tab nom\_type\_tableau ; -- Déclaration de la variable tab de type nom\_type\_tableau

-- Variable tab2 de type nom\_type\_tableau initialisée avec des valeurs

tab2 nom\_type\_tableau := nom\_type\_tableau('l1', 'l2') ;

--tab2 nom\_type\_tableau ;

BEGIN

tab2 := nom\_type\_tableau('l1', 'l2') ;

tab(1) := 'ligne 1' ;

tab(2) := 'ligne 2' ;

-- Affichage du premier élément de la variable tab, c'est à dire 'ligne 1'

DBMS\_OUTPUT.put\_line(tab(1)) ;

END; /

# Les collections TABLE

Déclaration d'un tableau : nom\_Tableau nom\_Type

Exemple

**declare**

-- collection de type nested table

**TYPE** TYP\_NES\_TAB **is table of** varchar2(100) ;

-- collection de type index by

**TYPE** TYP\_IND\_TAB **is table of** number **index by** binary\_integer ;

tab1 **TYP\_NES\_TAB** ;

tab2 **TYP\_IND\_TAB** ;

**Begin**

tab1 := **TYP\_NES\_TAB**('Lundi','Mardi','Mercredi','Jeudi') ;

**for** i **in** 1..10 **loop**

    tab2(i) := i ;

**end loop** ;

**End;**

/

# Les collections TABLE

- -- table de multiplication par 8 et par 9...
- **declare**
- **type** tablemul **is record** ( par8 number, par9 number);
- **type** tabledentiers **is table of** tablemul **index by binary\_integer**;
- ti tabledentiers;
- i number;
- **begin**
- **for** i **in** 1..10 **loop**
- ti(i).par9 := i\*9 ;
- ti(i).par8:= i\*8;
- **dbms\_output.put\_line** (i||'\*8='||ti(i).par8||' '||i||'\*9='||ti(i).par9 );
- **end loop**;
- **end**;
- /

# méthodes intégrées pour les collections

- Un constructeur de collection (constructeur) est une fonction définie par le système avec le même nom qu'un type de collection, qui renvoie une collection de ce type. La syntaxe d'un appel de constructeur est:

**collection\_type ( [ value [, value ]... ] )**

- EXISTS - Utilisé pour déterminer si un élément spécifique d'une collection existe. EXISTS est utilisé avec des tables imbriquées.
- COUNT - Retourne le nombre d'éléments actuellement contenus dans une collection sans inclure les valeurs NULL. Pour varrays count est égal à LAST. Pour les tables imbriquées, COUNT et LAST peuvent être différents en raison de valeurs supprimées dans les sites de données interstitielles dans la table imbriquée.
- LIMIT - Utilisé pour VARRAYS pour déterminer le nombre maximum de valeurs autorisées. Si LIMIT est utilisé sur une table imbriquée, elle renverra une valeur nulle.
- FIRST et LAST - Renvoie les plus petits et les plus grands numéros d'index pour la collection référencée. Naturellement, ils renvoient null si la collection est vide.
- VARRAYS FIRST retourne toujours 1, pour les tables imbriquées FIRST renvoie la valeur du premier point rempli pour cette entrée. LAST renvoie la dernière instance remplie d'un VARRAY et d'une table imbriquée. Pour un VARRAY COUNT sera toujours égal à DERNIER. Pour une table imbriquée, ils peuvent être différents, mais LAST doit toujours être supérieur à COUNT s'ils sont différents pour une table imbriquée.

# méthodes intégrées pour les collections

- **PRIOR et NEXT** - Renvoie la valeur précédente ou suivante en fonction de la valeur d'entrée de l'index de collection. PRIOR et NEXT ignorent les instances supprimées d'une collection.
- **EXTEND** - Ajoute des instances à une collection. EXTEND a trois formes, EXTEND, qui ajoute une instance nulle, EXTEND (n) qui ajoute "n" instances NULL et EXTEND (n, m) qui ajoute N copies de l'instance "m" à la collection. Pour les formes de collections spécifiées non nulles, les formes un et deux ne peuvent pas être utilisées.
- **TRIM** - Trim supprime les instances d'une collection. TRIM utilisé sans argument supprime la dernière instance, TRIM (n) supprime "n" instances de la collection.
- **DELETE** - DELETE supprime les éléments spécifiés d'une table imbriquée ou d'une table
- **VARRAY**. DELETE spécifié sans argument supprime toutes les instances d'une collection. Pour les tables imbriquées, seul DELETE (n) supprime la nième instance et DELETE (n, m) supprime la nième à travers les instances mth de la table imbriquée qui se rapportent à la fiche spécifiée.



# Collections & Exception

- Les méthodes de collecte peuvent générer les exceptions suivantes:
  - **COLLECTION\_IS\_NULL** - générer lorsque la collection référencée est nulle.
  - **NO\_DATA\_FOUND** - L'indice indique une instance nulle de la collection.
  - **SUBSCRIPT\_BEYOND\_COUNT** - L'indice spécifié dépasse le nombre d'instances de la collection.
  - **SUBSCRIPT\_OUTSIDE\_LIMIT** - L'indice spécifié est en dehors de la plage autorisée (généralement reçue des références VARRAY)
  - **VALUE\_ERROR** - L'indice est nul ou n'est pas un nombre entier.

# Comment parcourir un tableau associatif ?

- Accées direct à une valeur en utilisant la clé

nom\_Tableau(clé)

- FIRST et LAST

- Un tableau associatif (comme toutes les autres collections en PL/SQL) a les propriétés FIRST et LAST auxquelles on accède en faisant nom\_tableau.FIRST et nom\_tableau.LAST.
- FIRST renvoie **la première clé** issue d'un tri numérique ou alphabétique.
- Pour parcourir le tableau, on peut donc faire une boucle comme ceci :

```
FOR i IN nom.FIRST..nom.LAST LOOP
```

- PRIOR et NEXT

```
i := nom.FIRST;
```

```
WHILE i IS NOT NULL LOOP
```

```
...
```

```
i := nom.NEXT(i);
```

```
END LOOP;
```

- Cette boucle s'arrête au dernier élément parce que NEXT(i) renvoie NULL au dernier i.

- Opérateurs logiques **NOT, AND, OR**
- Opérateurs arithmétiques **+ - \* / \*\*** (puissance)
- Opérateurs de comparaison **=, >, !=, >, <=, >=, IS NULL, LIKE, BETWEEN**
- Opérateurs de chaîne de caractères **||** (Concaténation)

Utiliser des parenthèses pour simplifier l'écriture et la lecture des expressions

# Structures de contrôles

- Structure alternative
  - IF .... THEN.... END IF
  - IF .... THEN.... ELSE ... END IF
- Itérations
  - LOOP ..
  - FOR i.....
  - WHILE
- Branchement séquentiels
  - GOTO
  - NULL

# Structures alternative

- IF THEN

```
IF condition THEN  
    sequence_insts;  
END IF
```

- IF THEN ELSE

```
IF condition THEN  
    sequence_insts1;  
ELSE  
    sequence_insts2;  
END IF
```

- IF THEN ELSIF

```
IF condition1 THEN  
    sequence_insts1;  
ELSIF condition2 THEN  
    sequence_insts2;  
ELSE  
    sequence_insts3;  
END IF
```

# Structures alternative

- Exemple:

```
if var1>10 then
```

```
    var2:= var1+20;
```

```
elseif var1 between 7 and 8 then
```

```
    var2:= 2* var1;
```

```
else
```

```
    var2:=var1*var1;
```

```
end if
```

# Itérations

- LOOP-EXIT-WHEN-END

LOOP

....

EXIT WHEN CONDITION ou EXIT

....

END LOOP

- Exemple:

i:= 1 ;

LOOP

i:= i +1 ;

EXIT WHEN i >= 10 ;

....

END LOOP

# Itérations

- WHILE-LOOP-END

WHILE condition LOOP

....

END LOOP

- Exemple:

WHILE total <= 10000 LOOP

.....

SELECT salaire INTO S FROM employe WHERE.....

....

total:= total + S ;

END LOOP



# Itérations

- FOR-IN-LOOP-END

FOR compteur IN [ REVERSE ] inf..sup LOOP

.....

END LOOP

- Exemple:

FOR I IN 1..10 LOOP

J:= J\* 3 ;

INSERT INTO T VALUES ( I, J );

END LOOP

```
declare
  i int;
begin
  FOR i IN REVERSE 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE(i);
  END LOOP;
end;
```

# NULL

- Signifie «ne rien faire » , passer à l'instruction suivante
- Exemple:

```
IF I >= 10 THEN
```

```
    NULL ;
```

```
ELSE
```

```
    INSERT INTO T VALUES ( 'inférieur à 10', I );
```

```
END IF ;
```

# Curseurs

- Définition d'un curseur.
- Déclaration d'un curseur.
- Utilisation d'un curseur.

# Curseur - définition

- PL/SQL utilise des curseurs pour tous les accès à des informations de la base de données.
- Pointeurs associés au résultat d'une requête.
- Deux types de curseurs:
  - **Implicite**: créés automatiquement par Oracle pour chaque ordre SQL (**select, update, delete, insert**) .
  - **Explicite**: crée par le programmeur pour pouvoir traiter le résultat de requêtes retournant **plus** d'un tuple.

# Étapes de gestion d'un curseur

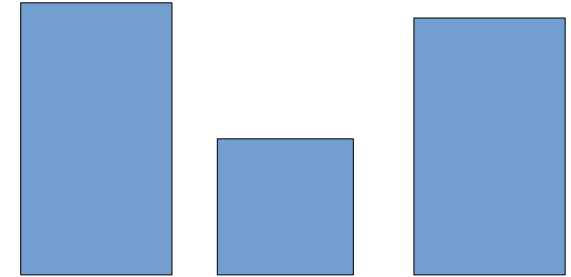
déclarative

\* 1

La déclaration du curseur  
lie le curseur et une requête SQL

CURSOR ...

Tables Oracle



exécution

\* 1

Ouverture du curseur  
Exécution de la requête SQL  
Chargement en mémoire +  
contexte

OPEN ...

\* n

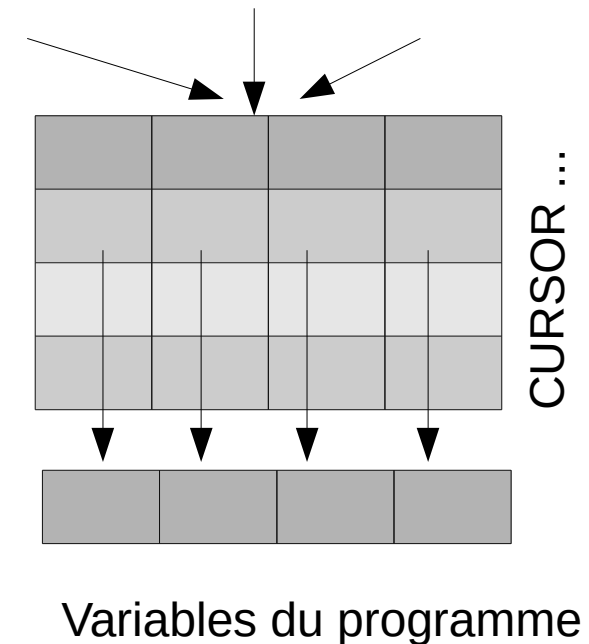
Accès séquentiel aux lignes  
via des variables  
Gestion d'une boucle

FETCH ...

\* 1

Fermeture du curseur  
Libération de la zone mémoire  
acquise

CLOSE ...



- Avec Oracle il n'est pas possible d'inclure un select sans « into » dans une procédure ; pour ramener des lignes, voir la suite du cours sur les curseurs
- Tous les curseurs ont des attributs que l'utilisateur peut utiliser
  - %ROWCOUNT : nombre de lignes traitées par le curseur
  - %FOUND : vrai si au moins une ligne a été traitée par la requête ou le dernier fetch
  - %NOTFOUND : vrai si aucune ligne n'a été traitée par la requête ou le dernier fetch
  - %ISOPEN : vrai si le curseur est ouvert (utile seulement pour les curseurs explicites)

# Curseur implicite

- Les curseurs implicites sont tous nommés SQL

**DECLARE**

nb\_lignes integer;

**BEGIN**

delete from emp

where dept = 10;

nb\_lignes := SQL%ROWCOUNT;

...

# Curseur explicite

- Pour traiter les select qui renvoient plusieurs lignes
- Ils doivent être déclarés
- Le code doit être utiliser explicitement avec les ordres **OPEN**, **FETCH** et **CLOSE**
- Le plus souvent on les utilise dans une boucle dont on sort quand l'attribut **NOTFOUND** du curseur est vrai



# Curseurs explicite - déclaration

- Déclaration du curseur sans parametres:

**DECLARE CURSOR** <nom\_curseur> **IS** Requete\_SELECT

- Exemple: Déclarer un curseur qui permet de lister les employés de rabat

**DECLARE**

v\_ville employe.ville%type := 'Rabat'

**CURSOR** employe\_rabat **IS**

**SELECT** numemp, nomemp **FROM** employe **WHERE** ville=  
v\_ville;

# Curseurs explicite- déclaration

- Déclaration du curseur **avec parametres:**

**DECLARE CURSOR** <nom\_curseur> ( para1, para2,.....)

**IS** Requete\_SELECT ;

- On doit fermer le curseur entre chaque utilisation de paramètres différents ( sauf si on utilise « for » qui ferme automatiquement le curseur )

- Exemple:

**DECLARE**

**CURSOR** employe\_rabat ( v **IN VARCHAR2**) **IS**

**SELECT** numemp, nomemp **FROM** employe **WHERE** ville= v;

# Curseurs explicite avec paramètres - Exemple

- **FOR LOOP** remplace **OPEN**, **FETCH** et **CLOSE**.
- Lorsque le curseur est invoqué, un enregistrement est automatiquement créé avec les mêmes éléments de données que ceux définies dans l'instruction select.

**declare**

```
CURSOR cur_employe( v_ville varchar2) IS
```

```
SELECT nomemp, sal FROM employe WHERE ville= v_ville ;
```

**begin**

```
FOR enrg_e IN cur_employe('rabat')
```

```
    /* traitement*/
```

```
END LOOP;
```

```
FOR enrg_e IN cur_employe('kenitra')
```

```
    /* traitement*/
```

```
END LOOP;
```

**end;**

# Curseurs explicite - déclaration

- Déclaration du curseur **avec clause RETURN:**

**DECLARE**

**CURSOR** <nom\_curseur> ( para1, para2,.....)

**RETURN** specification\_valeur\_retour

**IS** Requete\_SELECT ;

- Exemple:

**DECLARE**

**CURSOR** employe\_rabat ( v **IN VARCHAR2**)

**RETURN** employe%ROWTYPE

**IS**

**SELECT** numemp, nomemp **FROM** employe **WHERE** ville= v;

# Curseur - Ouverture

- L'ouverture du curseur se fait dans la section BEGIN du bloc par:

**OPEN** nom\_curseur

- Exemple:

**DECLARE**

**CURSOR** employe\_rabat **IS**

**SELECT** numemp, nomemp **FROM** employe **WHERE** ville= ' Rabat';

**BEGIN**

**OPEN** employe\_rabat ;

.....

.....

**END ;**

**Si aucune ligne n'est lue**

**⇒ OPEN ne déclenche pas d'exception**

**⇒ Le programme teste l'état du curseur après le FETCH**

# Curseur - accès aux données

- L'accès aux données se fait par la clause: **FETCH INTO**
- syntaxe:

**FETCH** nom\_curseur **INTO** < var1, var2,.....>

- Le **FETCH** permet de récupérer un tuple de l'ensemble réponse associé au curseur et stocker les valeurs dans des variables.
- Pour traiter n lignes, il faut une boucle.

# Curseurs - Exemple

```
CREATE TABLE resultat ( nom varchar2, sal number )
```

```
DECLARE
```

```
    CURSOR employe_rabat IS
```

```
    SELECT numemp, nomemp FROM employe WHERE ville= ' Rabat' ;
```

```
    nom employe.nomempl%TYPE ;
```

```
    salaire employe.sal %TYPE ;
```

```
BEGIN
```

```
    OPEN employe_rabat
```

```
    FETCH employe_rabat INTO nom, salaire ;
```

```
    WHILE employe_rabat%found LOOP
```

```
        IF salaire >3000 THEN
```

```
            INSERT INTO resultat VALUES ( nom, salaire) ;
```

```
        END IF ;
```

```
        FETCH employe_rabat INTO nom, salaire ;
```

```
    END LOOP;
```

```
    CLOSE employe_rabat ;
```

```
END ;
```

# Fermeture du curseur

- La fermeture du curseur se fait dans la section **BEGIN** du bloc par:

**CLOSE** nom\_curseur

- Exemple

**DECLARE**

**CURSOR** employe\_rabat **IS**

**SELECT** numemp, nomemp **FROM** employe **WHERE** ville= ' Rabat' ;

**BEGIN**

**OPEN** employe\_rabat

.....

**CLOSE** employe\_rabat ;

**END ;**



# Exemple

- Donner un programme PL/SQL utilisant un curseur pour calculer la somme des salaires des employés

**DECLARE**

**CURSOR** salaires **IS**

**select** sal **from** emp **where** dept = 10;

salaire **numeric**(8, 2);

total **numeric**(10, 2) := 0;

**BEGIN**

**open** salaires;

**loop**

**fetch** salaires **into** salaire;

**exit** **when** salaires%**notfound**;

**if** **salaire is not null** **then**

total := total + salaire;

**end if**;

**end loop**;

**close** salaires; -- Ne pas oublier

**DBMS\_OUTPUT.put\_line**(total);

**END**;

**Attention !**



# Type « row » associé à un curseur

- On peut déclarer un type « row » associé à un curseur
- Exemple :

**declare**

**cursor c is**

**select** matr, nome, sal **from** emp;

employe c%ROWTYPE;

**begin**

**open** c;

**fetch** c **into** employe;

**if** employe.sal **is null then** ...

# Boucle FOR pour un curseur

- Elle simplifie la programmation car elle évite d'utiliser explicitement les instruction open, fetch, close
- En plus elle déclare implicitement une variable de type «row» associée au curseur
- Exemple

declare

cursor c is select dept, nome from emp

where dept = 10;

begin

FOR employe IN c LOOP

dbms\_output.put\_line(employe.nome);

END LOOP;

end;

Variable de type  
c%rowtype



# Les attributs de curseur explicite

Attributs	Informations obtenues
<b>%ROWCOUNT</b>	Nombre de lignes concernées par la dernière requête
<b>%FOUND</b>	TRUE si la dernière requête concernait une ou plusieurs lignes
<b>%NOTFOUND</b>	TRUE si la dernière requête ne concernait aucune ligne
<b>%ISOPEN</b>	TRUE si le curseur est ouvert

Format d'utilisation

Nom\_curseur% ATTRIBUT

# Curseur modifiable

- Un curseur qui comprend plus d'une table dans sa définition ne permet pas la modification des tables de BD.
- **Seuls les curseurs définis sur une seule table sans fonction d'agrégation et de regroupement peuvent être utilisés dans une MAJ : delete, update.**
- **FOR UPDATE** [**OF** col1, col2,...]. Cette clause bloque toute la ligne ou seulement les colonnes spécifiées
- Les autres transactions ne pourront modifier les valeurs tant que le curseur n'aura pas quitté cette ligne
- Pour désigner la ligne courante à modifier on utilise la clause **CURRENT OF** nom\_curseur.

# Curseur modifiable - exemple

DECLARE

Cursor C1 is

select nomemp, sal from employe **for update of sal;**

resC1 C1%rowtype;

BEGIN

Open C1;

Fetch C1 into resC1;

While C1%found Loop

  If resC1.sal < 1500 then

    update employe

    set sal = sal \* 1.1

**where current of c1;**

  end if;

  Fetch C1 into resC1;

end loop;

close C1 ;

END; /

# Exceptions

- Une exception est une erreur qui survient durant une exécution
- Deux types d'exception :
  - prédéfinie par Oracle
  - définie par le programmeur
- Une exception ne provoque pas nécessairement l'arrêt du programme si elle est saisie par un bloc (dans la partie «**EXCEPTION**»)
- Une exception non saisie remonte dans la procédure appelante (où elle peut être saisie)
- Exceptions prédéfinies
  - **NO\_DATA\_FOUND**
  - **TOO\_MANY\_ROWS**
  - **VALUE\_ERROR** (erreur arithmétique)
  - **ZERO\_DIVIDE**

# Traitement des exceptions

**BEGIN**

...

**EXCEPTION**

**WHEN** exception1 [**OR** exception2 ...] **THEN**

...

**WHEN** exception3 [**OR** exception4 ...] **THEN**

...

[**WHEN OTHERS THEN** -- optionnel

...]

**END;**

- **WHEN** ..... Identifie une ou plusieurs exceptions
- **WHEN OTHERS** ..... Indique le traitement à effectuer si l'exception ne peut être appréhendée par une des clauses **WHEN**



# Exceptions: 2 types

- **Exceptions définies par ORACLE**
  - Nommées par oracle
    - Ex: `NO_DATA_FOUND`, `TOO_MANY_ROWS`,....
    - Se déclenchent automatiquement.
    - Nécessite de prévoir la prise en compte de l'erreur dans la section `EXCEPTION`.
- **Exceptions définies par l'utilisateur**
  - Nommées par l'utilisateur.
  - Arrêt de l'exécution du bloc.
  - Sont déclenchées par une instruction du programme soit automatiquement (`PRAGMA`).
  - Nécessite de prévoir la prise en compte de l'erreur dans la section `EXCEPTION`.

# Liste d'exceptions prédéfinies

Nom Exception	Numéro erreur	Description
NO_DATA_FOUND	ORA-01403	La SELECT into ne ramène pas de ligne
TOO_MANY_ROWS	ORA-01422	La SELECT into ramène plus d'une ligne
INVALID_CURSOR	ORA-01001	Opération sur un curseur invalide
ZERO_DIVIDE	ORA-01476	Tentative de division par 0
DUP_VAL_ON_INDEX	ORA-00001	Tentative d'insertion d'une valeur présente dans une colonne ayant un index unique

# Exceptions non prédéfinies - règles

## DECLARE

Définition

1/ Définir un nom pour l'exception

**Nom\_d'exception EXCEPTION;**

2/ Associer le nom de l'exception à un numéro d'erreur Oracle

**PRAGMA EXCEPTION\_INIT nom\_d'exception;**

## BEGIN

Déclenchement

Déclenchement automatique sous le contrôle du serveur Oracle

## EXCEPTION

Interception

Utiliser le mot clé identifiant l'erreur dans la structure de choix :

**WHEN nom\_d'exception THEN - - -;**

# Exceptions non prédéfinies - exemple

## DECLARE

s\_enrg\_fils EXCEPTION;

Définition d'un nom d'exception

PRAGMA EXCEPTION\_INIT

(s\_enrg\_fils ,-2292);

## BEGIN

Associe un nom d'erreur avec un numéro d'erreur ORACLE

---

## EXCEPTION

Prise en compte de l'erreur définie

WHEN s\_enrg\_fils THEN

DBMS\_OUTPUT.PUT\_LINE('Contrainte d'intégrité non respectée');

Gérer l'erreur -2292 qui correspond à la violation d'une contrainte d'intégrité référentielle.

END;/

# Exception définie par l'utilisateur: règles

## DECLARE

Définition

1/ Définir un nom pour l'exception

Nom\_d'exception EXCEPTION;

## BEGIN

Déclenchement

Déclenchement sous le contrôle du programme

RAISE Nom\_d'exception;

## EXCEPTION

Interception

Utiliser le mot clé identifiant l'erreur dans la structure de choix :

WHEN nom\_d'exception THEN - - -;

# Exception définie par l'utilisateur

**DECLARE**

.....

Nom\_exception **EXCEPTION**;

**BEGIN**

Instructions ;

**IF** ( condition\_erreur ) **THEN RAISE** Nom\_exception ;

.....

**EXCEPTION**

**WHEN** Nom\_exception **THEN** traitements ;

**END ;**

- Remarques:
  - on sort du bloc après l'exécution du traitement d'erreur.
  - Les règles de visibilité des exceptions sont les mêmes que celle des variables.

# Propagation des exceptions

- **Traitement de la référence à une exception:**
  - 1) Le traitement associé à l'exception est d'abord recherché dans le bloc courant. Si l'exception n'est pas trouvée, passer à l'étape 2:
  - 2) Si un bloc supérieur est trouvé, le traitement associé est recherché ensuite dans ce bloc
  - 3) Les étapes 1 et 2 sont répétées tant qu'il y a un bloc supérieur ou jusqu'à l'identification du traitement associé à l'exception.

# Exception - Exemple

DECLARE

CURSOR employe\_rabat IS

SELECT nomemp, sal FROM employe WHERE ville= ' Rabat' ;

nom employe.nomemp%TYPE ;

salaire employe.sal %TYPE ;

ERR\_salaire EXCEPTION ;

BEGIN

OPEN employe\_rabat;

FETCH employe\_rabat INTO nom, salaire ;

WHILE employe\_rabat%found LOOP

IF salaire IS NULL THEN

RAISE ERR\_salaire ;

.....

EXCEPTION

WHEN ERR\_salaire THEN

INSERT INTO temp ( nomempl || ' salaire non définie' ) ;

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Pas d'employe');

END ;



# Exception "others "

- Exception prédéfinie porte le nom de others.
- Permet de traiter toute autre exception non prévue.
- Syntaxe : **WHEN OTHERS THEN ...**
- Deux fonctions permettent de récupérer des informations sur l'erreur oracle:
  - **Sqlcode**: retourne une valeur numérique : numéro de l'erreur.
  - **Sqlerrm**: renvoie le libellé de l'erreur.

# Exemple

## DECLARE

```
salaire employe.sal %TYPE ;  
SAL_nulle EXCEPTION ;  
code number ;  
message char (50);
```

## BEGIN

```
SELECT sal INTO salaire from employe;  
IF salaire= 0 THEN
```

```
    RAISE SAL_nulle;
```

```
END IF
```

## EXCEPTION

```
WHEN SAL_nulle THEN
```

```
    -- gérer erreur salaire
```

```
WHEN TOO_MANY_ROWS THEN
```

```
    -- gérer erreur trop de lignes
```

```
WHEN NO_DATA_FOUND THEN
```

```
    -- gérer erreur pas de lignes
```

```
WHEN OTHERS THEN
```

```
    -- gérer toutes les autres erreurs
```

```
code:= sqlcode ;
```

```
message:= sqlerrm ;
```

```
dbms_output.put_line ( 'erreur: ' || code || message );  
END ;
```

# Types de blocs PL/SQL

```
[DECLARE]
```

```
BEGIN
```

```
-- phrases
```

```
[EXCEPTION]
```

```
END;
```

## Bloc anonyme

Imbriqué dans un programme

Emis interactivement (SQL\*PLUS)

```
PROCEDURE name IS  
[Déclaration_variables_locales]
```

```
BEGIN
```

```
-- phrases
```

```
[EXCEPTION]
```

```
END;
```

## Procédure

Effectue un traitement

Peut recevoir des paramètres

```
FUNCTION name IS  
RETURN datatype;  
[Déclaration_variables_locales]
```

```
BEGIN
```

```
-- phrases
```

```
[EXCEPTION]
```

```
END;
```

## Fonction

Effectue un calcul et donne un résultat

Peut recevoir des paramètres

# Procédures et fonctions

- Une procédure/fonction stockée est composée d'instructions compilées et enregistrées dans la BD. Elle est activée par des événements ou des applications. Elle comporte, outre des ordres SQL, des instructions de langage PL/SQL (branchement conditionnel, instructions de répétition, affectations,...).
- L'intérêt d'une procédure stockée est :
  - 1) D'alléger les échanges entre client et serveur de BD en stockant au niveau du serveur les procédures régulièrement utilisées.
  - 2) D'optimiser les requêtes au moment de la compilation des procédures plutôt qu'à l'exécution.
  - 3) De renforcer la sécurité : on peut donner l'autorisation { un utilisateur d'utiliser une procédure stockée sans lui donner les droits directement sur les tables qu'elle utilise.

# Procédures

- On définit une procédure de la sorte

```
CREATE [OR REPLACE] PROCEDURE nom_procedure  
    [ (liste_argument1 )] {IS|AS}
```

```
[ -- déclaration des variables, exceptions, procedures,...locales]
```

```
BEGIN
```

```
    -- Instructions PL/SQL
```

```
[exception
```

```
    -- gestion des exceptions
```

```
]
```

```
END [nom_procedure] ;
```

- Syntaxe de liste\_argument :

**nom\_argument [ IN|OUT ] type\_données [ { := | DEFAULT } valeur ]**

- **IN** : Paramètre d'entrée
- **OUT** : Paramètre de sortie
- **IN OUT** : Paramètre d'entrée/Sortie
- L'exécution de l'ordre **CREATE PROCEDURE ...** déclenche :
  - La compilation du code source avec génération de pseudo-code si aucune erreur n'est détectée.
  - Le stockage du code source dans la base même si une erreur a été détectée.
  - Le stockage du pseudo-code dans la base, ce qui évite la recompilation de la procédure à chaque appel de celle-ci.

# Procédures – Exemple 1

Exemple : procédure permettant d'afficher le nom de tous les employés d'un département dont le numéro est passé en paramètres.

```
CREATE OR REPLACE PROCEDURE Get_emp_names (Dept_num IN NUMBER) IS
```

```
Emp_name  VARCHAR2(10);
```

```
CURSOR    c1 (Depno NUMBER) IS
```

```
    SELECT Ename FROM Emp_tab
```

```
    WHERE deptno = Depno;
```

```
BEGIN
```

```
OPEN c1(Dept_num);
```

```
LOOP
```

```
    FETCH c1 INTO Emp_name;
```

```
    EXIT WHEN C1%NOTFOUND;
```

```
    DBMS_OUTPUT.PUT_LINE(Emp_name);
```

```
END LOOP;
```

```
CLOSE c1;
```

```
END;
```

# Procédures – exemple 2

- Exemple : Créer une procédure qui permet d'augmenter le prix d'un produit par un taux.

```
CREATE OR REPLACE PROCEDURE augmentation_prix ( numerop IN      NUMBER,  
Taux IN NUMBER ) IS
```

```
    non_trouve Excpetion;
```

```
BEGIN
```

```
    UPDATE produit SET PU= PU* ( 1+Taux)
```

```
    WHERE Numprod= numerop ;
```

```
    if ( SQL%NOTFOUND ) THEN
```

```
        RAISE non_trouve;
```

```
EXCEPTION
```

```
    WHEN non_trouve THEN
```

```
        dbms_output.put_line ( 'produit non trouvé ' );
```

```
END ;
```



# Procédures – Exemple 3

- Procédure Get\_emp\_rec, qui renvoie toutes les colonnes de la table Emp\_tab dans un enregistrement PL / SQL pour l'empno donné:

```
CREATE OR REPLACE PROCEDURE Get_emp_rec (Emp_number IN Emp_tab.Empno%TYPE, Emp_ret OUT  
Emp_tab%ROWTYPE) IS
```

```
BEGIN
```

```
    SELECT Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno
```

```
        INTO Emp_ret
```

```
        FROM Emp_tab
```

```
        WHERE Empno = Emp_number;
```

```
END;
```

```
DECLARE
```

```
    Emp_row    Emp_tab%ROWTYPE;  -- declare a record matching a
```

```
                                -- row in the Emp_tab table
```

```
BEGIN
```

```
    Get_emp_rec(7499, Emp_row); -- call for Emp_tab# 7499
```

```
    .....
```

```
END;
```

# Procédures – Exemple 4

- Exemple : Créer une procédure, ayant comme paramètre le numéro d'un employé, qui retourne les informations de l'employé passé en paramètre .

employe (numemp, nomemp, salaire, date\_embauche )

**CREATE PROCEDURE REQ\_EMPLOYE**

( numero IN employe.numemp%type,  
nom\_e OUT employe.nomemp%type,  
sal\_e OUT employe.salaire%type,  
date\_e OUT employe.date\_embauche%type)

**IS**

**BEGIN**

SELECT nomemp, salaire, date\_embauche  
into nom\_e, sal\_e, date\_e  
WHERE numemp = numero ;

**END ;**

# Optimisation des procédures

- Si la base de données évolue, il faut recompiler les procédures existantes pour qu'elles tiennent compte de ces modifications.
- La commande est la suivante:  
`ALTER { FUNCTION | PROCEDURE } nom COMPILE`
- Exemple :  
`ALTER PROCEDURE augmentation_prix COMPILE;`

# Suppression d'une procédure ( fonction)

- Pour supprimer une procédure

**DROP { FUNCTION |PROCEDURE } nom**

# Fonctions

- On définit une fonction de la sorte

```
CREATE [OR REPLACE] FUNCTION nom_fonction [(liste_argument)]  
{IS|AS} RETURN type_données {IS|AS}
```

```
    [déclaration_variables_locales]
```

```
BEGIN
```

```
    -- Instructions PL/SQL
```

```
[Gestions des exceptions]
```

```
END [nom_fonction] ;
```

# Fonctions - Exemple

Exemple : Créer une fonction qui retourne le nom d'un employé.

employe (numemp, nomemp, salaire, date\_embauche )

```
CREATE FUNCTION nom_employe ( numero IN NUMBER )
```

```
RETURN VARCHAR2
```

```
IS
```

```
nom employe. nomemp%type ;
```

```
BEGIN
```

```
SELECT nomemp into nom
```

```
FROM employe
```

```
WHERE numemp = numero ;
```

```
RETURN( nom);
```

```
END ;
```

# Exercice

- 1) Écrire une fonction qui permet de calculer le nombre d'employés dans un département . Cette fonction reçoit comme paramètre le code du département et retourne le nombre d'employés dans ce département.
- 2) Écrire une fonction qui permet d'insérer un enregistrement dans la table employés et qui retourne le numéro d'employé inséré. l'Insertion utilise une séquence.

# Exécution et suppression

- Sous SQL\*PLUS on exécute une procédure PL/SQL avec la commande EXECUTE :  
EXECUTE nomProcédure(param1, ...);

Une fonction peut être utilisée dans une requête SQL

## Exemple

```
select nome, sal, euro_to_fr(sal)
from emp;
```

- partir d'un bloc PL/SQL  
DECLARE V\_numP NUMBER := 14  
BEGIN  
 augmentation\_prix( V\_numP, 0.1);  
END ;

- Sous SQL PLUS:
  - 1) EXECUTE augmentation\_prix( 1, 0.2);
  - 2) VARIABLE nom varchar2(30)  
EXECUTE :nom= nom\_employe (2) ;  
PRINT nom;



# Type d'appel d'un sous-programme

- L'appel d'un sous-programme ( procédure ou fonction) peut être positionnel, mot clé (nommé) ou mixte.
  - Paramètres positionnels  
`augmentation_prix( 1, 0.2)`
  - Paramètres mot clé  
`augmentation_prix( Taux=>0.2, numerop=>1)`
  - Mixtes  
`augmentation_prix( 1, Taux=>0.2)`
- Attention: Pour tous les appels mixtes, il faut que les notations positionnelles précèdent les notations nommées.

# Exemple 1

```
DECLARE

TYPE Foursome IS VARRAY(4) OF VARCHAR2(15); -- VARRAY type
-- varray variable initialized with constructor:
team Foursome := Foursome('Khalid', 'Fatima', 'Aymane', 'Imane');

PROCEDURE print_team (heading VARCHAR2) IS

BEGIN
    DBMS_OUTPUT.PUT_LINE(heading);
    FOR i IN 1..4 LOOP
        DBMS_OUTPUT.PUT_LINE(i || '.' || team(i));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('---');
END;

BEGIN
    print_team('2001 Team:');
    team(3) := 'Latifa'; -- Change values of two elements
    team(4) := 'Farid';
    print_team('2005 Team:');
    -- Invoke constructor to assign new values to varray variable:
    team := Foursome('Redouane', 'Ilhame', 'Mohamed', 'Ihssane');
    print_team('2009 Team:');
END;

• /
```

Result:

```
2001 Team:
1.Khalid
2.Fatima
3.Aymane
4.Imane
---
2005 Team:
1.Khalid
2.Fatima
3.Latifa
4.Farid
---
2009 Team:
1.Redouane
2.Ilhame
3.Mohamed
4.Ihssane
---
```

# Exemple 2

- DECLARE
- TYPE Roster IS TABLE OF VARCHAR2(15); -- nested table type
- -- nested table variable initialized with constructor:
- names Roster := Roster('D Caruso', 'J Hamil', 'D Piro', 'R Singh');
- PROCEDURE print\_names (heading VARCHAR2) IS
- BEGIN
- DBMS\_OUTPUT.PUT\_LINE(heading);
- FOR i IN names.FIRST .. names.LAST LOOP -- For first to last element
- DBMS\_OUTPUT.PUT\_LINE(names(i));
- END LOOP;
- DBMS\_OUTPUT.PUT\_LINE('---');
- END;
- BEGIN
- print\_names('Initial Values:');
- names(3) := 'P Perez'; -- Change value of one element
- print\_names('Current Values:');
- names := Roster('A Jansen', 'B Gupta'); -- Change entire table
- print\_names('Current Values:');
- END;
- /

Result:

Initial Values:

D Caruso

J Hamil

D Piro

R Singh

---

Current Values:

D Caruso

J Hamil

P Perez

R Singh

---

Current Values:

A Jansen

B Gupta

# Exemple 3

- CREATE OR REPLACE TYPE nt\_type IS TABLE OF NUMBER;
- /
- CREATE OR REPLACE PROCEDURE print\_nt (nt nt\_type) IS
- i NUMBER;
- BEGIN
- i := nt.FIRST;
- IF i IS NULL THEN
- DBMS\_OUTPUT.PUT\_LINE('nt is empty');
- ELSE
- WHILE i IS NOT NULL LOOP
- DBMS\_OUTPUT.PUT('nt.( ' || i || ' ) = '); print\_nt(i);
- i := nt.NEXT(i);
- END LOOP;
- END IF;
- DBMS\_OUTPUT.PUT\_LINE('---');
- END print\_nt;
- /
- DECLARE
- nt nt\_type := nt\_type(); -- nested table variable initialized to empty
- BEGIN
- print\_nt(nt);
- nt := nt\_type(90, 9, 29, 58);
- print\_nt(nt);
- END;
- /

Result:

```
nt is empty
---
nt.(1) = 90
nt.(2) = 9
nt.(3) = 29
nt.(4) = 58
---
```

# Collections multidimensionnelles

- Bien qu'une collection ne comporte qu'une seule dimension,
- vous pouvez modéliser une collection multidimensionnelle avec une collection dont les éléments sont des collections.

# Exemple 4 Varray bidimensionnel (Varray de Varrays)

- DECLARE
- TYPE t1 IS VARRAY(10) OF INTEGER; -- varray of integer
- va t1 := t1(2,3,5);
- TYPE nt1 IS VARRAY(10) OF t1; -- varray of varray of integer
- nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);
- i INTEGER;
- va1 t1;
- BEGIN
- i := nva(2)(3);
- DBMS\_OUTPUT.PUT\_LINE('i = ' || i);
- nva.EXTEND;
- nva(5) := t1(56, 32); -- replace inner varray elements
- nva(4) := t1(45,43,67,43345); -- replace an inner integer element
- nva(4)(4) := 1; -- replace 43345 with 1
- nva(4).EXTEND; -- add element to 4th varray element
- nva(4)(5) := 89; -- store integer 89 there
- END;
- /

Result:

i = 73

# Exemple 5 : Tables de tables et de varrays de nombres entiers

- DECLARE
- TYPE tb1 IS TABLE OF VARCHAR2(20); -- nested table of strings
- vtb1 tb1 := tb1('one', 'three');
- TYPE ntb1 IS TABLE OF tb1; -- nested table of nested tables of strings
- vntb1 ntb1 := ntb1(vtb1);
- TYPE tv1 IS VARRAY(10) OF INTEGER; -- varray of integers
- TYPE ntb2 IS TABLE OF tv1; -- nested table of varrays of integers
- vntb2 ntb2 := ntb2(tv1(3,5), tv1(5,7,3));
- BEGIN
- vntb1.EXTEND;
- vntb1(2) := vntb1(1);
- vntb1.DELETE(1); -- delete first element of vntb1
- vntb1(2).DELETE(1); -- delete first string from second table in nested table
- END;
- /

# Exemple 6 Tables de tableaux associatifs et de varrays de chaînes

- DECLARE
- TYPE tb1 IS TABLE OF INTEGER INDEX BY PLS\_INTEGER; -- associative arrays
- v4 tb1;
- v5 tb1;
- TYPE ntb1 IS TABLE OF tb1 INDEX BY PLS\_INTEGER; -- nested table of
- v2 ntb1; -- associative arrays
- TYPE va1 IS VARRAY(10) OF VARCHAR2(20); -- varray of strings
- v1 va1 := va1('hello', 'world');
- TYPE ntb2 IS TABLE OF va1 INDEX BY PLS\_INTEGER; -- nested table of varrays
- v3 ntb2;
- BEGIN
- v4(1) := 34; -- populate associative array
- v4(2) := 46456;
- v4(456) := 343;
- v2(23) := v4; -- populate nested table of associative arrays
- v3(34) := va1(33, 456, 656, 343); -- populate nested table of varrays
- v2(35) := v5; -- assign empty associative array to v2(35)
- v2(35)(2) := 78;
- END;
- /



# Exemple 7 : Comparaison des variables Varray et Table à NULL

- DECLARE
- TYPE Foursome IS VARRAY(4) OF VARCHAR2(15); -- VARRAY type
- team Foursome; -- varray variable
- TYPE Roster IS TABLE OF VARCHAR2(15); -- nested table type
- names Roster := Roster('Adams', 'Patel'); -- nested table variable
- BEGIN
- IF team IS NULL THEN
- DBMS\_OUTPUT.PUT\_LINE('team IS NULL');
- ELSE
- DBMS\_OUTPUT.PUT\_LINE('team IS NOT NULL');
- END IF;
- IF names IS NOT NULL THEN
- DBMS\_OUTPUT.PUT\_LINE('names IS NOT NULL');
- ELSE
- DBMS\_OUTPUT.PUT\_LINE('names IS NULL');
- END IF;
- END;
- /

Result:

team IS NULL  
names IS NOT NULL

# Exemple 8 : Comparaison des tables pour l'égalité et l'inégalité

- DECLARE
- TYPE dnames\_tab IS TABLE OF VARCHAR2(30); -- element type is not record type
- dept\_names1 dnames\_tab := dnames\_tab('Shipping','Sales','Finance','Payroll');
- dept\_names2 dnames\_tab := dnames\_tab('Sales','Finance','Shipping','Payroll');
- dept\_names3 dnames\_tab := dnames\_tab('Sales','Finance','Payroll');
- BEGIN
- IF dept\_names1 = dept\_names2 THEN
- DBMS\_OUTPUT.PUT\_LINE('dept\_names1 = dept\_names2');
- END IF;
- IF dept\_names2 != dept\_names3 THEN
- DBMS\_OUTPUT.PUT\_LINE('dept\_names2 != dept\_names3');
- END IF;
- END;
- /

Result:

dept\_names1 = dept\_names2  
dept\_names2 != dept\_names3

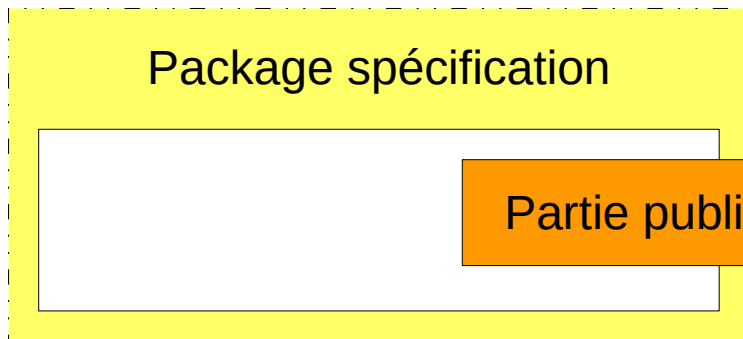
# Les packages

- Un package (paquetage) est **l'encapsulation** d'objets de programmation PL/SQL dans une même unité logique de traitement tels : types, constantes, variables, procédures et fonctions, curseurs, exceptions.
- La création d'un package se fait en deux étapes:
  - Création des **spécifications** du package
    - spécifier à la fois les fonctions et procédures **publiques** ainsi que les déclarations des types, variables, constantes, exceptions et curseurs utilisés dans le **paquetage et visibles par le programme appelant**.
  - Création du **corps** du package
    - définit les procédures (fonctions), les curseurs et les exceptions qui sont déclarés dans les spécifications de la procédure. Cette partie peut définir d'autres objets de même type non déclarés dans les spécifications. Ces objets sont alors **privés**.

Cette partie peut également contenir du code qui sera exécuté à chaque invocation du paquetage par l'utilisateur ( bloc d'initialisation).

# Structure d'un package

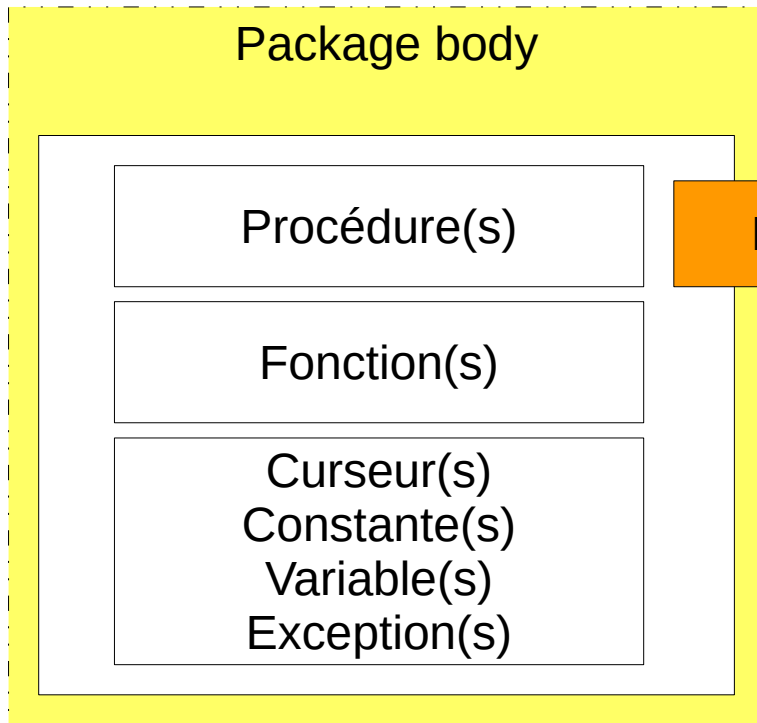
Un package est constitué de 2 composants



Partie public

- Déclaration des éléments publics du package.
- Informations accessibles à tout utilisateur autorisé.

**CREATE ou REPLACE ou DROP PACKAGE**



Partie privée

- Définition des éléments identifiés dans la partie « package spécification » .
- Définition de sous programmes (privés) utilisables seulement dans le package.

**CREATE ou REPLACE ou DROP PACKAGE BODY**

# Création d'une spécification du paquet

La syntaxe pour créer une spécification de paquet PL / SQL:

**CREATE [OR REPLACE] PACKAGE package\_name**

**[ AUTHID { CURRENT\_USER | DEFINER } ]**

**{ IS | AS }**

[definitions of public TYPES

,declarations of public variables, types, and objects

,declarations of exceptions

,pragmas

Public

,declarations of cursors, procedures, and functions

,headers of procedures and functions]

**END [package\_name];**

## Création du corps du package

- Syntaxe: CREATE [ OR REPLACE ] PACKAGE BODY nom\_package { IS| AS } specification PL/SQL  
specification PL/SQL ::= declaration\_de\_variable|  
declaration\_d\_enregistrement|  
declaration\_de curseur|  
declaration\_d\_exception|  
declaration\_de\_procedure|  
declaration\_de\_fonction...
- L'identificateur de la spécification et du corps du package doivent être les mêmes.

# Exemple 1

```
CREATE OR REPLACE PACKAGE gest_empl IS
```

```
-- Variables globales et publiques
```

```
Sal EMP.sal%Type ;
```

```
-- Fonctions publiques
```

```
FUNCTION F_Augmentation (Numemp IN EMPLOYEE.noemp%Type  
                        ,Pourcent IN NUMBER ) Return NUMBER ;
```

```
-- Procédures publiques
```

```
PROCEDURE Test_Augmentation (
```

```
    Numemp IN EMPLOYEE.noemp%Type --numéro del'employé
```

```
    ,Pourcent IN OUT NUMBER  -- pourcentage d'augmentation
```

```
);
```

- End gest\_empl;

## Exemple 1 (suite 1)

```
CREATE OR REPLACE PACKAGE BODY gest_empl IS
  --Variables globales privées
  Emp EMPLOYE%Rowtype ;
  -- Procédure privées
  PROCEDURE Affiche_Salaires IS
    CURSOR C_EMP IS select * from EMPLOYE ;
  BEGIN
    OPEN C_EMP ;
    Loop
      FETCH C_EMP Into Emp ;
      Exit when C_EMP%NOTFOUND ;
      dbms_output.put_line( 'Employé ' || Employee.name || ' -> ' |
                           To_char( Employee.sal );
    End loop ;
    CLOSE C_EMP ;
  END Affiche_Salaires ;
```



## Exemple 1 (suite 2)

-- Fonctions publiques

```
FUNCTION F_Augmentation ( Numemp IN EMPLOYEE.noemp%Type , Pourcent IN NUMBER ) Return NUMBER IS
    Salaire EMPLOYEE.sal%Type ;
BEGIN
    Select sal Into Salaire From EMPLOYEE
        Where noemp = Numemp ; -- augmentation virtuelle de l'employé
    Salaire := Salaire * Pourcent ;-- Affectation de la variable globale publique
    Sal := Salaire ;
    Return( Salaire ) ; -- retour de la valeur
END F_Augmentation;
```

-- Procédures publiques

```
PROCEDURE Test_Augmentation ( Numemp IN EMPLOYEE.noemp%Type ,Pourcent IN OUT NUMBER ) IS
    Salaire EMPLOYEE.sal%Type ;
BEGIN
    Select sal Into Salaire From EMPLOYEE Where noemp = Numemp ;
    Pourcent := Salaire * Pourcent ; -- augmentation virtuelle de l'employé
    Affiche_Salaires ;-- appel procédure privée
END Test_Augmentation;

END gest_empl; /
```

# Exemple

- **Spécification**

```
CREATE OR REPLACE PACKAGE compteur IS
```

```
    procedure reset;
```

```
    function nextValue return number;
```

```
END compteur; /
```

- **Corps**

```
CREATE OR REPLACE PACKAGE BODY compteur IS
```

```
    cpt NUMBER := 0;
```

```
    PROCEDURE reset IS
```

```
    BEGIN
```

```
        cpt := 0;
```

```
    END;
```

```
    FUNCTION nextValue RETURN NUMBER IS
```

```
    BEGIN
```

```
        cpt := cpt + 1;
```

```
        RETURN cpt - 1;
```

```
    END;
```

```
END compteur; /
```

- **L'accès à un objet d'un paquetage**

- L'accès à un objet d'un paquetage est réalisé avec la syntaxe suivante :  
`nom_paquetage.nom_objet[(liste paramètres)]`
- Exemple: Appel de la fonction F\_Augmentation du paquetage

Declare

`Salaire emp.sal%Type ;`

Begin

```
Select sal Into Salaire From EMPLOYE Where noempno = 100 ;  
dbms_output.put_line( 'Salaire de 100 avant augmentation ' ||  
To_char( Salaire ) ) ;  
  
dbms_output.put_line( 'Salaire de 100 après augmentation ' ||  
To_char(gest_empl.F_Augmentation( 100, 0.2 ) ) ) ;
```

End ; /

# Exemple d'utilisation d'un package

**DECLARE**

nb NUMBER;

**BEGIN**

FOR nb IN 4..20 LOOP

DBMS\_OUTPUT.PUT\_LINE(COMPTEUR.nextValue())

END LOOP;

COMPTEUR.RESET();

FOR nb IN REVERSE 0..10 LOOP

DBMS\_OUTPUT.PUT\_LINE(COMPTEUR.nextValue());

END LOOP;

**END; /**

# Déclencheur (trigger)

- Un trigger est un morceau de code PL/SQL **associé à une vue ou une table** :
  - stocké dans la base,
  - déclenché lors de l'occurrence d'un **événement particulier de langage de manipulation de données (DML)**

- **Syntaxe :**

```
CREATE [or REPLACE] TRIGGER <nom>
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
ON <table>
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
[WHEN (...)]
-- bloc PL/SQL
```

- **Avec**

- listeEvénements : liste d'événements séparés par une virgule **DELETE, INSERT, ou UPDATE**
- Si **UPDATE** on peut préciser les attributs concernés (**UPDATE OF** listeAttributs).

# Déclencheur (trigger)

- Ainsi les Triggers :
  - permettent de synchroniser des opérations entre plusieurs tables
  - peuvent être utilisés pour implémenter certaines règles de gestion (souvent les contraintes remplissent plus efficacement ce rôle)
  - sont généralement déclenchés par la modification du contenu d'une table
  - les ordres du LDD (CREATE, ALTER, DROP, ...) et de gestion de transactions (COMMIT, SAVEPOINT,...) sont interdits dans les Triggers.

# Les types de triggers

- Il existe deux types de triggers différents :
  - les triggers de table (STATEMENT)
  - les triggers de ligne (ROW).
- Les triggers de table sont exécutés **une seule fois** lorsque des modifications surviennent sur une table (même si ces modifications concernent plusieurs lignes de la table). Ils sont utiles si des opérations de groupe doivent être réalisées (comme le calcul d'une moyenne, d'une somme totale, d'un compteur, ...).
- Les triggers lignes sont exécutés **«séparément» pour chaque ligne** modifiée dans la table. Ils sont très utiles s'il faut mesurer une évolution pour certaines valeurs, effectuer des opérations pour chaque ligne en question.

# Les types de triggers

- Le traitement spécifié dans un trigger peut se faire :
  - pour chaque ligne concernée par l'événement => trigger de niveau ligne (**FOR EACH ROW**)
- une seule fois pour l'ensemble des lignes concernées par l'événement :  
=> trigger de niveau table : **pas de clause FOR EACH ROW**
- Quand le Trigger est déclenché ? **BEFORE | AFTER | INSTEAD OF**
  - trigger de niveau table : déclenché avant ou après l'événement
  - trigger de niveau ligne : exécuté avant ou après la modification de CHAQUE ligne concernée
  - INSTEAD OF : est utilisée pour créer un trigger sur une vue.



# Accès aux lignes en cours de modification

- Dans les **FOR EACH ROW** triggers, il est possible avant la modification de chaque ligne, de lire l'ancienne ligne et la nouvelle ligne par l'intermédiaire des deux variables structurées old et new.
- exemple :
  - Si nous ajoutons un client dont le nom est toto alors nous récupérons ce nom grâce à la variable **:new.nom**
  - Dans le cas de suppression ou modification, les anciennes valeurs sont dans la variable **:old.nom**

# L'option REFERENCING

- Seule la ligne en cours de modification est accessible par l'intermédiaire de 2 variables de type enregistrement **OLD** et **NEW**
- Les noms de ces deux variables sont fixés par défaut, mais il est possible de les modifier en précisant les nouveaux noms dans la clause **REFERENCING**

```
CREATE [or REPLACE] TRIGGER <nom>
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
ON <table>
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
[WHEN (...)]
-- bloc PL/SQL
```

# Conditions d'un triggers

- Le trigger se déclenche lorsqu'un événement précis survient : BEFORE UPDATE, AFTER DELETE, AFTER INSERT, .... Ces événements sont importants car ils définissent le moment d'exécution du trigger.
- Pour rappel, lorsqu'un trigger ligne est mentionné à
  - INSERT Pas d'accès à l'élément OLD (qui n'existe pas)
  - UPDATE Accès possible à l'élément OLD et NEW
  - DELETE Pas d'accès à l'élément NEW (qui n'existe plus)

# Triggers - Modification

- On ne peut modifier la définition du trigger, il faut la remplacer (d'où l'intérêt du CREATE OR REPLACE).
- Quand on crée le trigger, il est automatiquement activé. On peut le désactiver, puis le réactiver :

**ALTER TRIGGER nom\_trigger {disable|enable|compile}** pour désactiver/réactiver/recompiler

**ALTER TABLE nom\_table DISABLE ALL TRIGGERS** pour désactiver tous les triggers d'une table

**DROP TRIGGER nom\_trigger** pour la suppression d'un trigger

## Exemple 1

```
CREATE OR REPLACE TRIGGER TRG_BDR_EMP -- BeforDeleteRecord
    BEFORE DELETE -- avant suppression
    ON EMP      -- sur la table EMP
    FOR EACH ROW -- pour chaque ligne
Declare
    LC$Chaine VARCHAR2(100);
Begin
    dbms_output.put_line( 'Suppression de l''employé n° ' ||
        To_char( :OLD.empno ) || ' -> ' || :OLD.ename ) ;
End ;
```

# La procédure `raise_application_error`

- La procédure `raise_application_error` (`error_number,error_message`)
  - **`error_number`** doit être un entier compris entre -20000 et -20999
  - **`error_message`** doit être une chaîne de 500 caractères maximum.
  - Quand cette procédure est appelée, elle termine le trigger, défait la transaction (ROLLBACK), renvoie un numéro d'erreur défini par l'utilisateur et un message à l'application.

# Gestion des exceptions

- Si une erreur se produit pendant l'exécution d'un trigger, toutes les mises à jour produites par le trigger ainsi que par l'instruction qui l'a déclenché sont défaites.
- On peut introduire des exceptions en provoquant des erreurs.
  - Une exception est une erreur générée dans une procédure PL/SQL.
  - Elle peut être prédéfinie ou définie par l'utilisateur.
  - Un bloc PL/SQL peut contenir un bloc EXCEPTION gérant les différentes erreurs possibles avec des clauses WHEN.
  - Une clause WHEN OTHERS THEN ROLLBACK; gère le cas des erreurs non prévues.

## Exemple 2

- La DRH annonce que désormais, tout nouvel employé devra avoir un numéro supérieur ou égal à 10000 Il faut donc interdire toute insertion qui ne reflète pas cette nouvelle directive

```
CREATE OR REPLACE TRIGGER TRG_BIR_EMP
```

```
    BEFORE INSERT -- avant insertion
```

```
    ON EMP      -- sur la table EMP
```

```
    FOR EACH ROW -- pour chaque ligne
```

```
Begin
```

```
    If :NEW.empno < 10000 Then
```

```
        RAISE_APPLICATION_ERROR ( -20010, 'Numéro employé inférieur à 10000' );
```

```
    End if;
```

```
End;
```



# Les prédicats conditionnels **INSERTING**, **DELETING** et **UPDATING**

- Quand un trigger comporte plusieurs instructions de déclenchement (par exemple **INSERT OR DELETE OR UPDATE**), on peut utiliser des prédicats conditionnels
  - **INSERTING**,
  - **DELETING** et
  - **UPDATING**
- Pour exécuter des blocs de code spécifiques pour chaque instruction de déclenchement.

# Les prédicats conditionnels INSERTING, DELETING et UPDATING

*Exemple :*

*CREATE TRIGGER ...*

*BEFORE INSERT OR UPDATE ON employe*

*.....*

*BEGIN*

*.....*

*IF INSERTING THEN ..... END IF;*

*IF UPDATING THEN ..... END IF;*

*.....*

*END;*

# Les prédicats conditionnels INSERTING, DELETING et UPDATING

**UPDATING** peut être suivi d'un nom de colonne :

*CREATE TRIGGER ...*

*BEFORE UPDATE OF salaire, commission ON employe*

*.....*

*BEGIN*

*.....*

*IF UPDATING ('salaire') THEN ..... END IF;*

*.....*

*END;*

## Exemple 3

```
CREATE OR REPLACE TRIGGER TRG_BIUDR_EMP
  BEFORE INSERT OR UPDATE OR DELETE-- avant insertion, modification ou suppression
  ON EMP    -- sur la table EMP
  FOR EACH ROW -- pour chaque ligne
Begin
  If INSERTING Then
    dbms_output.put_line( 'Insertion dans la table EMP' );
  End if ;
  If UPDATING Then
    dbms_output.put_line( 'Mise à jour de la table EMP' );
  End if ;
  If DELETING Then
    dbms_output.put_line( 'Suppression dans la table EMP' );
  End if ;
End ;
```

# Quelques exemples

```
CREATE TABLE Livre (  
    noLivre NUMERIC PRIMARY KEY,  
    prix NUMERIC(9,2)  
)  
CREATE TABLE PrixLivre (  
    nb NUMERIC,  
    somme NUMERIC(12,2)  
)  
INSERT INTO PrixLivre VALUES (1,0);
```

- Créer un Trigger qui permet de vérifier avant chaque insertion d'un livre son prix :
- Si le prix est supérieur à 0
  - S'il est compris entre le prix moyen \* 0.7 et le prix moyen \* 1.3 , mettre à jour la table PrixLivre (somme et nb)
  - Si le prix est  $> \text{prix\_moyen} * 0.7$  et  $< \text{prix\_moyen} * 1.3$  soulever une exception
- Sinon mettre à jour uniquement le champ somme de la table PrixLivre

# Quelques exemples (suite)

```
CREATE OR REPLACE TRIGGER moyenne_prix
  BEFORE INSERT OF Prix
  ON Livre
  FOR EACH ROW
DECLARE
  prix_moyen number;
BEGIN
  SELECT SOMME/NB INTO prix_moyen
  FROM PrixLivre;
  IF (prix_moyen>0) THEN
    IF (:new.PRIX < prix_moyen*0.7 OR :new.PRIX > prix_moyen*1.3) THEN
      raise_application_error(-20001,'Prix modifiant trop la moyenne');
    END IF;
    IF (:new.PRIX > prix_moyen*0.7 AND :new.PRIX < prix_moyen*1.3)
    THEN UPDATE PrixLivre SET NB=NB+1,
      SOMME=SOMME+:new.PRIX;
    END IF;
  ELSE
    UPDATE PrixLivre SET SOMME=SOMME+:new.PRIX;
  END IF;
END;
```

# Quelques exemples (suite)

Soit une table quelconque TABL, dont la clé primaire CLENUM est numérique.

- **Définir un trigger en insertion** permettant d'implémenter une numérotation automatique de la clé. Le premier numéro doit être 1.

# Quelques exemples (suite)

```
create or replace trigger cleauto
  before insert on tabl
  for each row
declare
  n integer;
  newkey integer;
  mon_exception exception;
begin
  -- Recherche s'il existe des tuples dans la table
  select count(*) into n from tabl;
  if n=0 then
    raise mon_exception; -- Premiere insertion
  end if;
  -- Recherche la valeur de cle C la plus elevee
  -- et affecte C+1 a la nouvelle cle
  select max(clenum) into newkey from tabl;
  :new.clenum := newkey + 1;
exception
  -- Premier numero = 1
  when mon_exception then :new.clenum := 1;
end;
```



# Quelques exemples (suite)

- CLIENT (NUMCL, NOM, PRENOM, ADR, CP, VILLE, SALAIRE, CONJOINT)  
DETENTEUR (NUMCL, NUMCNP)  
COMPTE (NUMCNP, DATEOUPV, SOLDE)
- **Écrire un trigger en insertion permettant de contrôler les contraintes suivantes :**
  - le département dans lequel habite le client doit être 01, 07, 26, 38, 42, 69, 73, ou 74 (sinon il n'est pas en France\*)
  - le nom du conjoint doit être le même que celui du client.

# Quelques exemples (suite 1/3)

```
CREATE TRIGGER INS_CLIENT
  BEFORE INSERT ON CLIENT
  FOR EACH ROW
DECLARE
  nom_conjoint CLIENT.NOM%TYPE ;
  compteur CLIENT.NUMCL%TYPE ;
  pb_dept EXCEPTION ;
  pb_conjoint1 EXCEPTION ;
  pb_conjoint2 EXCEPTION ;
BEGIN
  -- Contrainte sur le département
  IF (:NEW.CP) NOT IN (01, 07, 26, 38, 42, 69, 73, 74) THEN
    RAISE pb_dept ;
  END IF ;
```

## Quelques exemples (suite 2/3)

```
-- Contrainte sur le nom du conjoint (+ test d'existence du conjoint)
IF :NEW.CONJOINT IS NOT NULL THEN
    SELECT COUNT(*), NOM INTO compteur, nom_conjoint
    FROM CLIENT
    WHERE NUMCL = :NEW.CONJOINT
    GROUP BY NOM ;
IF compteur = 0 THEN -- Pas de conjoint
    RAISE pb_conjoint1 ;
END IF ;
IF nom_conjoint != :NEW.NOM THEN
    RAISE pb_conjoint2 ;
END IF ;
END IF ;
```

# Quelques exemples (suite 3/3)

EXCEPTION

```
    WHEN pb_dept THEN  
        RAISE_APPLICATION_ERROR (-20501, 'Insertion impossible : le  
client n'habite pas en France!') ;  
    WHEN pb_conjoint1 THEN  
        RAISE_APPLICATION_ERROR (-20502, 'Insertion impossible : le  
conjoint du client n'existe pas !') ;  
    WHEN pb_conjoint2 THEN RAISE_  
        APPLICATION_ERROR (-20503, 'Insertion impossible : le nom du  
conjoint est différent de celui du client !') ;  
END ;
```

# Ch. IV Pages web pilotées par une base de données

- Ouverture d'une connexion vers une base de données
- Optimisation des requêtes (Partitionnement des tables et indexes, indexation des opérations en ligne et en parallèle, Maintenance et configuration des indexes).
- Stockage de données capturées par des formulaires
- Envoi de requêtes dynamiques à une base de données
- Générer une page web affichant les résultats d'une requête

- Un site dynamique est construit sur une base de données.
  - l'utilisateur sollicite une page, la demande est envoyée au site, le serveur exécute un programme,
  - les informations utiles sont extraites de la base de données et rapatriées au serveur,
  - une feuille de style est appliquée, un document html est construit et envoyé sur le réseau,
  - le document est affiché dans le navigateur de l'utilisateur.
- l'intérêt est de **distinguer les données des traitements et de l'affichage** : pour modifier le contenu du site, il suffit de **mettre à jour les informations** dans la base de données ; pour modifier la présentation, il suffit de **modifier le style** du document Web.

# Ouverture d'une connexion vers une base de données

\*db.properties

```
1 jdbc.driver=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/DB_CONTACTS
3 jdbc.user=root
4 jdbc.password=
```

```
try {
ClassLoader classLoader =
Thread.currentThread().getContextClassLoader();
InputStream in = classLoader.getResourceAsStream("db.properties");
prop.load(in);
in.close();
// Extraction des propriétés
String driver = prop.getProperty("jdbc.driver");
String url = prop.getProperty("jdbc.url");
String user = prop.getProperty("jdbc.user");
String password = prop.getProperty("jdbc.password");
Class.forName(driver);
Connection connect = DriverManager.getConnection(url, user,
password);

} catch (Exception e) {
e.printStackTrace();
}
```

# Ouverture d'une connexion vers une base de données

```
<?php
$link = mysql_connect('hostname','dbuser','dbpassword');
if (!$link) {
    die('Could not connect to MySQL: ' . mysql_error());
}

echo 'Connection OK';
mysql_close($link);
?>
```



# Stockage de données capturées par des formulaires

```
<?php
    $do_this = $_POST["do_this"];
    if(strcmp ($do_this,"Ajouter la commande")==0)
    {
        $ncom = $_POST["ncom"];
        $ncli = $_POST["ncli"];
        $datecom=$_POST["datecom"];
        $adrliv=$_POST["adrliv"];
        if(($ncom=intval($ncom))==false)
        {
            $erreur="Le numéro de la commande est mal saisi";
            require "erreur.php";
        }else if(($ncli=intval($ncli))==false)
        {
            $erreur="Le numéro de client est mal saisi";
            require "erreur.php";
        } else
```

# Stockage de données capturées par des formulaires

```
if($time = date($datecom)==false)
{
    $erreur="La date est mal saisi";
    require "erreur.php";
}
else
{
    // traitement de la requête
    if($db=new PDO('sqlite:commande.db'))
    {
        $sql="INSERT INTO commande(NCOM,NCLI, DATECOM,ADRLIV) VALUES($ncom,$ncli,
        $datecom,$adrliv)";
        $infG=$db->exec($sql);
        $db=NULL;
    }
    else
    {
        die ($sqliteerror);
    }
}
```

# Exemple authentication

```
<?php
// auth.php - Authentification des admins Bases Hacking
$login = $_POST["pseudo"];
$mdp = $_POST["mdp"];
echo($login.' '.$mdp);
if ($login != "" && $mdp != "") {
    echo("Authentification..");
    @mysql_connect("localhost", "root", "") or die("Impossible de se connecter à la base
de données");
    @mysql_select_db("users") or die("Table inexistante");
    $resultat = mysql_numrows(mysql_query("SELECT * from admin WHERE login='$login'
AND mdp='$mdp'"));
    mysql_close();
    if ($resultat >= 1){
        echo("Authentification réussie, vous allez être redirigés immédiatement.");
        header("location:./admin.php");
    }
    else header("Location: ./erreur.php");
} else header("Location: ./erreur1.php");
?>
```

# Générer une page web affichant les résultats d'une requête

```
<?php
    $ncom = $_POST["ncom"];
    $row = array();
    if ($ncom != "") {
        if(is_numeric($ncom)){
            $ncom=intval($ncom);
        }else{
            $erreur="Le numéro de la commande est mal saisi";
            require "erreur.php";
            goto fin;
        }
        if($db=new PDO('sqlite:commande.db'))
        {
            $sql1="select C.NCLI ,C.NOM, CM.ADRLIV from COMMANDE CM, CLIENT C
                    where C.NCLI=CM.NCLI and CM.NCOM=$ncom";
            $infG=$db->query($sql1);
            if(count ($infG)==0)
            {
                $erreur="le numéro entré ne correspond à aucune commande!";
                $db=NULL;
                require 'erreur.php';
                goto fin;
            }
        }
        else{
```

# Générer une page web affichant les résultats d'une requête

```
Montant
    $sql="select D.NPRO, P.LIBELLE, P.Prix as 'Prix Unit', D.QCOM, (D.QCOM*P.Prix) as
    from COMMANDE CM
    JOIN DETAIL D on CM.NCOM = D.NCOM
    JOIN PRODUIT P on D.NPRO = P.NPRO
    Where CM.NCOM=$ncom";
    $resultat=$db->query($sql);
    $db=NULL;
    $row=$infG->fetch(PDO::FETCH_NUM);
    require 'Facturesucces.php';
    goto fin;
}
}
else{
    die ($sqliteerror);
}
} else {
    $erreur="Veuillez entrer le numéro de commande!";
    require 'erreur.php';
}
fin:
?>
```

# Références

- <http://odile.papini.perso.esil.univmed.fr/sources/BD.html>
- <http://combot.univ-tln.fr/loris/admin/plsql/hash.html>
- <https://www.db.bme.hu/files/Manuals/Oracle/Oracle11gR2/appdev.112/e17126/composites.htm#CHDEIJHD>
-