

# SQL : le Langage de Définition des Données LDD

HLIN511

**Pascal Poncelet**

Pascal.Poncelet@umontpellier.fr

<http://www.lirmm.fr/~poncelet>



# Introduction

---

- Le **L**angage de **D**éfinition des **D**onnées permet de gérer la définition d'une base de données et de tous les éléments qui la compose
- Il permet par exemple de :
  - Créer une relation, de modifier une relation ou de supprimer une relation
- Toutes les opérations de mise à jour du schéma et de son interrogation se font via le LDD



# Création de la base

---

- Un schéma SQL (une base) est identifié par un nom de schéma
- **CREATE {DATABASE | SCHEMA} [IF NOT EXISTS]**  
Nom\_Base [*specification ...*]
- Les termes **DATABASE** ou **SCHEMA** peuvent être utilisés

**CREATE DATABASE PILOTE\_AVION\_VOL;**



# Création de relations

---

- **CREATE TABLE** a pour effet de créer une nouvelle relation, initialement vide, dans la base de données courante
- Il est possible de préciser un nom de schéma  
**CREATE TABLE MONSCHEMA.MATABLE;**
- La table est alors créée dans le schéma « MONSCHEMA »
- La table créée appartient à l'utilisateur qui l'a créé



# Création de relations

---

- Chaque relation/table possède un nom unique dans la base
- Une table est composée d'une ou plusieurs colonnes (attributs)
- Chaque colonne possède un nom unique à l'intérieur de la table
- Possibilité de spécifier des contraintes sur les valeurs d'une colonne (**NOT NULL, UNIQUE, CHECK, ...**)
- Les clauses de contraintes spécifiées dans la table imposent que les nouveaux tuples insérés ou les tuples mis à jour doivent vérifier les contraintes



# Les types de données de base

---

Alphanumérique	<b>CHAR(n)</b>	Chaîne de longueur n
Alphanumérique	<b>VARCHAR(n)</b>	Chaîne de n caractères max
Numérique	<b>NUMERIC (n,[d])</b>	Nombre de n chiffres avec en option le nombre de décimales
Numérique	<b>INTEGER</b>	Entier signé
Numérique	<b>SMALLINT</b>	Entier signé petit
Numérique	<b>FLOAT</b>	Nombre à virgule flottante
Temps	<b>DATE</b>	Date
Temps	<b>TIMESTAMP</b>	Date et Heure



# Création de relations

---

```
CREATE TABLE NOM_DE_LA_TABLE (  
    <NOM COL1> <TYPE COL1> <CONTRAINTE COL1>,  
    <NOM COL2> <TYPE COL2> <CONTRAINTE COL2>,  
    ...  
    <NOM COLn> <TYPE COLn> <CONTRAINTE COLn>  
);
```



# Saisie des dates

---

```
SQL> INSERT INTO R VALUES ('14-02-2016');
```

```
*ERROR at line 1:ORA-01843: not a valid month
```

Solution :

```
SQL> ALTER session SET NLS_DATE_FORMAT='DD-MM-YYYY' ;
```

Session altered.

```
SQL> INSERT INTO R VALUES ('14-02-2016');
```

```
1 row created.
```

Remarque : Attention toutes les dates devront avoir ce format





# Saisie des dates

---

Une autre saisie de date par nom de mois

```
SQL> INSERT INTO R VALUES ('14-FEV-2016');
```

```
*ERROR at line 1:ORA-01843: not a valid month
```

Problème plus important. Il en est de même pour les accents

```
SQL> INSERT INTO UNERELATION VALUES (100, 'éphémère');
```

```
SQL> SELECT * FROM UNERELATION WHERE ID=100;
```

ID	NAME
----	------

-----	
-------	--

100	?ph?m?re
-----	----------



# Saisie des dates

Solution : avant de lancer SQL\*Plus

Windows	LINUX/UNIX
set NLS_LANG=.AL32UTF8 set NLS_LANG=.UTF8	export NLS_LANG=.AL32UTF8 set NLS_LANG=.UTF8

Puis au lancement de SQL\*Plus

```
SQL> INSERT INTO R VALUES ('14-FEV-2016');
```

1 row created.

```
SQL>INSERT INTO UNERELATION VALUES (101, 'éphémère');
```

1 row created.

```
SQL> SELECT * FROM UNERELATION WHERE ID=101;
```

→

ID	NAME
101	éphémère

Remarques : Attention toutes les dates devront avoir ce format. Pour les accents les anciennes saisies doivent être réécrites :

```
SQL> SELECT * FROM UNERELATION WHERE ID=100;
```

→

ID	NAME
100	?ph?m?re



# Création de domaine

---

- Il est possible de définir son propre domaine pour faciliter la lecture des schémas

**CREATE DOMAIN** Nom\_Domaine **AS** Type;

**CREATE DOMAIN** Adresse **AS VARCHAR(20);**

- Attention ne fonctionne pas avec tous les SGBD. Ne fonctionne pas sous ORACLE g11.



# Les contraintes d'intégrité

---

- **PRIMARY KEY** : pour spécifier une clé primaire
- **FOREIGN KEY** : pour spécifier une clé étrangère
- **REFERENCES** : pour les contraintes d'inclusion
- **CHECK** : contrainte générale
- **UNIQUE** : oblige le fait d'avoir une valeur unique
- **NOT NULL** : pour obliger à saisir une valeur
- **CONSTRAINT** : pour nommer une contrainte –  
Très utile



# CREATE TABLE : contraintes de clés

---

- **PRIMARY KEY** indique que l'attribut est une clé primaire
- On peut avoir plusieurs attributs pour la clé mais il ne peut y avoir qu'une clé primaire

```
CREATE TABLE ECRIT(  
    Auteursnum NUMERIC(10),  
    ISBN NUMERIC(10),  
    Date DATE,  
    CONSTRAINT PK_ECRIT  
        PRIMARY KEY (Auteursnum, ISBN)  
);
```



# CREATE TABLE : contraintes de clés

---

- **PRIMARY KEY** est différent de spécifier **UNIQUE** et **NOT NULL**
- La clé primaire offre des métadonnées sur les concepts du schéma en outre elle apparaît dans les contraintes de la méta-base
- Le fait de nommer une contrainte permet de la voir apparaître facilement dans la méta-base



# CREATE TABLE : clés étrangères

---

- Toujours créer les relations statiques en premier
- Il existe deux méthodes pour définir les clés étrangères
  - lors de la déclaration de l'attribut s'il est unique  
att **TYPE REFERENCES** <relation>(att')
  - avec la contrainte  
**FOREIGN KEY** (att1, ... att2) **REFERENCES** <relation> (att1', ...att2')
- Les attributs référencés doivent être déclarés
- Attention une clé étrangère peut être non unique et nulle. Penser à ajouter une contrainte **UNIQUE** et **NOT NULL**



# CREATE TABLE : contraintes de colonnes

---

- La contrainte de colonne ne s'applique qu'à l'attribut
- **NOT NULL** : impose de mettre une valeur pour l'attribut
- **UNIQUE** : impose une valeur différente de celles des autres attributs





# CREATE TABLE : contraintes CHECK

---

- Elles permettent de prendre en compte la contrainte de domaine du modèle relationnel
- **CHECK** (condition) où condition utilisent les opérateurs vus dans le **select**
- **A BETWEEN *a* AND *b***
- **A IN (*a1*, *a2*, ..*an*)**
- **A LIKE *expression***
- **A >, <, =, ... *valeur***
- **DEFAULT *valeur par défaut***



# Exemple

---

- Création d'une relation

```
CREATE TABLE AVION (  
    Avnum NUMBER(7) NOT NULL,  
    Avnom VARCHAR (50) NOT NULL,  
    Couleur VARCHAR(10) CHECK  
        (Couleur IN ('BLANC', 'NOIR',  
                'ROUGE', 'VERT', 'BLEU'))),  
    Loc VARCHAR(20) DEFAULT 'MONTPELLIER',  
    CONSTRAINT PK_AVION PRIMARY KEY (Avnum)  
);
```



# Exemple

---

- Création d'une relation

```
CREATE TABLE VOL (  
    Volnum NUMBER(7),  
    Plnum NUMBER(7),  
    ...  
    CONSTRAINT PK_VOL PRIMARY KEY (Volnum),  
    CONSTRAINT FK_VOL_PILOTE FOREIGN KEY (Plnum)  
        REFERENCES PILOTE(Plnum)  
);
```



# Exemple

---

```
CREATE TABLE VOL(  
    Volnum NUMBER(7),  
    Plnum NUMBER(7),  
    Avnum NUMBER(7),  
    Frequence NUMBER (7),  
    CONSTRAINT PK_VOL PRIMARY KEY(Volnum),  
    CONSTRAINT FK_PILOTE_VOL FOREIGN KEY(Plnum)  
        REFERENCES PILOTE(Plnum),  
    CONSTRAINT FK_AVION_VOL FOREIGN KEY (Avnum)  
        REFERENCES AVION(Avnum),  
    CONSTRAINT FREQUENCECHECK CHECK (Frequence > 10)  
);
```

Attention un numéro d'avion ou de pilote peut être NULL et non unique ici



# Exemple

---

```
CREATE TABLE PILOTE(  
    Plnum NUMBER(7),  
    Plnom VARCHAR(10),  
    Sal NUMBER (7),  
    Email VARCHAR(50) NOT NULL, CHECK (Email LIKE '%@%'),  
    CONSTRAINT PK_PILOTE PRIMARY KEY(Plnum),  
    CONSTRAINT SALAIRECHECK CHECK (Sal > 15)  
);
```

SALAIRECHECK est une contrainte nommée qui oblige à avoir un salaire supérieur à 15 Keuros lors des insertions ou mises à jour

Email nécessite d'avoir un @ dans l'adresse mail



# Exemple

```
CREATE TABLE PILOTE(  
    Plnum NUMBER(7) PRIMARY KEY  
);
```

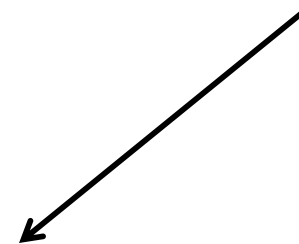
```
SQL> INSERT INTO PILOTE VALUES (100);
```

1 row created.

```
SQL> INSERT INTO PILOTE VALUES (100);
```

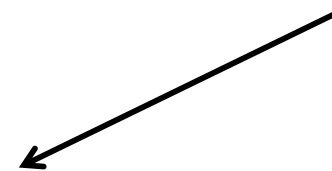
ERROR at line 1:ORA-00001: unique constraint (USER.**SYS\_C00139510**) violated

Un numéro



```
CREATE TABLE PILOTE(  
    Plnum NUMBER(7),  
    CONSTRAINT PK_PILOTE PRIMARY KEY(Plnum)  
);
```

Nom de la contrainte  
PK = Primary Key



```
SQL> INSERT INTO PILOTE VALUES (100);
```

ERROR at line 1:ORA-00001: unique constraint (USER.**PK\_PILOTE**) violated



# Création d'une relation à partir d'un résultat

---

- Il est possible de créer une relation à partir d'une requête :

```
CREATE TABLE COPIEAVION AS  
(SELECT * FROM AVION);
```

```
CREATE TABLE AVIONPARIS AS  
(SELECT *  
FROM AVION  
WHERE Loc='PARIS');
```



# Mise à jour de tuples

---

- Rappel : cela dépend du Langage de Manipulation de Données (LMD)

```
INSERT INTO PILOTE (PInum, PInom, Adr, Sal)  
  VALUES (206, 'DUPOND', 'MONTPELLIER', 30);
```

```
UPDATE PILOTE SET Adr='PARIS', Sal=Sal*1.1  
  WHERE PInom = 'DUPONT';
```

```
DELETE FROM PILOTE WHERE PInum=206;
```





# Mise à jour de tuples

---

```
CREATE TABLE VOL(  
  Volnum NUMBER(7),  
  Plnum NUMBER(7),  
  CONSTRAINT FK_PILOTE_VOL FOREIGN KEY(Plnum) REFERENCES PILOTE(Plnum)  
);
```

- Le pilote 206 est utilisée dans la relation VOL

```
DELETE FROM PILOTE WHERE Plnum=206;
```

```
*ERROR at line 1:ORA-02292: integrity constraint (USER.FK_VOL_PILOTE)  
violated - child record found
```

- Indique que la contrainte de clé étrangère n'est pas respectée



# Mise à jour de tuples

---

- C'est une opération dangereuse ! Le SGBD doit garantir les contraintes de domaine, de clé primaire et de clé étrangère
- L'insertion ne pose pas de problème elle est gérée par le SGBD et les contraintes
- Modification ou suppression de clé étrangère
  - Quid lorsque l'on supprime/modifie un attribut clé primaire qui est référencé par une clé étrangère ?
- Il existe une contrainte supplémentaire pour traiter ce problème

**ON [DELETE | UPDATE] [CASCADE | SET NULL]**



# Stratégie de propagation des modifications

---

- **DEFAULT** : rejet de la modification
- **CASCADE** : fait les mêmes changements que dans la relation R. Si un tuple de R est effacé ou mis à jour alors tous les tuples de S qui référencent un tuple de R sont effacés ou mis à jour
- **SET NULL** : les tuples de S référençant un tuple de R qui est supprimé sont mis à **NULL** pour l'attribut clé de R
- (rappel : avec **SET NULL** on voit bien qu'on peut avoir des valeurs NULLES pour des clés étrangères)



# Stratégie de propagation des modifications

---

- La stratégie de propagation est définie lors de la déclaration d'une clé étrangère

**ON [DELETE | UPDATE] [CASCADE | SET NULL]**

- La stratégie par défaut est de rejeter l'effacement et la mise à jour
- Les différents cas
  - **ON DELETE CASCADE – ON DELETE SET NULL**
  - **ON UPDATE CASCADE – ON UPDATE SET NULL**



# ON DELETE

---

- **ON DELETE CASCADE**

- Objectif : supprimer automatiquement toutes les valeurs des attributs qui référencent la valeur
- Conséquences : les tuples dans la relation associée sont supprimés

- **ON DELETE SET NULL**

- Objectif : mettre à **NULL** toutes les valeurs de clés étrangères associées
- Conséquences : possible si la clé étrangère associée est **NULL**. Impossible si la clé a la contrainte **NOT NULL**



# Exemple

```
CREATE TABLE VOL(  
    Volnum NUMBER(7),  
    Plnum NUMBER(7),  
    Avnum NUMBER(7),  
    CONSTRAINT PK_VOL PRIMARY KEY(Volnum),  
    CONSTRAINT FK_PILOTE_VOL FOREIGN KEY(Plnum)  
        REFERENCES PILOTE(Plnum) ON DELETE CASCADE,  
    CONSTRAINT FK_AVION_VOL FOREIGN KEY (Avnum)  
        REFERENCES AVION(Avnum) ON DELETE CASCADE  
);
```

Il s'agit bien de la relation  
qui contient la clé étrangère

La suppression d'un pilote ou d'un avion éliminera automatiquement tous les tuples qui les référencent dans la relation VOL



# Exemple

---

```
CREATE TABLE VOL(  
    Volnum NUMBER(7),  
    Plnum NUMBER(7),  
    Avnum NUMBER(7),  
    CONSTRAINT PK_VOL PRIMARY KEY(Volnum),  
    CONSTRAINT FK_PILOTE_VOL FOREIGN KEY(Plnum)  
        REFERENCES PILOTE(Plnum) ON DELETE SET NULL,  
    CONSTRAINT FK_AVION_VOL FOREIGN KEY (Avnum)  
        REFERENCES AVION(Avnum) ON DELETE SET NULL  
);
```

La suppression d'un pilote ou d'un avion mettra à **NULL** les pilotes de la relation VOL lorsque l'on supprimera un pilote dans la relation PILOTE



# ON DELETE

---

```
CREATE TABLE VOL(  
    Volnum NUMBER(7),  
    Plnum NUMBER(7) NOT NULL,  
    CONSTRAINT FK_PILOTE_VOL FOREIGN KEY(Plnum)  
        REFERENCES PILOTE(Plnum) ON DELETE SET NULL,  
);
```

```
SQL> DELETE PILOTE WHERE Plnum=1;
```

\*ERROR at line 1:ORA-01407: cannot update ( "USER"."VOL"."PLNUM") to NULL ON DELETE CASCADE

- **Solutions :**
  - Supprimer la contrainte **NULL** (voir Alter Table)
  - Supprimer le tuple à la main dans VOL
  - ...





# ON UPDATE

---

- **LE ON UPDATE** n'est pas si simple !
- Cela veut dire que l'on change une clé primaire. Quelles sont les conséquences ?
- **ON UPDATE CASCADE**
  - Objectif : répercuter les modifications dans la relation associée
  - Conséquences : les tuples associés sont modifiés
- **ON UPDATE SET NULL**
  - Objectif : mettre à **NULL** toutes les valeurs associées
  - Conséquences : **NULL** pour la clé étrangère associée



# Exemple

---

```
CREATE TABLE VOL(  
    Volnum NUMBER(7),  
    Plnum NUMBER(7),  
    CONSTRAINT PK_VOL PRIMARY KEY(Volnum),  
    CONSTRAINT FK_PILOTE_VOL FOREIGN KEY(Plnum)  
        REFERENCES PILOTE(Plnum) ON UPDATE CASCADE  
);
```

```
UPDATE PILOTE SET Plnum = 20 WHERE Plnum = 10;
```

Modifiera automatiquement dans VOL les tuples où le numéro de pilote Plnum était égal à 10



# La réalité

- Etant donné le risque associé, même s'il existe dans la norme le **ON UPDATE CASCADE** est très peu implanté dans les SGBD.  
Fonctionne sur MySQL mais pas sous ORACLE

- Généralement utilisation de triggers (voir plus tard)

```
CREATE OR REPLACE TRIGGER Update_Table_PILOTE
AFTER UPDATE OF Plnum ON Table_Pilote
REFERENCING OLD AS OLD NEW AS NEW FOR EACH ROW
DECLARE
    V_NEWID NUMBER (7);
    V_OLDID NUMBER(7);
BEGIN
    V_NEWID := :NEW.Plnum;
    V_OLDID := :OLD.Plnum;
    UPDATE Table_PILOTE SET Plnum = V_NEWID WHERE Plnum = V_OLDID;
END;
/
```



# Création d'index

---

- Les index sont utilisés pour accélérer les accès à une relation
- Ils sont là pour optimiser les requêtes

```
CREATE INDEX nom_index ON nom_table  
      (nomdesattributs);
```

```
CREATE INDEX IPILOTE ON PILOTE (PLNUM);
```



# Création d'index

---

- La plupart du temps lorsque vous créez une contrainte de clé primaire, étrangère ou une contrainte d'unicité, le SGBDR implante automatiquement un index pour assurer la mécanisme de contrainte avec des performances correctes.
- En effet une contrainte d'unicité est facilité si un tri sur les données de la colonne peut être activé très rapidement.



# Création d'index

---

- Pour une relation donnée, il convient d'indexer dans l'ordre :
  - les colonnes composant la clé primaire
  - les colonnes composant les clés étrangères
  - les colonnes composant les contraintes d'unicité
  - les colonnes dotées de contraintes de validité
  - les colonnes fréquemment mises en relation, indépendamment des jointures
  - les colonnes les plus sollicitées par les recherches



# Création d'index

---

- Lors de la création d'une relation il faut a minima :
  - Créer un index sur la clé primaire
  - Créer un index sur les clés étrangères
- Elles seront toutes les deux très sollicitées par les requêtes

# Suppression d'une base

---

**DROP SCHEMA nom\_base [CASCADE | RESTRICT];**

- **CASCADE** : effacer toute la base
- **RESTRICT** : effacer seulement si la base n'a plus de tuples





# Suppression d'une relation

---

- **DROP TABLE** nom de la relation [**CASCADE** | **RESTRICT**];

**DROP TABLE AVION;**

- Si l'option **RESTRICT** est spécifiée, la table ne sera supprimée que si elle n'est pas référencée par aucune contrainte (e.g. clé étrangère)



# Suppression d'une relation

---

**DROP TABLE AVION IF EXISTS;**

- Existe sous MySQL mais n'existe pas sous ORACLE. N'est pas dans la norme



# Suppression d'une relation

---

- Sous Oracle :

**BEGIN**

**EXECUTE IMMEDIATE 'DROP TABLE AVION';**

**EXCEPTION**

**WHEN OTHERS THEN**

**IF SQLCODE != -942 THEN**

**RAISE;**

**END IF;**

**END;**

**/**

Le / est important

**DROP TABLE AVION IF EXISTS;**

PL/SQL procedure successfully completed.

- Peut être mis dans un script. Explication dans le cours de PL/SQL



# Modification d'une relation

---

- **ALTER TABLE nom\_colonne  
[ADD|MODIFY|DROP][CASCADE|RESTRICT];**

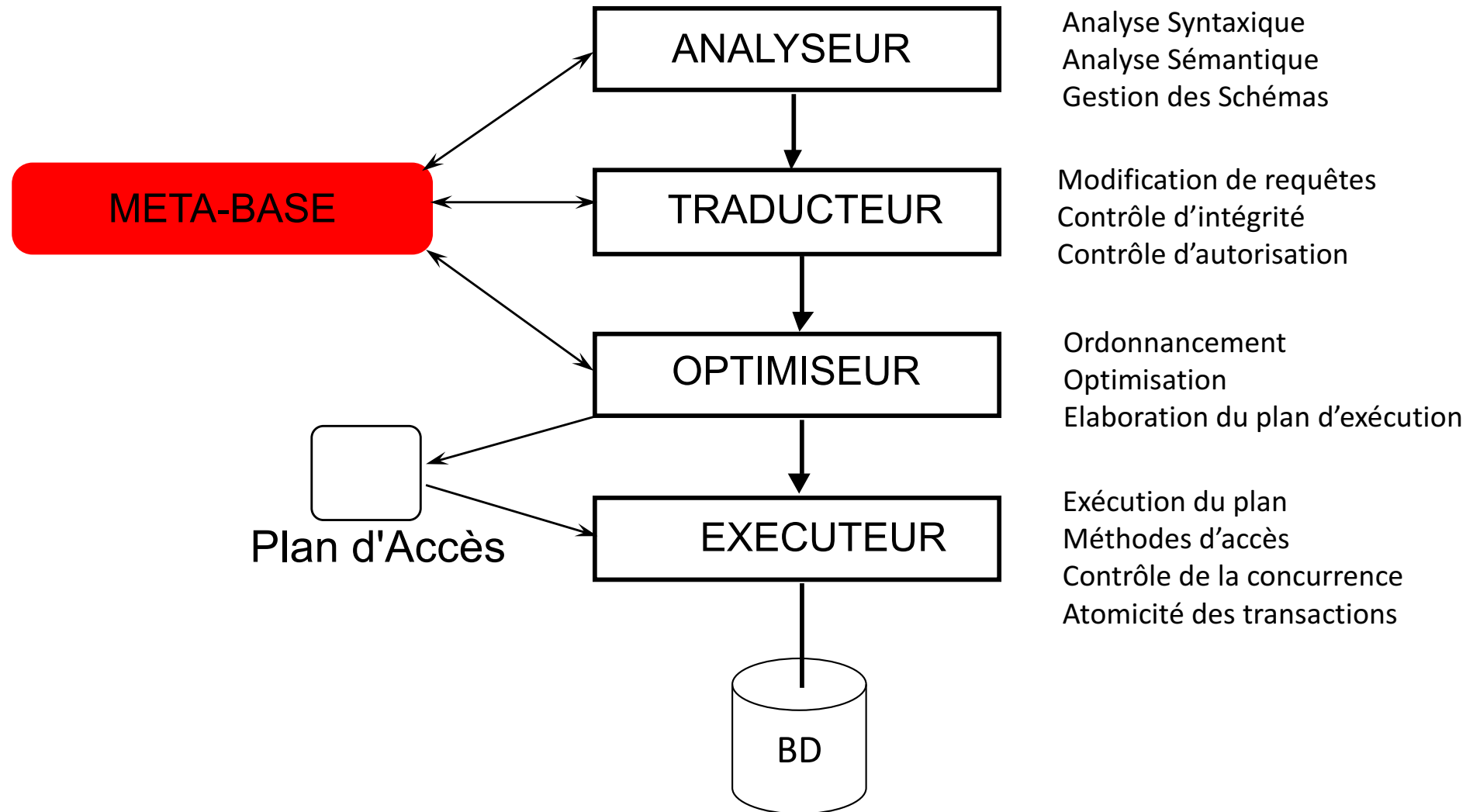
**ALTER TABLE PILOTE ADD TEL VARCHAR(10);**

**ALTER TABLE PILOTE DROP CONSTRAINT SALAIRECHECK;**

**ALTER TABLE PILOTE MODIFY (Loc DEFAULT 'PARIS');**

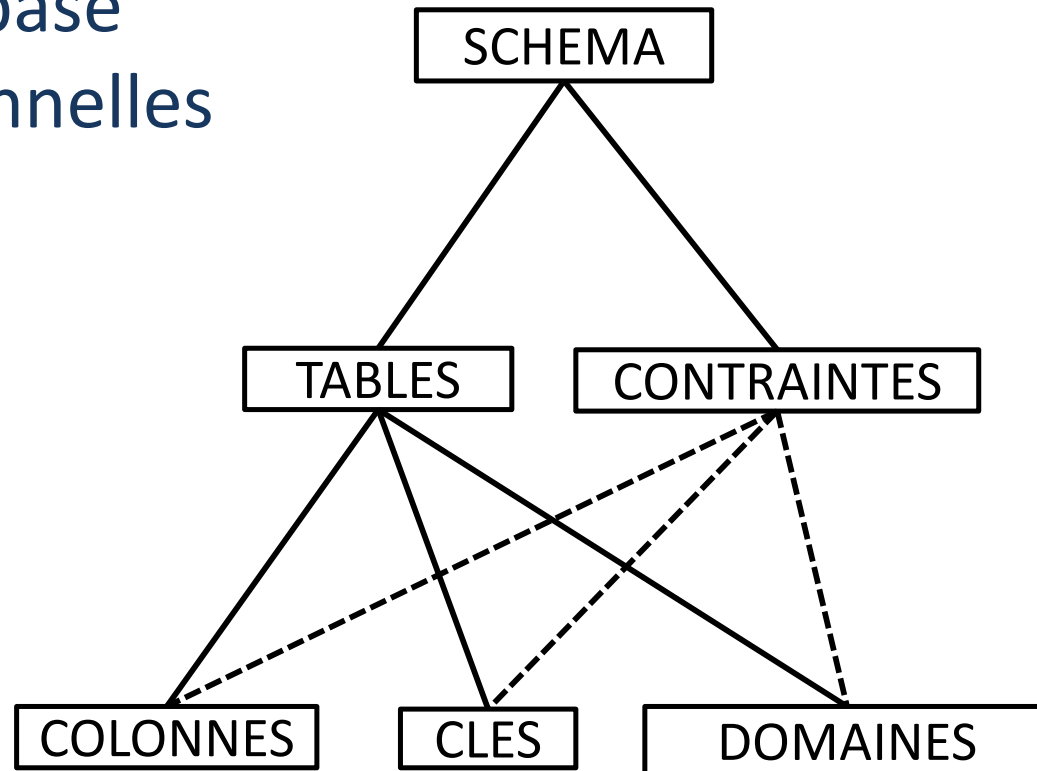


# Structure d'un SGBD



# Gestion du schéma

- Les schémas des BD sont gérés comme une base de données relationnelles appelée méta-base



# Méta-base relationnelle

---

- Structure typique de la méta-base
  - SCHEMAS (CATALOG, NOMB, Créateur, Caractère\_Set, ...)
  - TABLES (CATALOG, NOMB, NOMR, Type, ...)
  - DOMAINS (CATALOG, NOMB, NOMD, Type, Défaut, Contrainte, ...)
  - COLUMNS (CATALOG, NOMB, NOMR, NOMA, Pos, Type, ...)
  - TYPES (CATALOG, NOMB, NOM, MaxL, Precision, ...)
  - CONSTRAINTS (CATALOG, NOMB, NOMC, TypeC, NomR, ...)
  - USERS (NOM, ...)
- Manipulable directement via SQL



# Méta-base relationnelle

---

- Attention dépendant du SGBD :
  - Mysql* : **SHOW** TABLES;
  - Sybase* : **SELECT \* FROM** SYSTABLES;
  - Oracle* : **SELECT \* FROM** ALL\_TABLES;
- ORACLE : Tables systèmes importantes
  - ALL\_TABLES :
    - table des tables - contient des informations sur les différentes relations de la base
  - ALL\_TAB\_COLUMNS
    - table des colonnes - décrit les attributs de toutes les relations de la base (de données ou système)
  - ALL\_CONSTRAINTS

**DESCRIBE TABLE** NOMREL; : description de la relation

**DESC** NOMREL; : description de la relation





# Méta-base relationnelle

---

- Lister les tables du schéma de l'utilisateur courant :  
**SELECT TABLE\_NAME FROM USER\_TABLES;**
- Lister les tables accessibles par l'utilisateur courant :  
**SELECT TABLE\_NAME FROM ALL\_TABLES;**
- Lister les tables d'un utilisateur DUPONT :  
**SELECT TABLE\_NAME FROM ALL\_TABLES WHERE  
OWNER='DUPONT';**
- Lister toutes les tables (il faut être ADMINISTRATEUR) :  
**SELECT TABLE\_NAME FROM DBA\_TABLES;**



# Méta-base relationnelle

---

- De l'intérêt de renommer ses contraintes
- Il est possible d'interroger la méta-base pour connaître l'ensemble des contraintes de type clé primaire :

```
SELECT *  
FROM ALL_CONSTRAINTS  
WHERE CONSTRAINT_NAME LIKE 'PK_%';
```



# Architecture ANSI/SPARC

---

- Trois niveaux d'abstraction qui assurent :
  - l'indépendance logique et physique des données,
  - autorisent la manipulation de données,
  - garantissent l'intégrité des données et
  - optimisent l'accès aux données.
- L'architecture ANSI/SPARC : un standard pour tout SGBD



# Les différents types d'utilisateurs

---



- Utilisateurs

- Cherchent les informations sans connaître la base de données
- Utilisent des interfaces visuelles, éventuellement du SQL



## Programmeurs d'application

- Construisent les interfaces pour les usagers interactifs
- Spécialistes de SQL



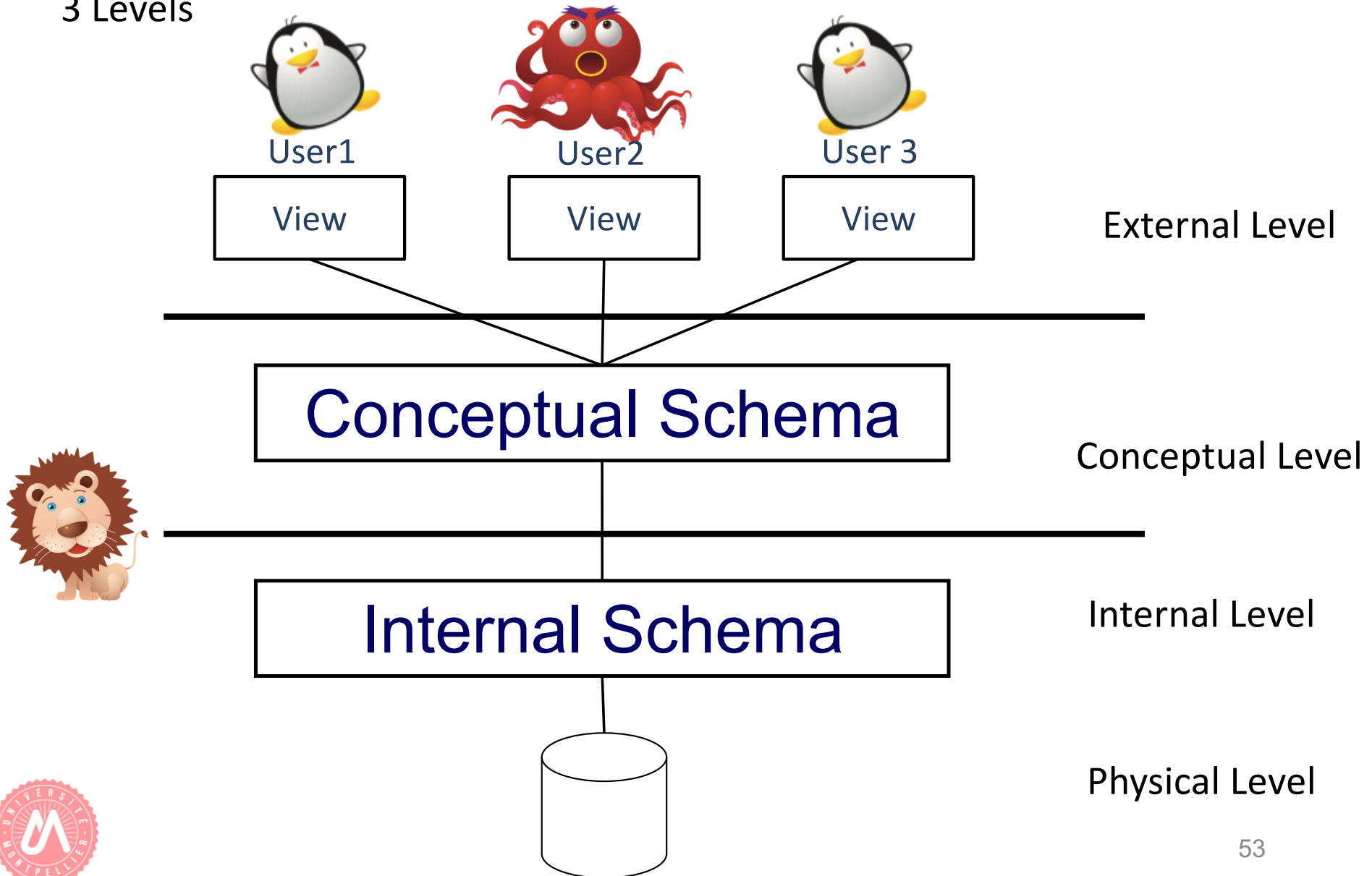
- Administrateur de la base de données

- Définit et maintient la cohérence de la base de données
- A la priorité sur tous les autres usagers



# Architecture ANSI/SPARC

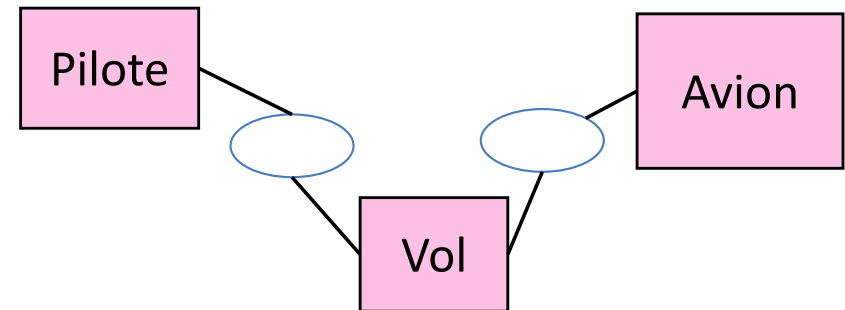
3 Levels



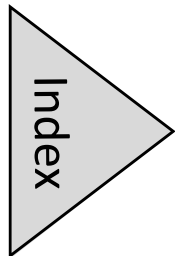
# Architecture ANSI/SPARC

- Externes (vues)
  - description des entités et associations ou plutôt des relations vues par un utilisateur
- Conceptuel
  - description des entités et associations du monde réel
- Interne
  - implémentation physique des entités et associations dans la base sous la forme de relations et d'index

NbVols par pilote		
Plnum	Plnom	NbVols
1	DUPONT	5
2	DUPONT	8
3	DURANT	7



PILOTE		
Plnum	Plnom	Adr
1	DUPONT	NICE
2	DUPONT	PARIS
3	DURANT	LYON



# La notion de vues

---

- Objectif
  - Indépendance logique des applications par rapport à la base
  - Elles permettent de réaliser le niveau externe des SGBD (ANSI/SPARC)
  - Les vues garantissent une meilleure indépendance logique des programmes par rapport aux données
    - Le programme reste invariant aux modifications de la base s'il accède via une vue
- Techniques développées à la fin des années 70
  - Ingres à Berkeley - System R à San José
  - De plus en plus d'actualité (cf. entrepôts)



# La notion de vues

---

- Rôle de sécurité
  - L'utilisateur ne peut accéder qu'aux données des vues auxquelles il a le droit d'accès
- Contraintes d'intégrité
  - Mise à jour au travers de vues
- Rôle de simplification : simplifier les requêtes de l'utilisateur





# La notion de vues

---

- Importance croissante
  - Elles définissent des « relations virtuelles »
  - Client/serveur
    - Optimisation de performances
      - Jointure de deux relations : éviter de faire les opérations sur le client ... seules les données résultantes sont exportées chez le client
  - Entrepôt de données/décisionnel
    - Réaliser à l'avance des cumuls selon plusieurs dimensions



# Définition

---

- Vue
  - Base de données virtuelle dont le schéma et le contenu sont dérivés de la base réelle par un ensemble de requêtes
- Une vue est donc un **ensemble de relations déduites d'une bases de données, par composition des relations de la base**
- Abus de langage : une vue relationnelle est une relation virtuelle



# Création et Destruction

---

- Création d'une vue

**CREATE VIEW** <NOM DE VUE> [ (LISTE D'ATTRIBUT)] **AS**  
<REQUETE>

**[WITH CHECK OPTION]**

- La clause **WITH CHECK OPTION** permet de spécifier que les tuples de la vue insérés ou mis à jour doivent satisfaire aux conditions de la requête



# Création et Destruction

---

- Création d'une vue

**CREATE VIEW** <NOM DE VUE> [ (LISTE D'ATTRIBUTS)] **AS**  
<REQUETE>

**[WITH CHECK OPTION]**

- Ces conditions sont vérifiées après la mise à jour : le SGBD vérifie que les tuples insérés ou modifiés appartiennent à la vue
  - Si la vue possède des attributs d'une seule table et la requête une jointure, les tuples insérés doivent pouvoir vérifier la condition de jointure : contrainte de clé étrangère lors de l'insertion



# Création et Destruction

---

- Destruction d'une vue
  - **DROP** <NOM DE VUE>
  - Destruction de la vue dans la méta-base
  - Remarques :
    - Une vue n'a pas d'existence physique
    - Une destruction n'entraîne pas la suppression de tuples de la base



# Exemple

---

- Création d'une vue pour les pilotes Niçois :

```
CREATE VIEW PILOTESNICOIS (Plnum, Plnom, Sal) AS  
SELECT Plnum, Plnom, Sal  
FROM PILOTE  
WHERE Adr = 'NICE';
```



# Exemple

---

- Création d'une vue pour les gros salaires qui utilisent des Airbus :

```
CREATE VIEW PILOTEAIRBUSGROSSALAIRES AS  
SELECT PILOTE.Plnum, Plnom, Adr, Sal  
FROM PILOTE, AVION, VOL  
WHERE PILOTE.Plnum=VOL.Plnum  
AND AVION.Avnum=VOL.Avnum  
AND Avnom LIKE 'AIRBUS%'  
AND Sal > 20  
WITH CHECK OPTION;
```

**WITH CHECK OPTION** : vérifie lors d'une insertion qu'il existe un pilote et qu'il a bien piloté un Airbus et que son salaire est supérieur à 20



# Exemple

---

- Création d'une vue pour les nombres d'avions pilotés par pilote :

```
CREATE VIEW NBAVIONS (PInum, Total) AS  
SELECT PInum, COUNT(*)  
FROM PILOTE, VOL  
WHERE PILOTE.PInum = VOL.PInum  
GROUP BY PInum;
```





# Importance du **WITH CHECK OPTION**

---

**CREATE VIEW SALELEVE AS**

**SELECT \***

**FROM PILOTE**

**WHERE Sal > 40;**

*View created.*

**INSERT INTO SALELEVE VALUES (50, 'RICHE', 'NICE', 20);**

*1 row created.*

**SELECT \* FROM SALELEVE WHERE Plnum=50;**

*No rows selected*



# Importance du **WITH CHECK OPTION**

---

**SELECT \* FROM SALELEVE WHERE Plnum=50;**

*Pas de pilote numéro 50*

**SELECT \* FROM PILOTE WHERE Plnum=50;**

**<(50, 'RICHE', 'NICE', 20)>**



# Importance du **WITH CHECK OPTION**

---

**UPDATE SALELEVE SET Sal=100 WHERE Plnum=50;**  
*0 rows updated.*

**DELETE FROM SALELEVE WHERE Plnum=50;**  
*0 rows deleted.*

- Solution : détruire le tuple directement dans la relation PILOTE

**DELETE FROM PILOTE WHERE Plnum=50;**  
*1 row deleted.*



# Importance du **WITH CHECK OPTION**

---

**CREATE VIEW SALELEVE AS**

**SELECT \* FROM PILOTE**

**WHERE Sal > 40**

**WITH CHECK OPTION;**

**INSERT INTO SALELEVE VALUES (50, 'RICHE', 'NICE', 20);**

\*ERROR at line 1:ORA-01402: view WITH CHECK OPTION where-clause violation



# Raison d'être des vues

---

- Indépendance logique
  - Le schéma relationnel permet d'isoler l'utilisateur de toute modification intervenant au niveau des chemins d'accès mais **pas des changements concernant la structure des relations**
  - Ces vues sont en général des vues simples comprenant des jointures (avec l'attribut de jointure faisant parti de la vue) mais pas de restrictions



# Raison d'être des vues

---

- Sécurité des données
  - Assurer la sécurité des données manipulées par l'utilisateur
  - Les vues de sécurité sont généralement des vues de restriction (sélection ou projection)

# Raison d'être des vues

---

- Sécurité des données
  - Un utilisateur ne peut pas voir l'attribut Sal de la relation PILOTE

**CREATE VIEW PILOTESANSSALAIRE AS**

**SELECT** Plnum, Plnom, Adr

**FROM** PILOTE

**WITH CHECK OPTION;**

- Un utilisateur ne peut pas voir les données relatives qu'aux pilotes qui n'habitent pas PARIS

**CREATE VIEW PILOTENONPARISIEN AS**

**SELECT \* FROM** PILOTE

**WHERE** Adr <> 'PARIS'

**WITH CHECK OPTION;**



# Raison d'être des vues

---

- Cas de l'opérateur de jointure dans une vue de sécurité : généralement plus pour implanter des restrictions que pour fusionner des informations de plusieurs relations.
  - Ex : « Les pilotes en service au départ de Paris »
  - Rôle de la jointure : vérifier l'appartenance à une « classe »





# Mise à jour de vues

---

- Problème : traduire une mise à jour (modification, suppression, insertion) sur une vue en mise à jour sur les relations de la base
- Conséquences :
  - Problèmes de valeurs indéfinies
  - Problèmes de choix multiple
  - Problème de violation de contrainte d'intégrité



# Mise à jour de vues

---

- Problème des valeurs indéfinis
- Vue : PILOTE1 (Plnum, Plnom)

**INSERT INTO PILOTE1 VALUES (10, 'DUPONT');**

- Donne dans la relation PILOTE le tuple :

**<(10, 'DUPONT', NULL, NULL)>**



# Mise à jour de vues

---

- VOL1(Volnum, VD, VA)
- VOL2 (Volnum, Plnum, VD, VA)
- Problème du **choix multiple**
- Dans la vue VOL1, la modification du trajet d'un vol peut avoir des répercussions possibles dans la relation VOL
- 2 cas possibles :
  - Les Pilotes (Plnum) et les Avions (Avnum) sont toujours affectés au même vol et donc pas modifiés
  - Les Pilotes (Plnum) et les Avions (Avnum) peuvent avoir une affection différente



# Mise à jour de vues

---

- VOL1(Volnum, VD, VA)
- VOL2 (Volnum, plnum, VD, VA)
- Problème de la **violation d'une contrainte d'intégrité**
- Dans la vue VOL2, la modification de l'affectation d'un pilote sur un trajet sans connaître les horaires peut entraîner la violation d'une contrainte d'intégrité dans la relation VOL du type « à une heure donnée (HD ou HA) un pilote ne peut être en service que sur un seul vol »



# Mise à jour de vues

---

- Toute vue ne peut pas mettre à jour
- Vue mettable à jour
  - Vue comportant suffisamment d'information pour permettre un report des mises à jour dans la base sans ambiguïté
  - Une vue peut être mettable à jour en suppression, modification ou insertion



# Mise à jour de vues

---

- La vue les pilotes PARISIENS est tout à fait mettable à jour :

```
CREATE VIEW PILOTEPARISIENS(Plnum, Plnom, Adr, Sal) AS  
SELECT *  
FROM PILOTE WHERE Adr = ' PARIS' ;
```

- Toute opération INSERT, DELETE ou UPDATE est reportable sur la base



# Mise à jour de vues

---

- Création d'une vue pour les nombres d'avions pilotés par pilote :

```
CREATE VIEW NBAVIONS (Plnum, Total) AS  
SELECT Plnum, COUNT(*)  
FROM PILOTE, VOL  
WHERE PILOTE.Plnum = VOL.Plnum  
GROUP BY Plnum;
```

- Impossible de déterminer le nombre d'avions



# Mise à jour de vues

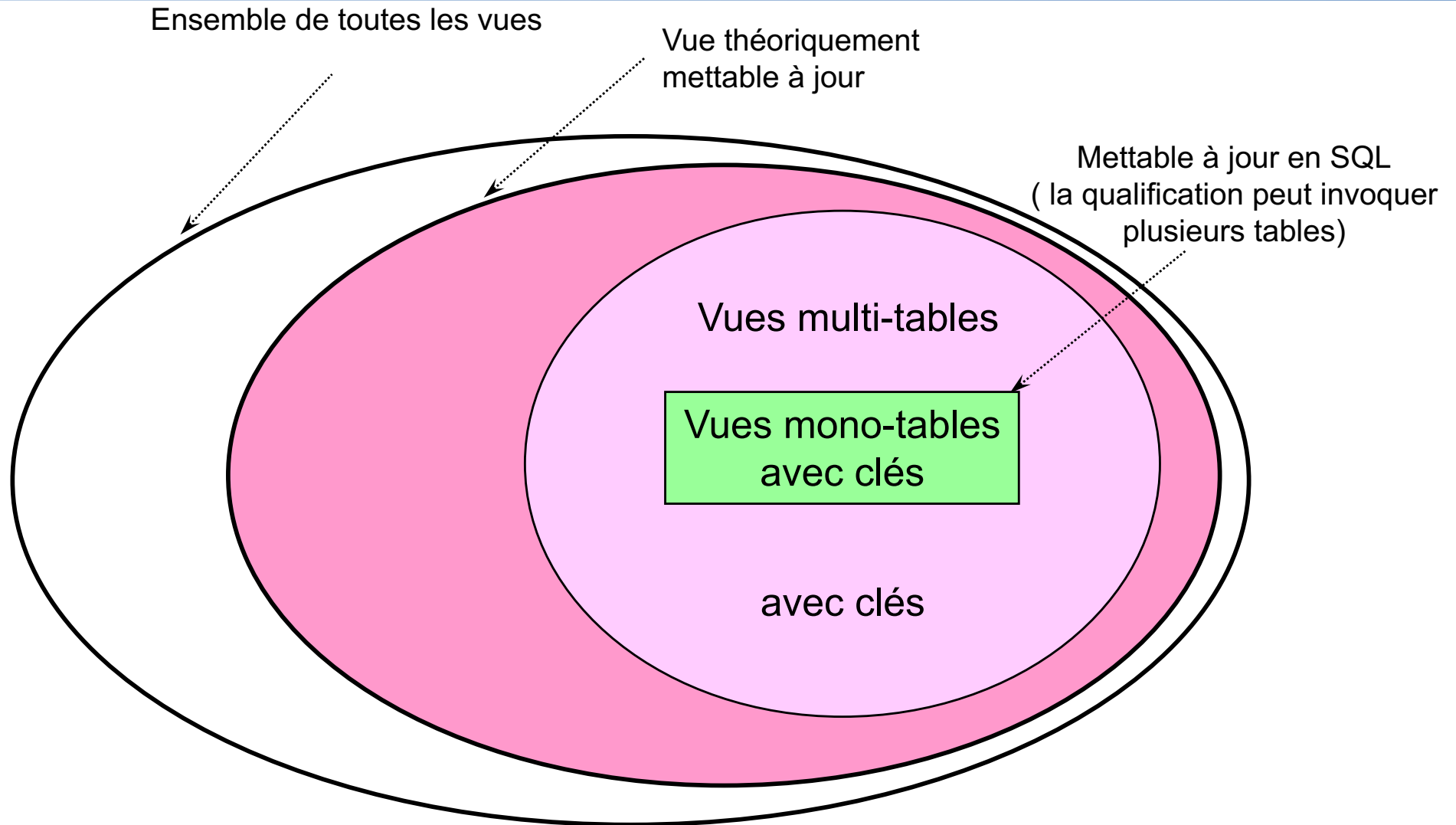
---

- En pratique : seuls les attributs d'une table de la base doivent apparaître dans la vue
- Imposer que la clé de la table soit présente
- Tenir compte des problèmes potentiels d'insertion, de modification ou de suppression
- Importance de la clause **WITH CHECK OPTION** : forcer à faire la vérification





# La réalité ... commerciale



[Gardarin2003]

- 
- Des questions ?

