

# Plan détaillé

- Architecture et objectifs d'un SGBD
- Schémas de base de données
- Passage de la modélisation à l'implémentation
  - MCD – MLD – MPD
- Le langage SQL
  - Manipulation et Interrogation des données
  - Jointures
  - Sous interrogation
  - Opérateurs et fonctions

# Plan détaillé

- Le langage SQL
  - La gestion des vues
  - Procédures et fonctions
  - Déclencheurs
- Manipulation PL/SQL sous oracle
  - Structure
  - Variables
  - Traitements
  - Curseurs
  - Gestions des erreurs

# Plan détaillé

- **Optimisation et réécriture de requêtes**
  - Objectifs de l'optimisation
  - Requêtes statiques et dynamiques
  - Analyse des requêtes
- **Les transactions**
  - Généralités sur les transactions
  - Les transactions dans SQL
- **Gestion des accès concurrents**
  - Problèmes liés aux accès concurrents
  - Mécanismes pour assurer la concurrence et la reprise
- **Programmation avec une BDD**
  - Couplage avec les langages de programmations classiques

# Connaissances acquises

- Comprendre les notions de base liées aux SGBD relationnelles
  - Concevoir des schémas de base de données
  - Manipuler des informations
    - Interroger
    - Saisir et modifier
    - Etc.
- Appréhender des compétences en gestion de transactions et d'accès concurrents

# Méthodes d'évaluation

- Un contrôle écrit
- Un examen écrit
- Tps notés
- Une note d'appréciation personnelle

# Références

- **Notes de cours du professeur**
- **Les moteurs de recherche (google, yahoo, etc.)**
- Langage SQL, version 5.7 du polycopié de Richard Grin- Université de Nice Sophia-Antipolis – 4 janvier 2008
- Bases de données, Best of Georges GARADIN Edition Eyrolles, Sixième tirage 2005, ISBN: 2-212-11281-5



# **SYSTÈME DE GESTION DE BASE DE DONNÉES**

# Des données ? Est ce important pour vous ?

- Des relevés de banques, de cartes de crédit
- Des carnets d'adresses
- La consommation de téléphone
- Des inscriptions à des clubs, associations,
- Des horaires et disponibilités de transport
- Des programmes de télé
- Etc.

# Nécessités...

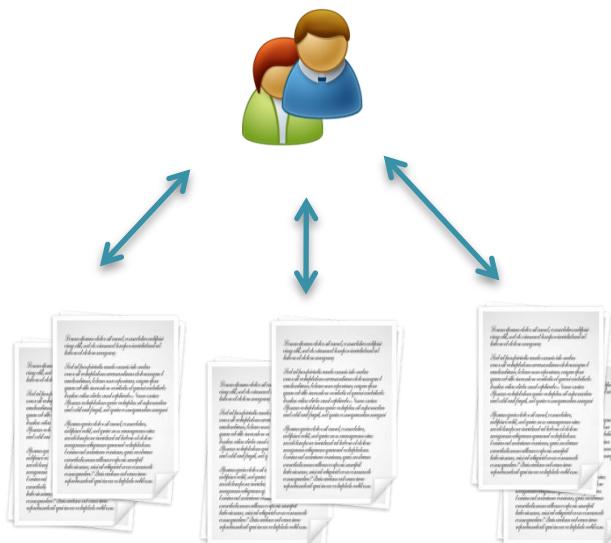
- Stocker les données
- Assurer l'accès aux données
  - Modification
  - Recherche
  - Etc.
- Assurer la sécurité de ces données
  - Confidentialité
  - Authentification
  - Signature digitale
  - Intégrité

# Historique...

- **1950-1960**
  - Des fichiers séquentiels, du ‘batch’
- **1960 – 1970**
  - Le début des bases de données hiérarchiques
- **1970 – 1980**
  - La naissance du modèle relationnel
- **Début des années 90**
  - Sql, l'aide à la décision
- **Fin des années 90**
  - Croissance du volume des données, Internet, modèle multi tiers

# Les limites à l'utilisation des fichiers (I)

- L'utilisation des fichiers impose à l'utilisateur de connaître
  - le mode d'accès (séquentielle, indexée, ...)
  - la localisation des fichiers qu'il utilise afin de pouvoir accéder aux informations dont il a besoin.
  - créer de nouveaux fichiers qui contiendront peut-être des informations déjà présentes dans d'autres fichiers



## Les limites à l'utilisation des fichiers (2)

- De telles applications sont
  - rigides
  - contraignantes
  - longues et coûteuse à mettre en œuvre
- Les données associées sont :
  - mal définies et mal désignées
  - redondantes
  - peu accessibles de manière ponctuelle
  - peu fiables

## Les limites à l'utilisation des fichiers (3)

- Source des difficultés avec les fichiers
  - Le modèle des données est intégré dans les programmes
  - Absence de contrôle pour l'accès et la manipulation des données

**Solution ???**



# **BASE DE DONNÉES**

# Définition intuitive d'une BDD (I)

- Base de Données (BDD)
  - grande quantité de données (ou ensemble d'informations), centralisées ou non
  - permet de communiquer avec une ou plusieurs applications
  - Interrogeable et modifiable par un groupe d'utilisateurs travaillant en parallèle
- Exemples d'application
  - Pages Jaunes
  - Catalogue électronique
  - Etc.

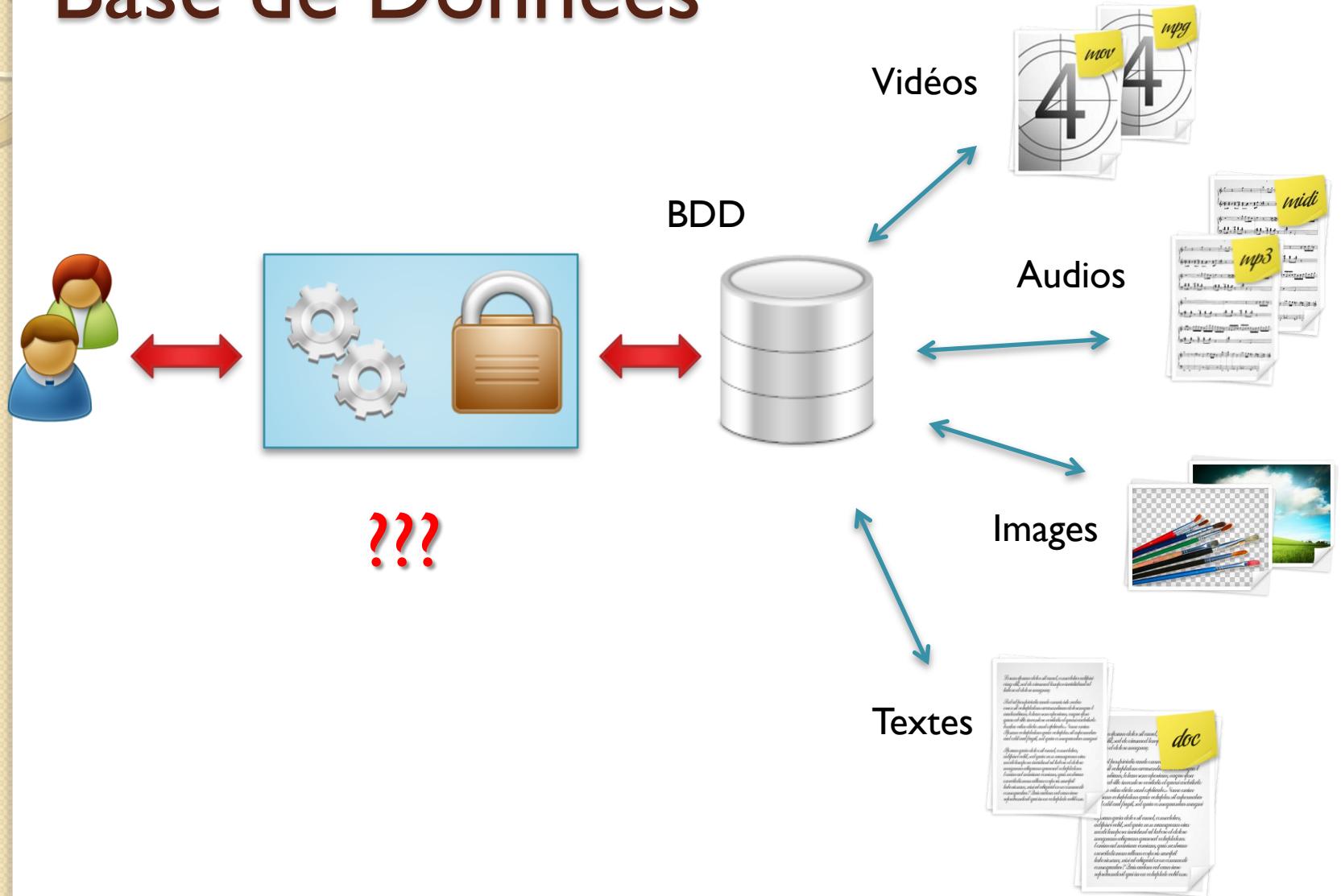
The image contains three separate screenshots:

- Top Screenshot:** A search results page from "pages jaunes". The search bar shows "Quoi, Qui hospital" and the results are filtered for "Où dijon". It lists several entries for hospitals in Dijon, such as "HOPITALS" and "Centre hospitalier universitaire de Dijon".
- Middle Screenshot:** A screenshot of the "PowerBoutique" website. The header says "La solution pour le site internet de votre entreprise". It features sections for "ESSAI GRATUIT", "ÉTUDE DE PROJET", and "CONTACTEZ-NOUS". Below this, there's a sidebar with links like "Présentation", "Graphisme & apparence", "Visibilité & référencement", "Développer votre activité", and "Gérer votre site internet".
- Bottom Screenshot:** A screenshot of a Google search results page for the query "Pages jaunes". The top result is a link to "www.pagesjaunes.com". Other results include "PagesJaunes.com - Dictionnaire des noms de famille", "PagesJaunes.com - Annuaire des professionnels", and "PagesJaunes.com - Annuaire des entreprises".

# Définition intuitive d'une BDD (2)

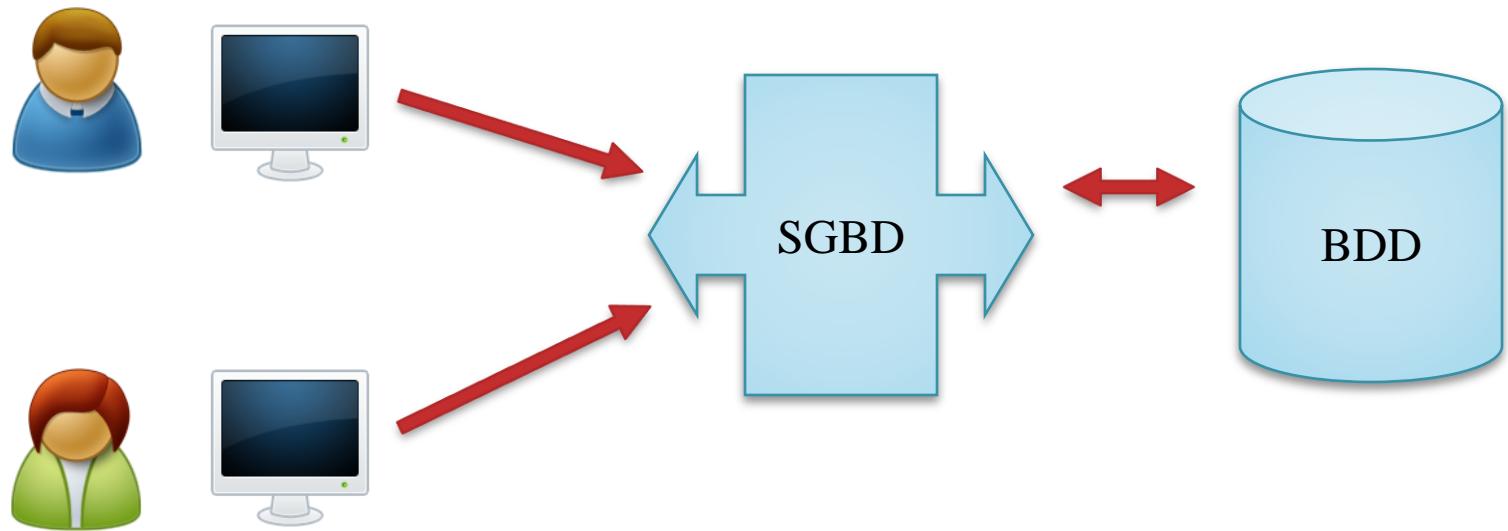
- Une base de données est un ensemble structuré de données
  - Stockage sur disque
  - Partage des données
  - Permet un accès simultané par plusieurs utilisateurs d'une manière sélective
  - Confidentialité
  - Performance

# Base de Données



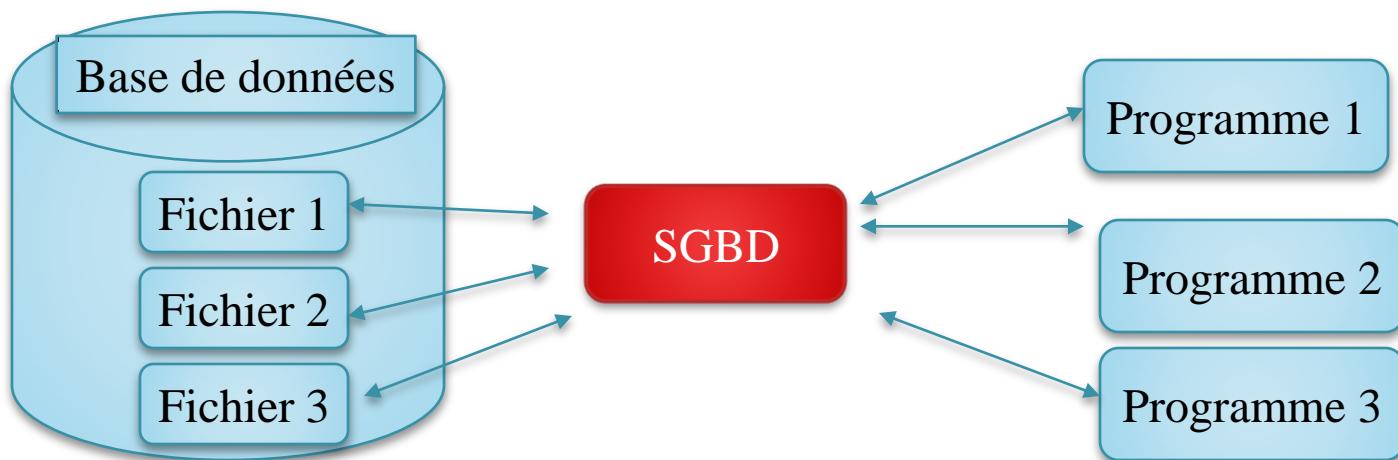
# SGBD (I)

Définition : Le logiciel qui permet d'interagir avec une BDD est le Système de Gestion de Base de Données (SGBD)



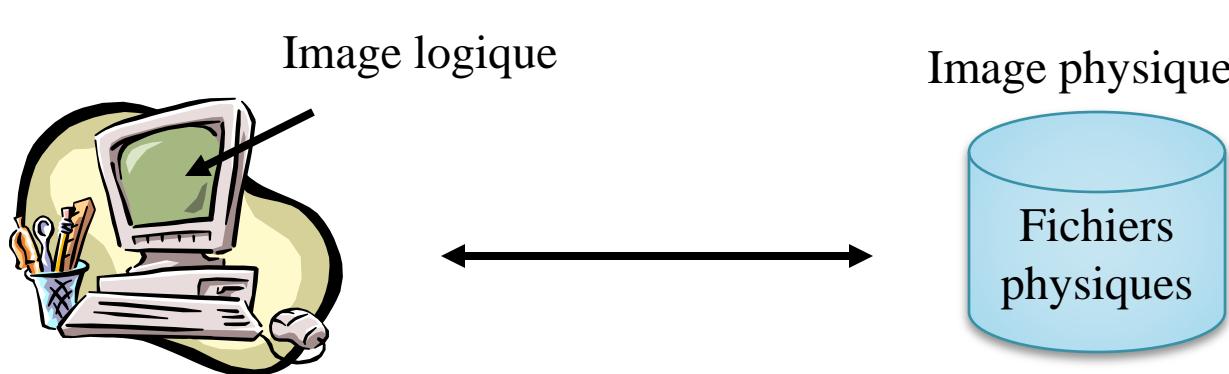
# SGBD (2)

- Un SGBD est un intermédiaire entre les utilisateurs et les fichiers physiques
- Un SGBD facilite
  - la gestion de données
    - une représentation intuitive simple sous forme de table par exemple
  - la manipulation de données
    - On peut insérer, modifier les données et les structures sans modifier les programmes qui manipulent la base de données



# Objectifs des SGBD (I)

- Faciliter la représentation et la description de données
  - Indépendance physique
    - Plus besoin de travailler directement sur les fichiers physiques (tels qu'ils sont enregistrés sur disque).
    - On parle de description logique ou conceptuelle ou encore de schéma logique par exemple le modèle de tables, appelé le modèle relationnel.)



# Objectifs des SGBD (2)

- **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.
- **Manipulations des données par des non informaticiens**
  - Il faut pouvoir accéder aux données sans savoir programmer ce qui signifie des langages « quasi naturels ».
- **Efficacité des accès aux données**
  - Ces langages doivent permettre d'obtenir des réponses aux interrogations en un temps « raisonnable ». Ils doivent donc être optimisés et, entre autre, il faut un mécanisme permettant de minimiser le nombre d'accès disques. Tout ceci, bien sur, de façon complètement transparente pour l'utilisateur.

# Objectifs des SGBD (3)

- **Administration centralisée des données**
  - Des visions différentes des données (entre autres) se résolvent plus facilement si les données sont administrées de façon centralisée.
- **Cohérence des données**
  - Les données sont soumises à un certain nombre de contraintes d'intégrité qui définissent un état cohérent de la base.
  - Les données doivent pouvoir être exprimées et vérifiées automatiquement à chaque insertion, modification ou suppression de données, par exemple :
    - l'âge d'une personne supérieur à zéro
    - Salaire supérieur à zéro
    - Etc
  - Dès que l'on essaie de saisir une valeur qui ne respecte pas cette contrainte, le SGBD la refuse.

# Objectifs des SGBD (4)

- Non redondance des données
  - Afin d'éviter les problèmes lors des mises à jour, chaque donnée ne doit être présente qu'une seule fois dans la base.
- Possibilité de partager des données :
  - Il s'agit de permettre à plusieurs utilisateurs d'accéder aux mêmes données au même moment.
    - Permettre à deux (ou plus) utilisateurs de modifier la même donnée « en même temps »;
    - Assurer un résultat d'interrogation cohérent pour un utilisateur consultant une table pendant qu'un autre la modifie.

# Objectifs des SGBD (5)

- Sécurité des données

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

- Résistance aux pannes

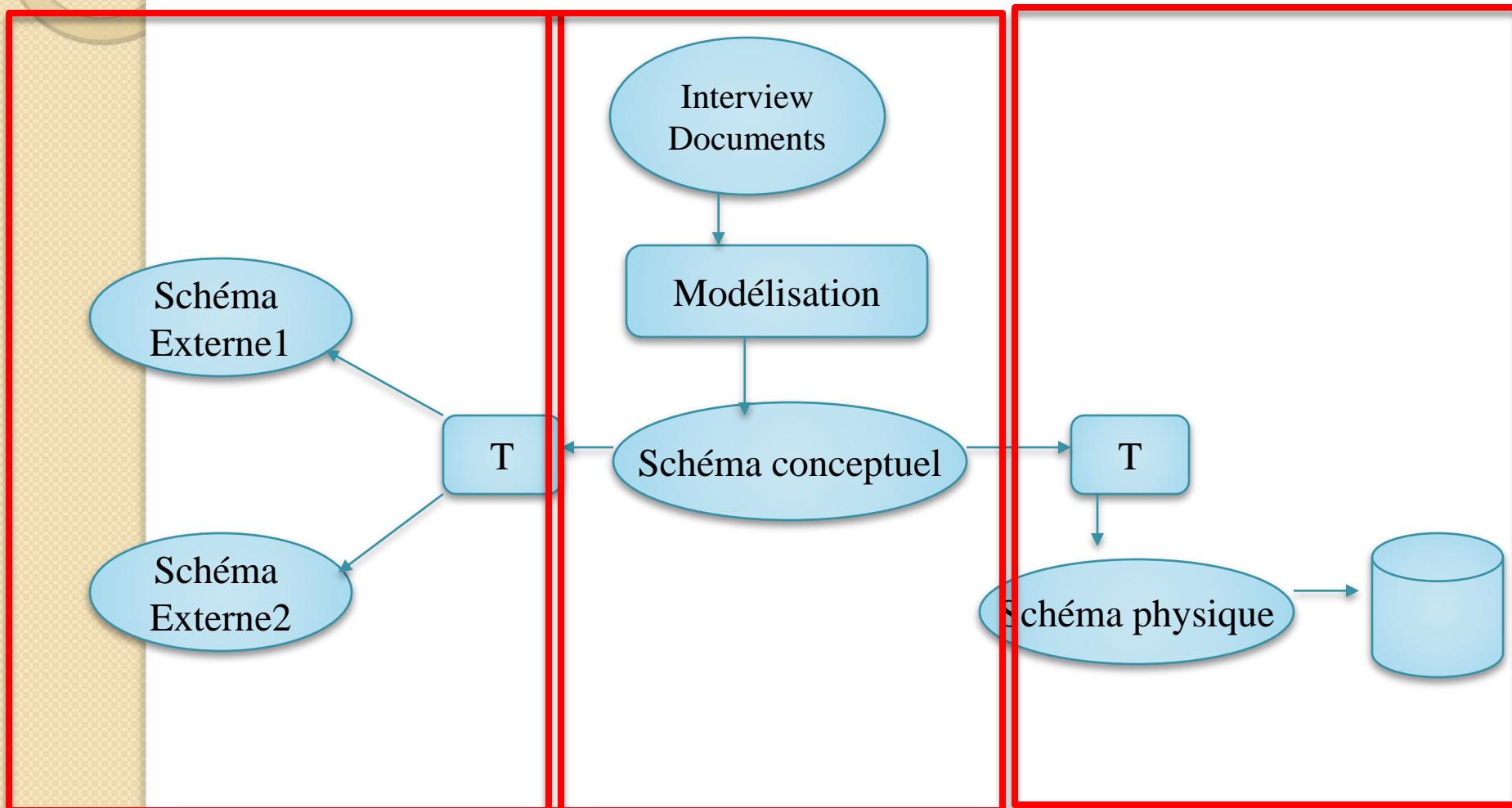
- Que se passe-t-il si une panne survient au milieu d'une modification, si certains fichiers contenant les données deviennent illisibles?
  - Après une panne intervenant au milieu d'une modification deux solutions sont possibles
    - soit récupérer les données dans l'état dans lequel elles étaient avant la modification
    - soit terminer l'opération interrompue

# Trois Fonctions d'un SGBD

- Description des données : codification structuration, grâce à un Langage de Description de Données (LDD)
- Manipulation et restitution des données (insertion, mise à jour, interrogation)
  - Mise en œuvre à l'aide d'un Langage de Manipulation de Données (LMD)
  - S.Q.L. (Structures Query Language) : Langage standard
- Contrôle (partage, intégrité, confidentialité, sécurité)

# Définition et description des données

3 niveaux de description



# Définition et description des données niveau physique

- Description informatique des données et de leur organisation : en terme de fichiers, d'index, de méthodes d'accès, ...
- Passage du modèle logique au modèle physique tend à être assisté par le SGBD : transparent et/ou semi-automatique
- Objectifs : optimiser les performances

# Définition et description des données niveau logique (conceptuel)

- Permet la description
  - Des objets : exemple OUVRAGES, ETUDIANTS
  - Des propriétés des objets (attributs) : exemple Titre de OUVRAGES
  - Des liens entre les objets : un OUVRAVE peut être emprunté par un ETUDIANT
  - Des contraintes : le nombre d'exemplaires d'un OUVRAVE est supérieur à zéro
- Cette description est faite selon un modèle de données
- Un modèle de données est un ensemble de concepts permettant de décrire la structure d'une base de données.
  - Le modèle de données le plus utilisé est le modèle relationnel
- Cette description va donner lieu à un schéma de base de données
  - Un schéma de base de données se compose d'une description des données et de leurs relations ainsi que d'un ensemble de contraintes d'intégrité.

# Définition et description des données niveau externe

- Description des données vues par un utilisateur (ou un groupe d'utilisateurs)
  - Objectifs : simplification, confidentialité
  - Exemple : OUVRAGE édité par des éditeurs français

# Manipulation et restitution des données

- Utiliser SQL pour réaliser les opérations suivantes
  - Insertion : saisir des données
  - Supprimer
  - Modifier
  - Interroger : rechercher des données via des requêtes

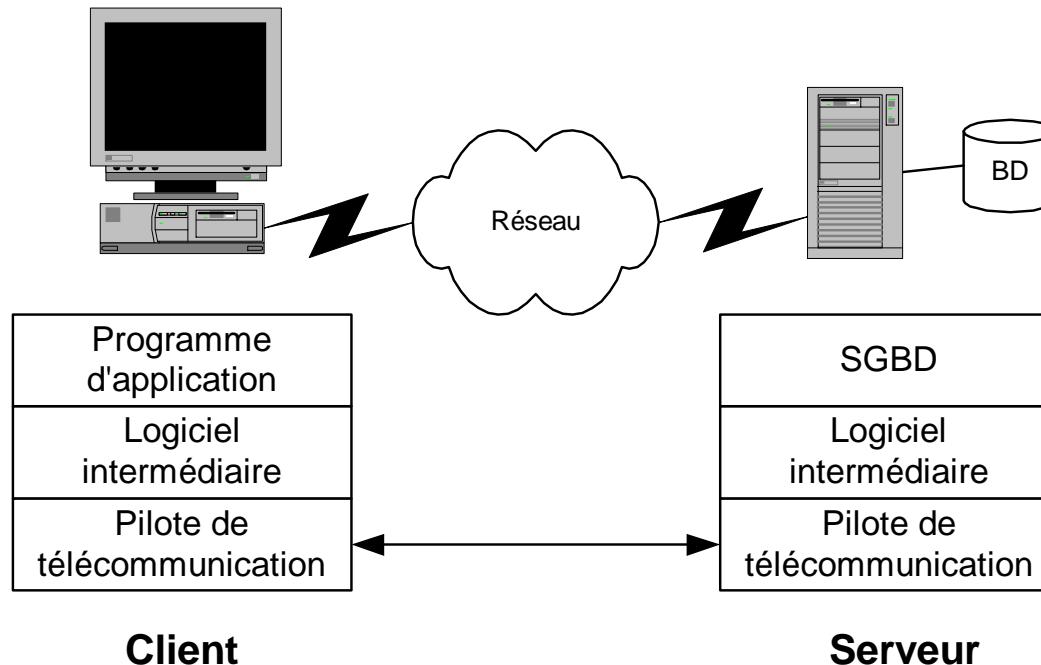
# Modèles de SGBD

- Quelques modèles logiques
  - Modèle hiérarchique
  - Modèle réseau
  - Modèle relationnel
  - Modèle objet
- Quelques SGBD (relationnels du marché)
  - Micro :ACCESS, Paradox, Dbase, PostgreSQL, MySQL, ...
  - Gros systèmes :DB2, ORACLE, SYBASE, ...

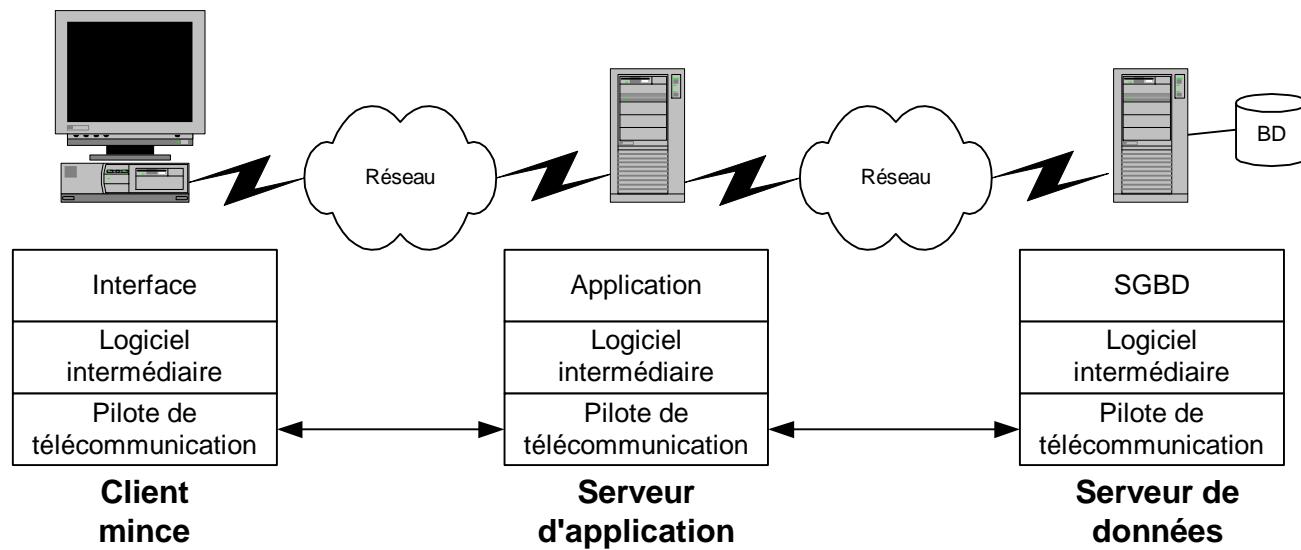
# Architecture

- Architecture centralisée
  - programme d'application et SGBD sur même machine (même site)
- Architecture du type client-serveur (*client-server architecture*)
  - programme d'application = *client*
    - interface (« GUI ») + traitement du domaine d 'application
  - SGBD = *serveur* (*de données* « *data server* »)
  - machines (sites) différentes
  - *deux couches, niveaux, strates* (“*two tier* ”)

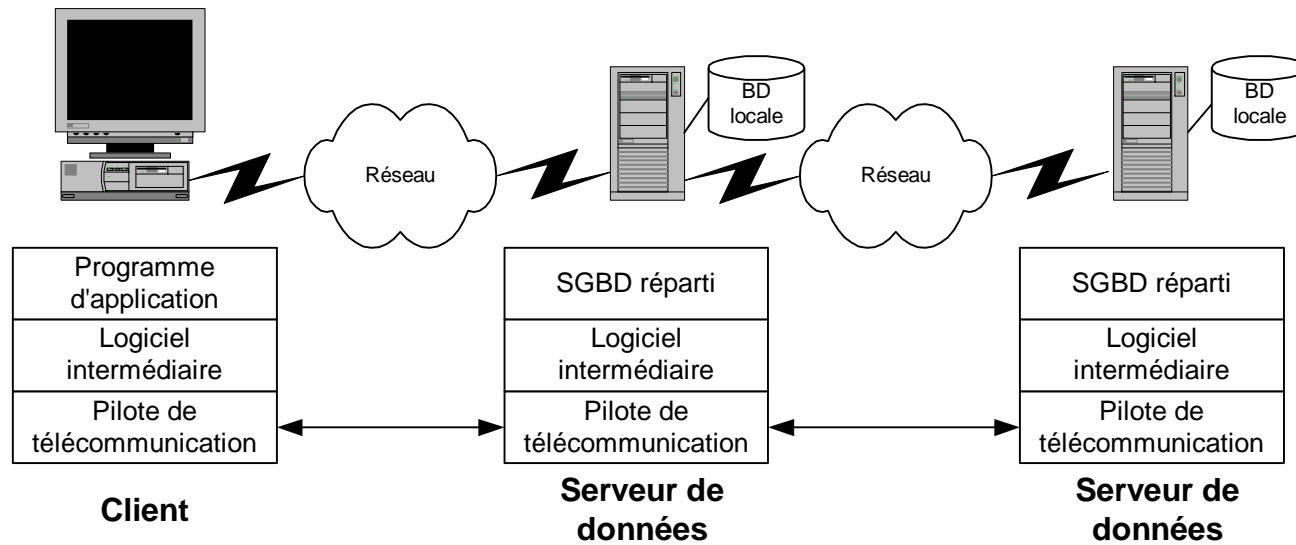
# Architecture client / serveur



# Architecture 3 tiers



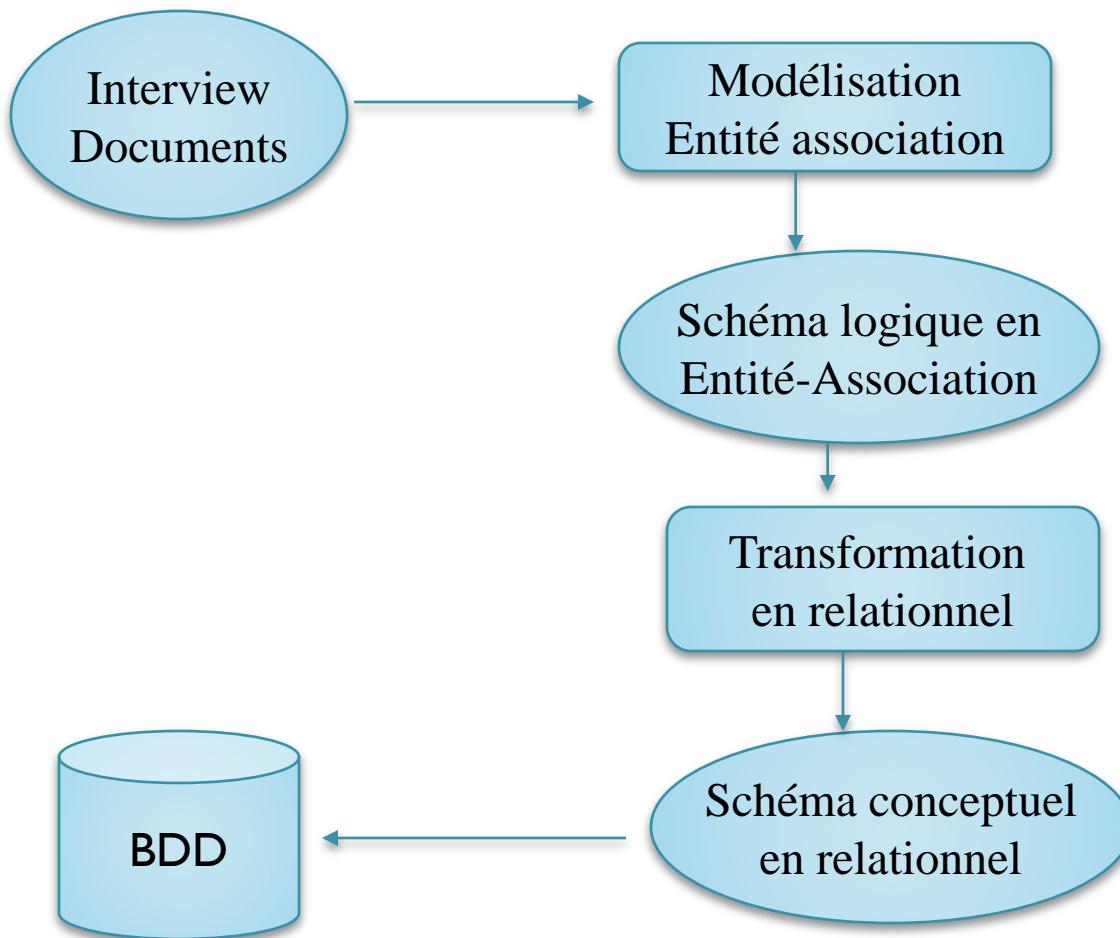
# Bases de données distribuées





# **COMMENT CONCEVOIR UNE BASE DE DONNÉES RELATIONNELLE ?**

# Démarche de construction d'une BDD relationnelle





# MODÉLISATION DE DONNÉES

# Modèle conceptuel de données

- Les travaux d'analyse et de conception doivent être conduits par une méthodologie intégrant la modélisation *Entité-Association*
- Le modèle *Entité-Association* représente un formalisme du Modèle conceptuel de données (MCD)

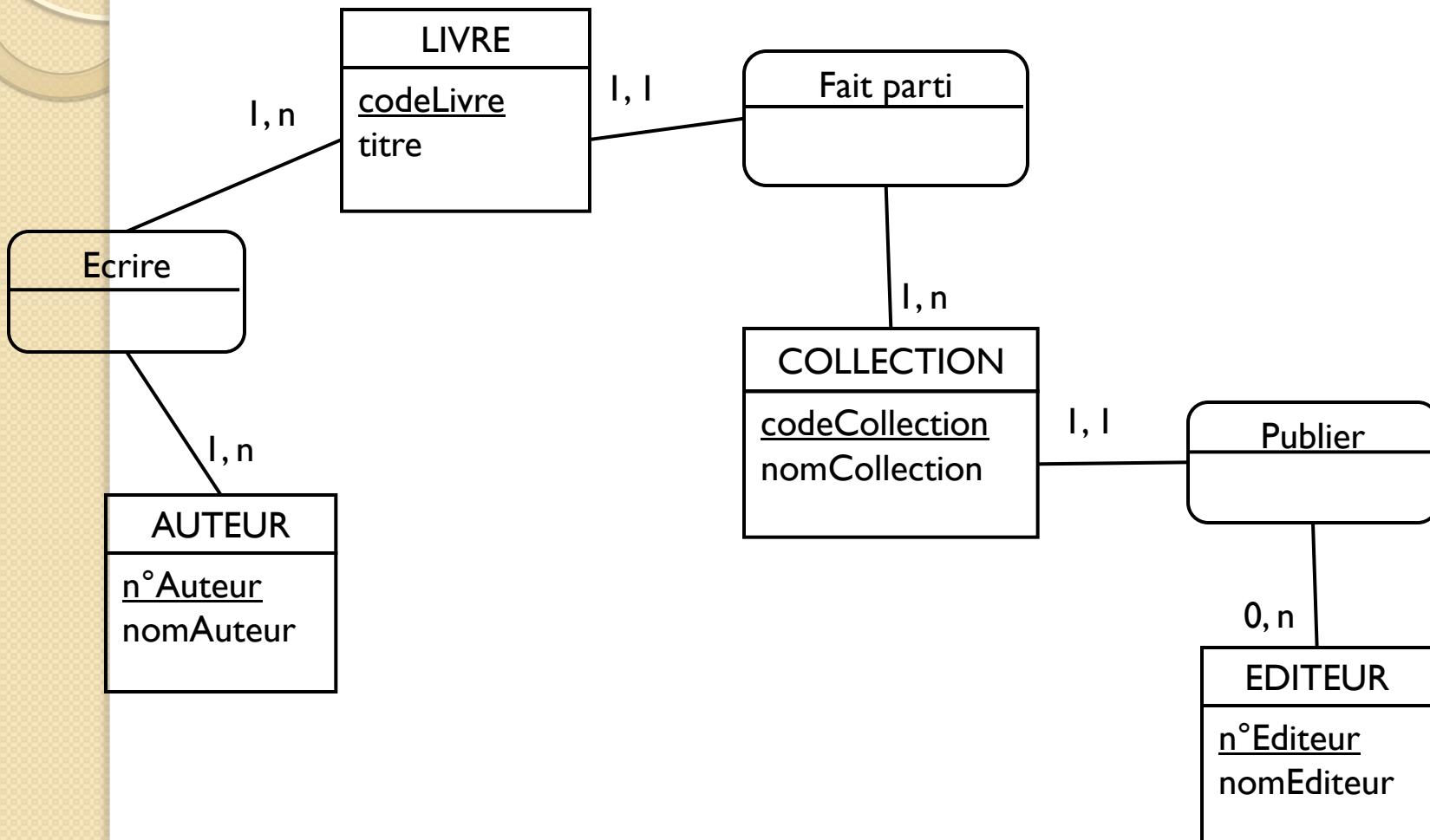
# Modèle Entité-Association

- Outil utilisé dans la gestion et la structuration des données
- Outil rigoureux qui permet d'arriver au schéma relationnel (modèle logique de données)
- Sert à guider la réflexion et forcer à se poser les bonnes questions.

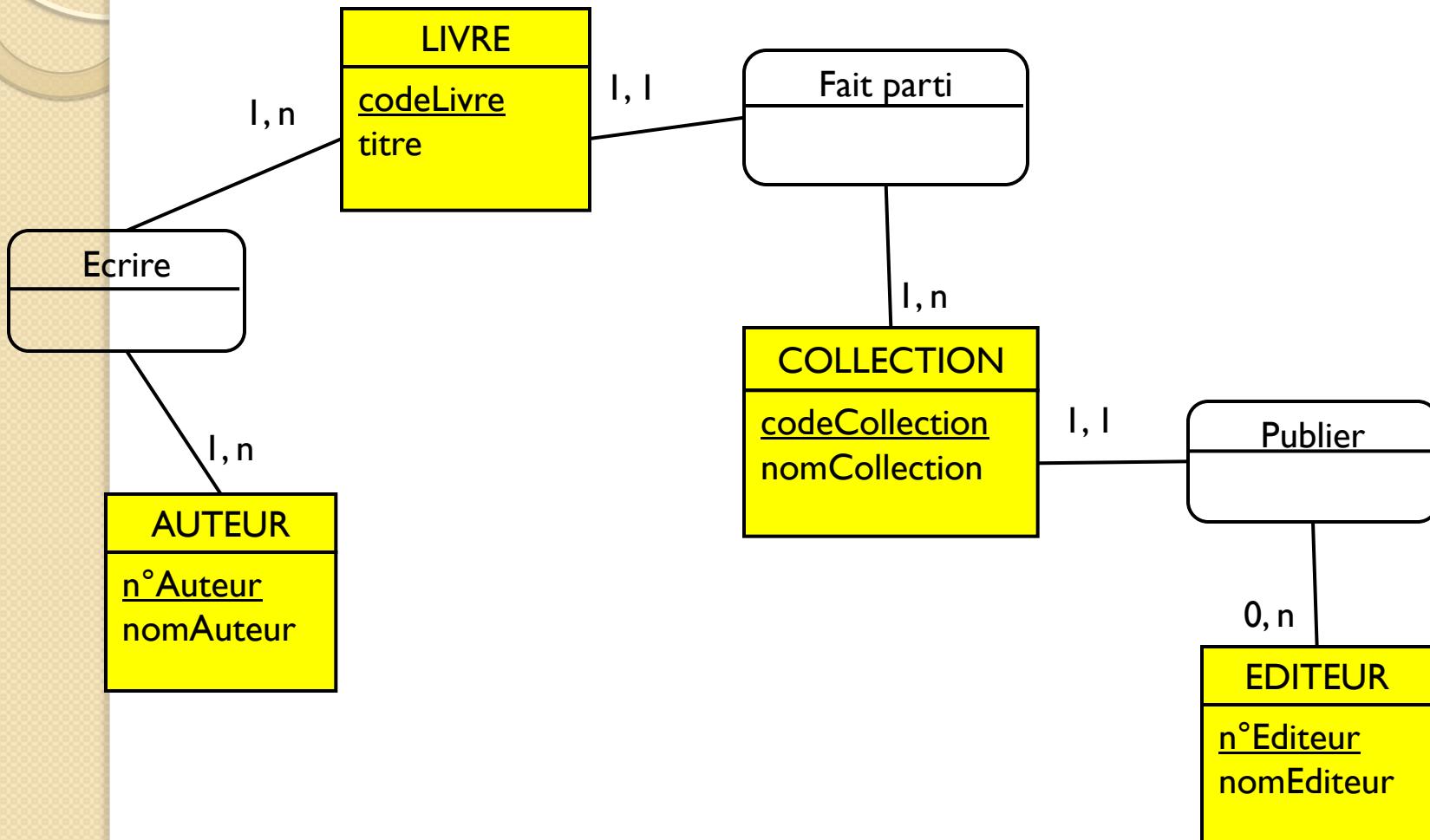
# Modèle Entité-Association

- Modèle E/A
  - Entités
  - Associations
  - Attributs ou Propriétés

# Exemple (MCD)



# Exemple





# **MODÉLISATION LOGIQUE DE DONNÉES**

# Objectif

- ▶ D'apporter à la formalisation conceptuelle et organisationnelle les notions de réalisation technique



**indépendant de la structure de la base de données**

# Modèles Logiques

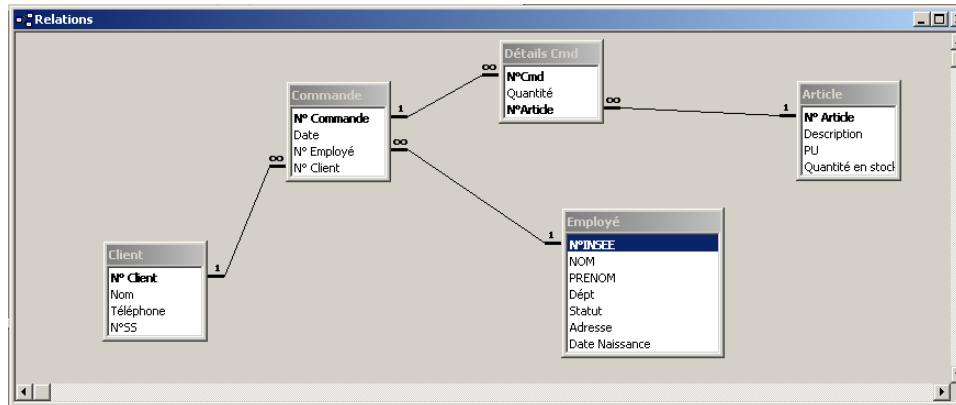
- Il existe plusieurs types de MLD
  - Le MLD « réseau » ou MLD « CODASYL »
  - Le MLD « fichier »
  - **Le MLD « relationnel »**
  - Le MLD « Objet »
  - Le MLD « relationnel-objet »

# Modèle Logique de Données (MLD)



# Le modèle relationnel

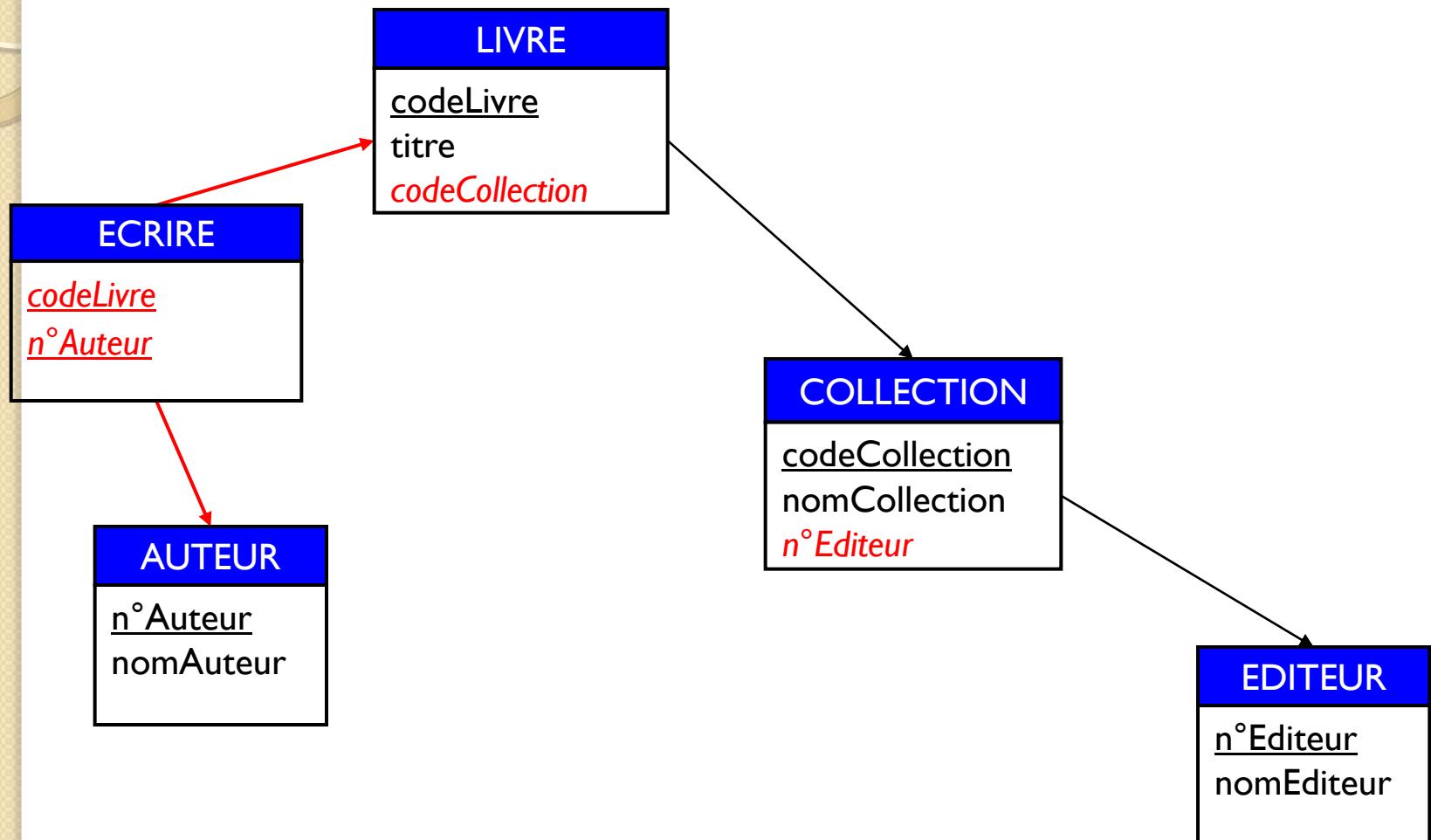
- Défini en 1970 par E.F. Codd à IBM San José
  - Ses concepts découlent de la théorie des ensembles
- Se caractérise par :
  - Une démarche permettant de représenter les données dans une collection (ou schéma) de **tables relationnelles (ou relations)**
    - ATTENTION : à ne pas confondre avec le concept *relation* dans le MCD



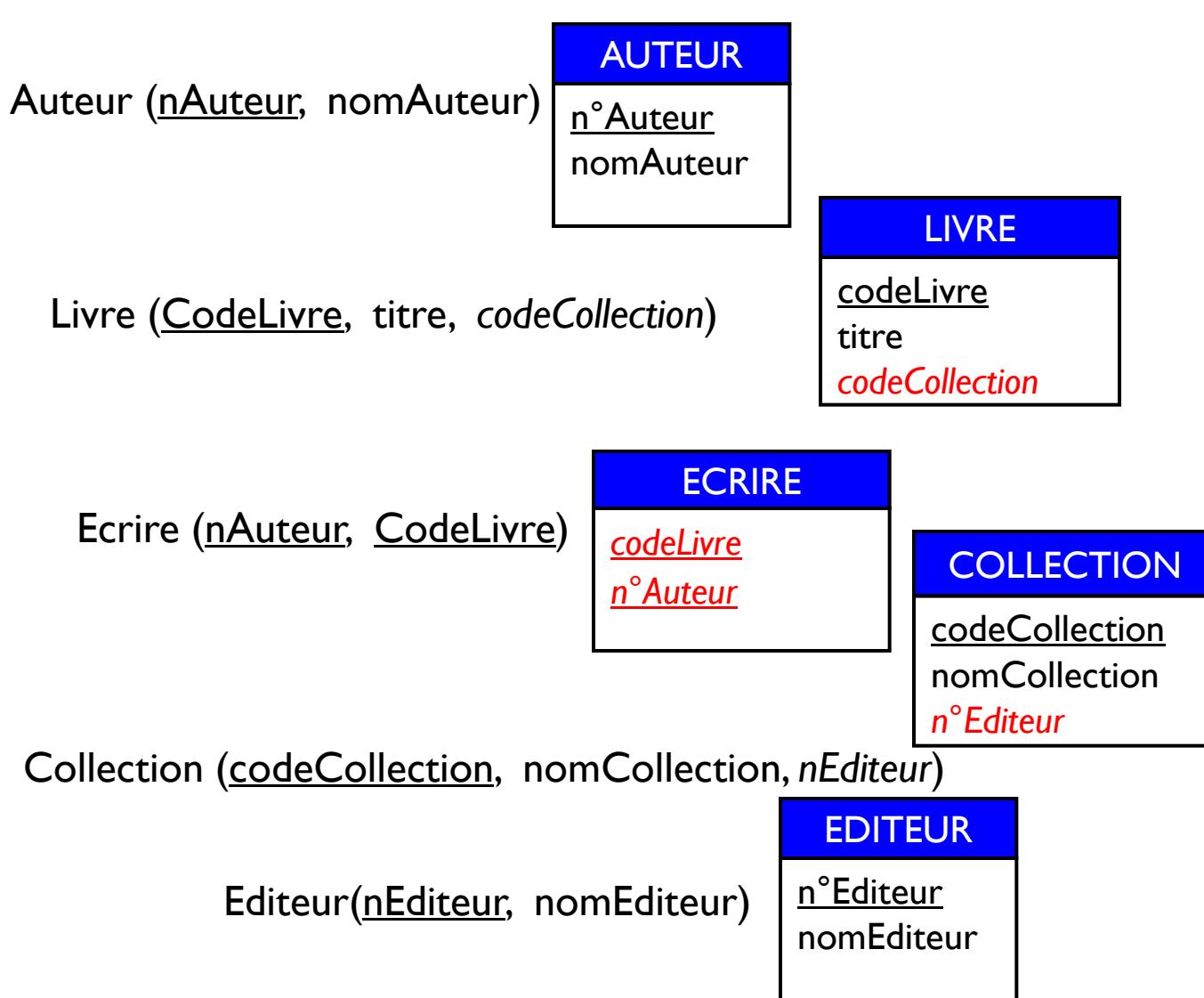
# Le modèle relationnel

- Intègre les notions suivantes :
  - Attribut (ou champ)
  - Domaine
  - Tuple (ou n-uplet, enregistrement)
  - Clé
  - Cardinalité
  - Degré

# Exemple (MLD- Graphique)



# Exemple (MLD Relationnel)





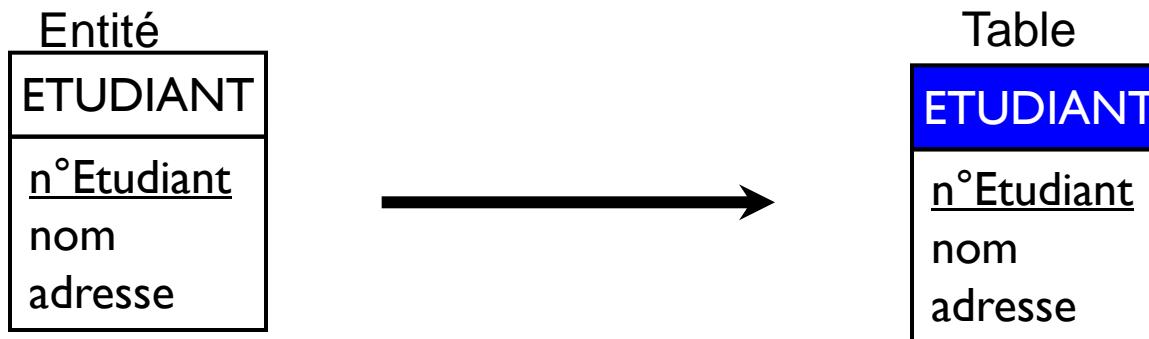
**PASSAGE MCD -> MLD**

# Passage MCD → MLD relationnel

- Est réalisé selon les règles suivantes
- Est entièrement algorithmique
  - Mais n'est pas totalement réversible

# Règle I - Entité

- Entité → Table
- Propriété → Attribut
- Identifiant → Clé primaire



ETUDIANT (n°Etudiant, nom, adresse)



# Associations

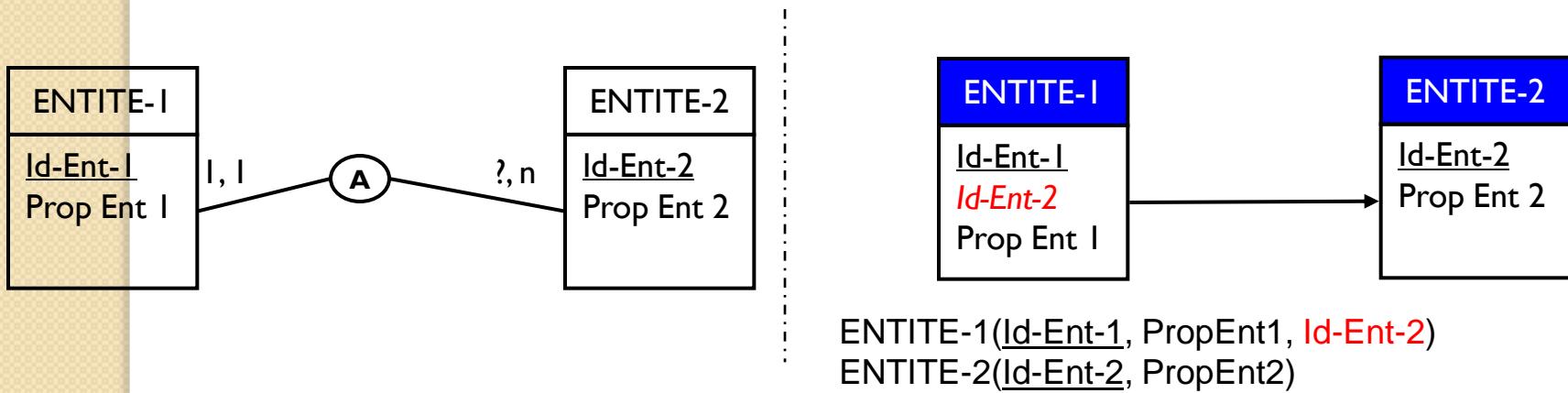
- Prendre en considération les cardinalités
  - $(*,n)$ - $(1,1)$
  - $(0,1)$ - $(1,1)$
  - $(1,1)$ - $(1,1)$
  - $(*,n)$ - $(0,1)$
  - $(0,1)$ - $(0,1)$
  - $(0,n)$ - $(0,n)$

## Règle 2 – Relation binaire $(?,*)-(1,1)$

- Elle est matérialisée par une flèche entre les deux entités concernées
- On duplique la clé de la table issue de l'entité à cardinalité « \* » dans la table issue de l'entité à cardinalité « 1 » où elle devient une **clé externe**
- On peut éventuellement **renommer** l'attribut dupliqué

# Règle 2 – Relation binaire $(?,*)-(1,1)$

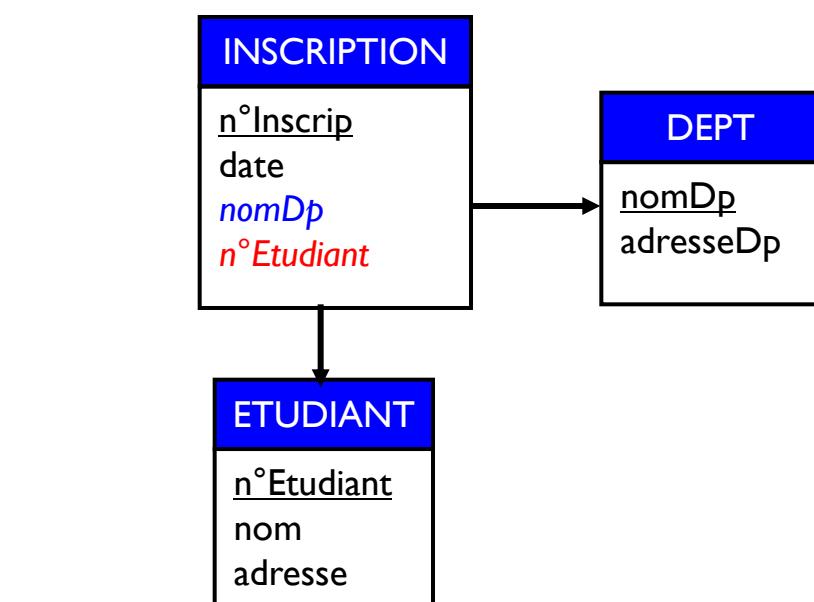
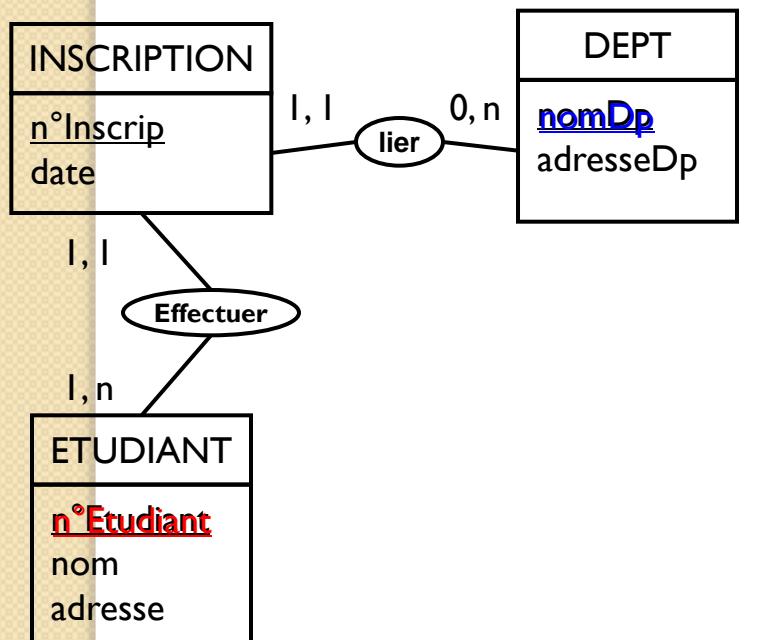
- Cas I – Relation binaire  $(*,n)-(1,1)$



# Règle 2 – Relation binaire $(?,*)-(I,I)$

- Cas I – Relation binaire  $(*,n)-(I,I)$

Exemple I (Clé simple)

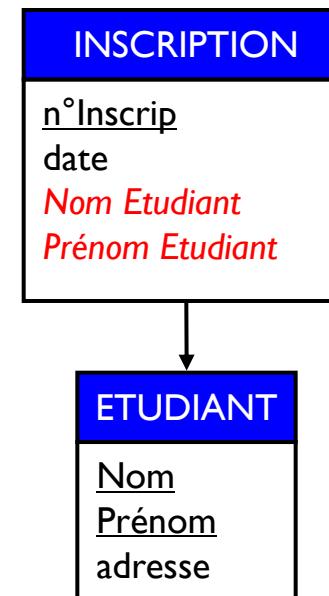
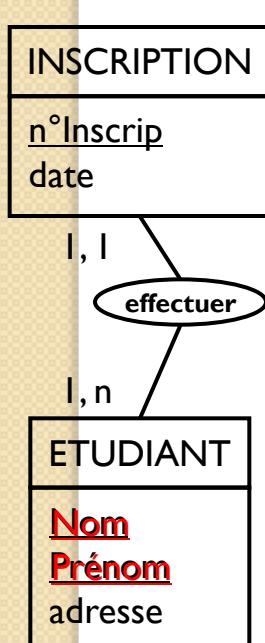


ETUDIANT(n°Etudiant, nom, adresse)  
DEPT(nomDp, adresseDp)  
INSCRIPTION (n°Inscrip, date, nomDp, n°Etudiant)

# Règle 2 – Relation binaire $(?,*)-(I,I)$

- Cas I – Relation binaire  $(*,n)-(I,I)$

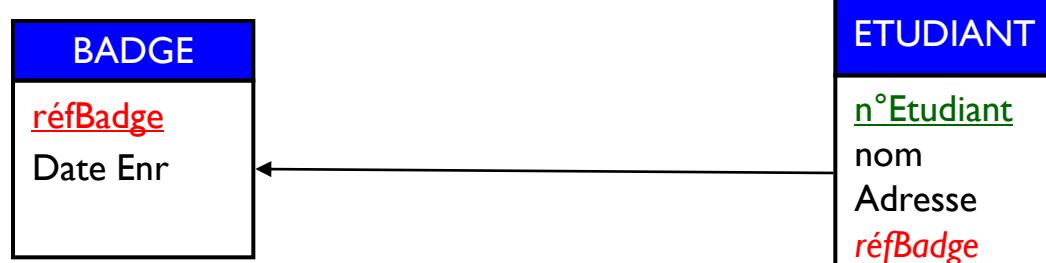
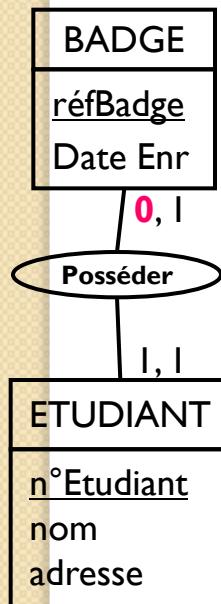
Exemple 2 (Clé composée et renommage)



ETUDIANT(Nom, Prénom, adresse)  
INSCRIPTION (n°Inscrip, date, *nom Etudiant*, *Prénom Etudiant*)

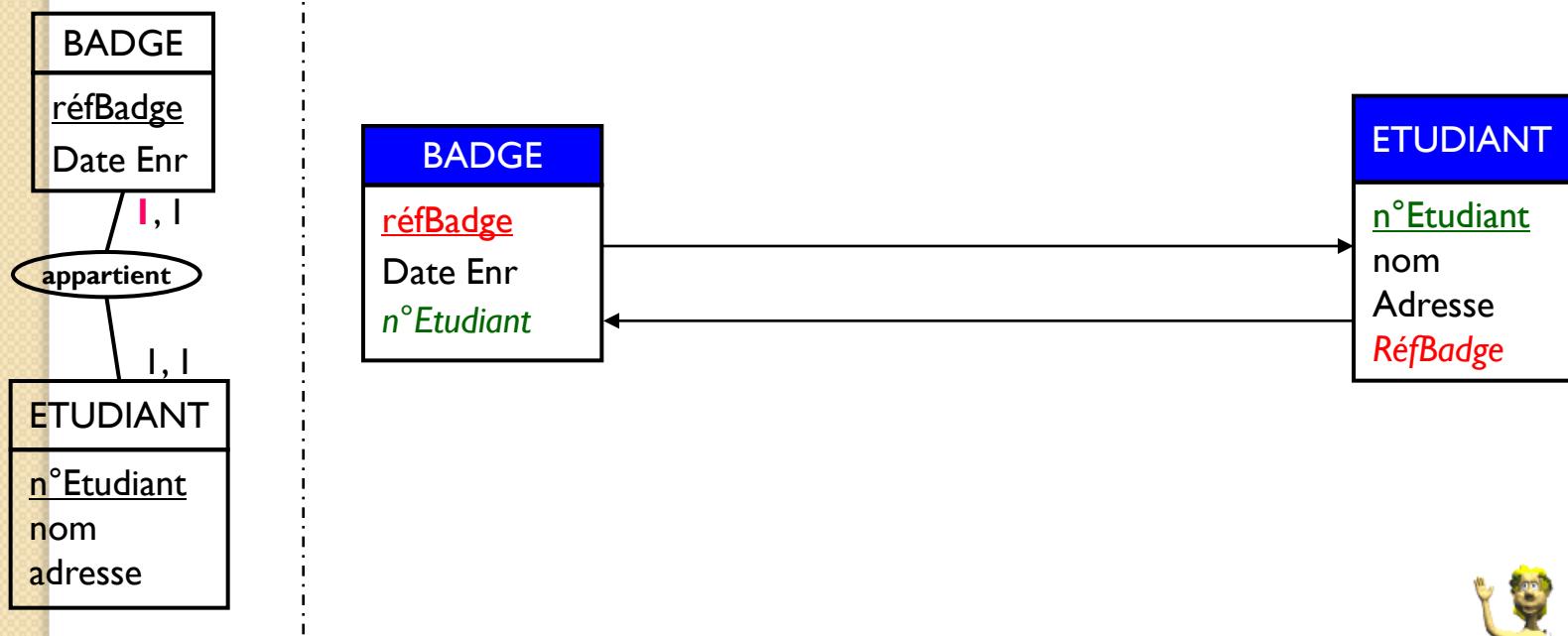
# Règle 2 – Relation binaire $(?,*)-(1,1)$

- Cas 2 – Relation binaire  $(0,1)-(1,1)$ 
  - Une Solution (Voir 2<sup>ème</sup> solution dans règle 3)



# Règle 2 – Relation binaire $(?,*)-(1,1)$

- Cas 3 – Relation binaire  $(1,1)-(1,1)$



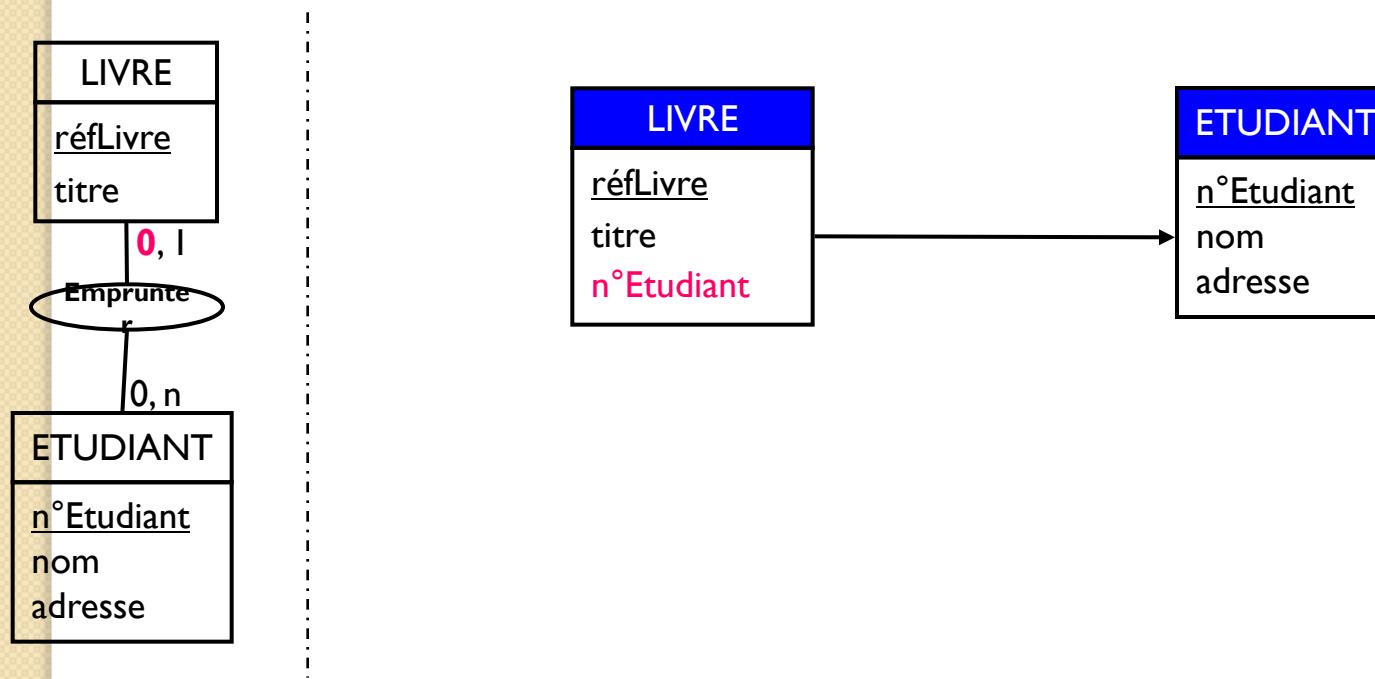
# Règle 3 – Relation binaire $(?,*)-(0,1)$

- 1<sup>ère</sup> solution
  - Appliquer la règle 2
  - A condition que le SGBD cible puisse accepter des **valeurs nulles (NULL) sur la clé étrangère**
- 2<sup>ème</sup> solution
  - Créer une nouvelle table avec
    - Identifiant de l'entité à cardinalité 0,1 comme clé primaire
    - Identifiant de l'autre entité comme clé étrangère



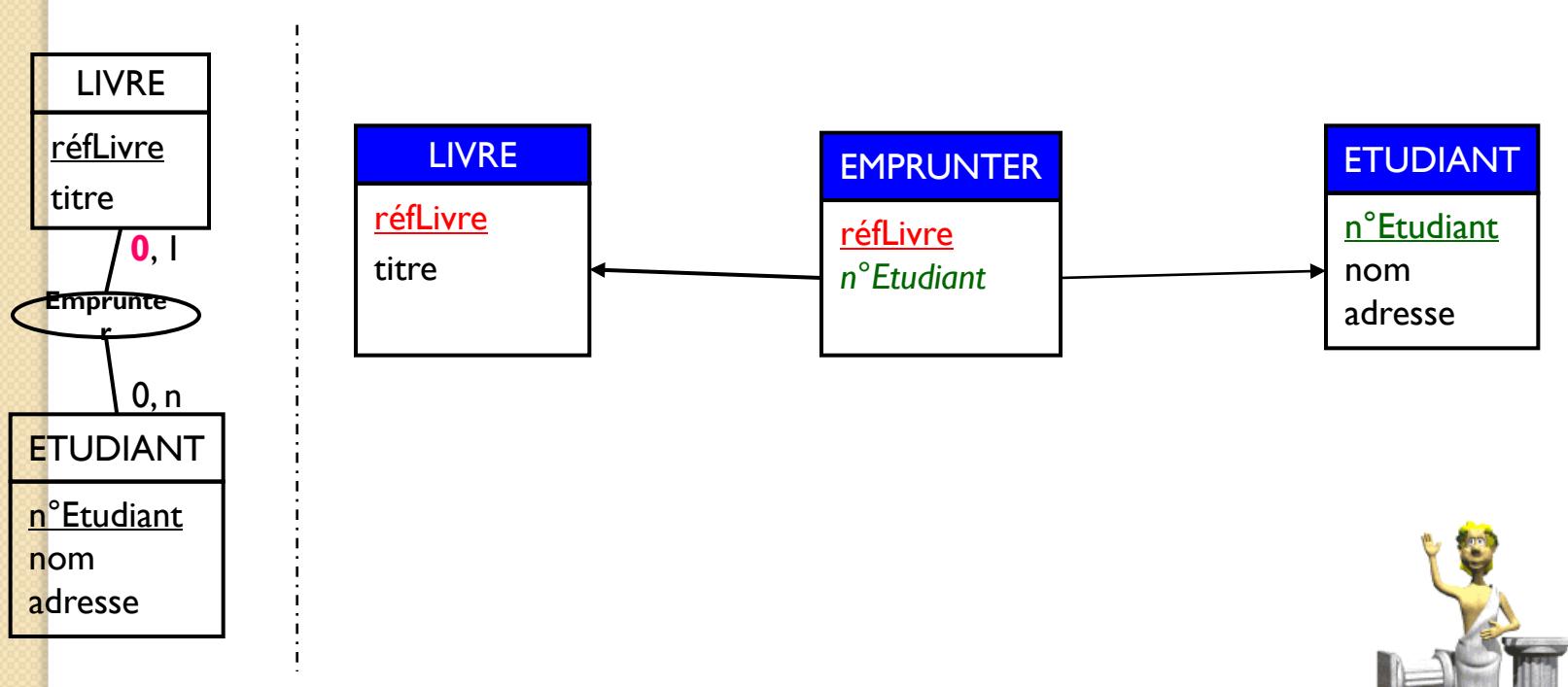
# Règle 3 – Relation binaire $(?,*)-(0,1)$

- Cas I – Relation binaire  $(*,n)-(0,1)$ 
  - 1<sup>ère</sup> solution avec la règle 2



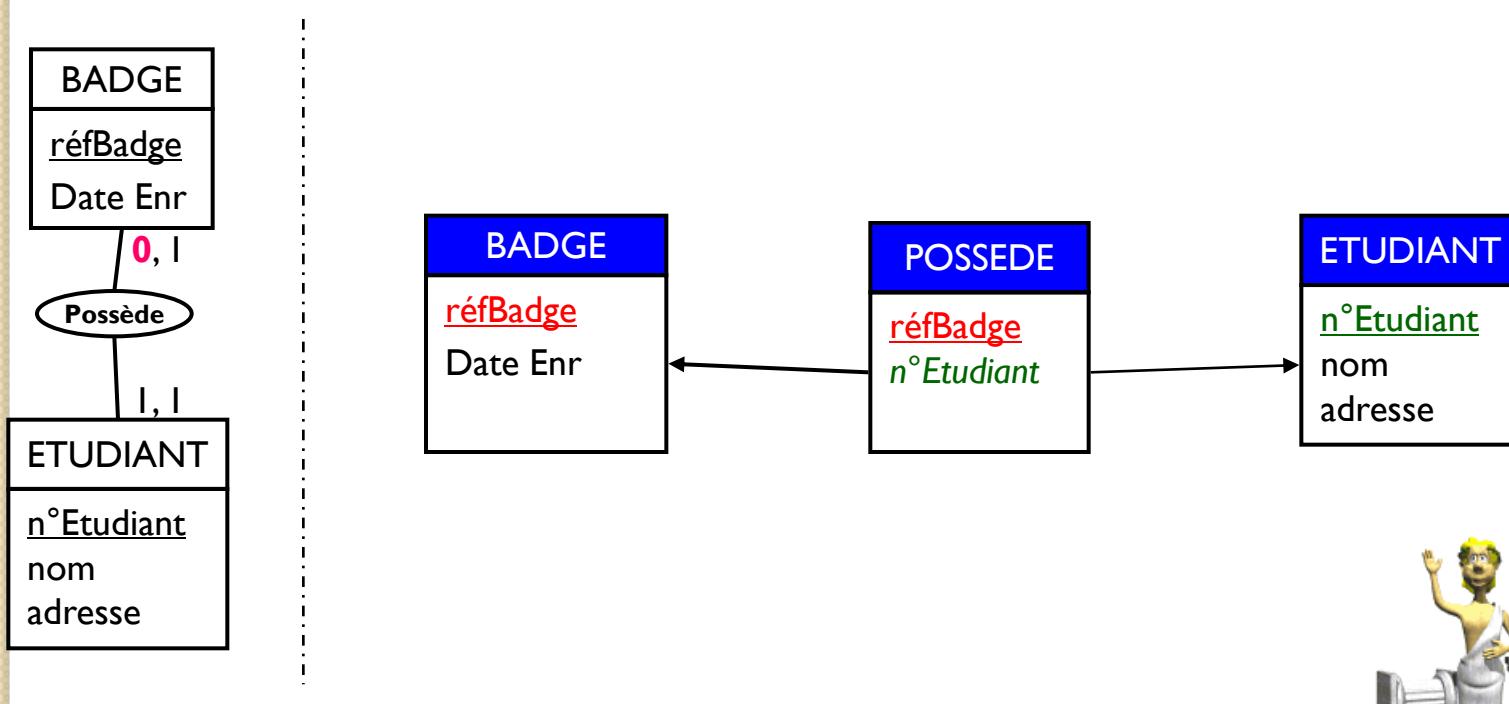
# Règle 3 – Relation binaire $(?,*)-(0,1)$

- Cas I – Relation binaire  $(*,n)-(0,1)$ 
  - 2<sup>ème</sup> solution



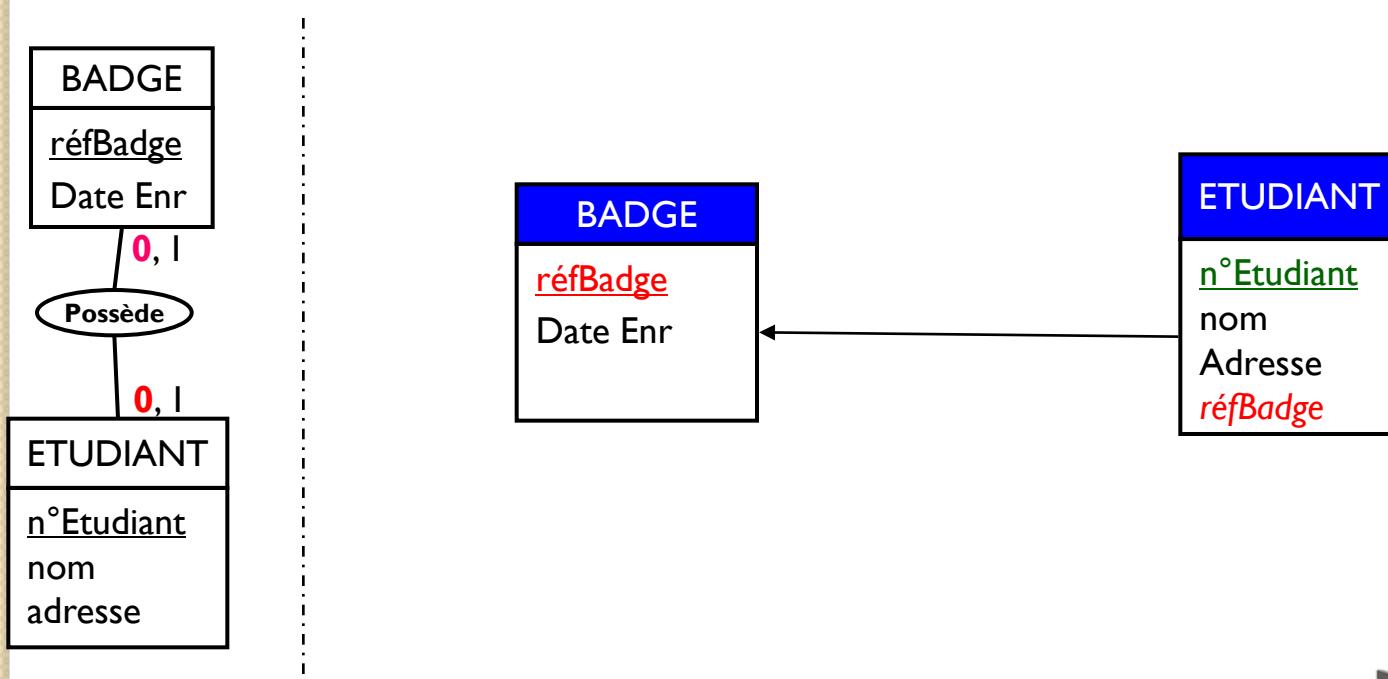
# Règle 3 – Relation binaire $(?,*)-(0,1)$

- Cas 2 – Relation binaire  $(1,1)-(0,1)$ 
  - (Cas 2 de la règle 2)



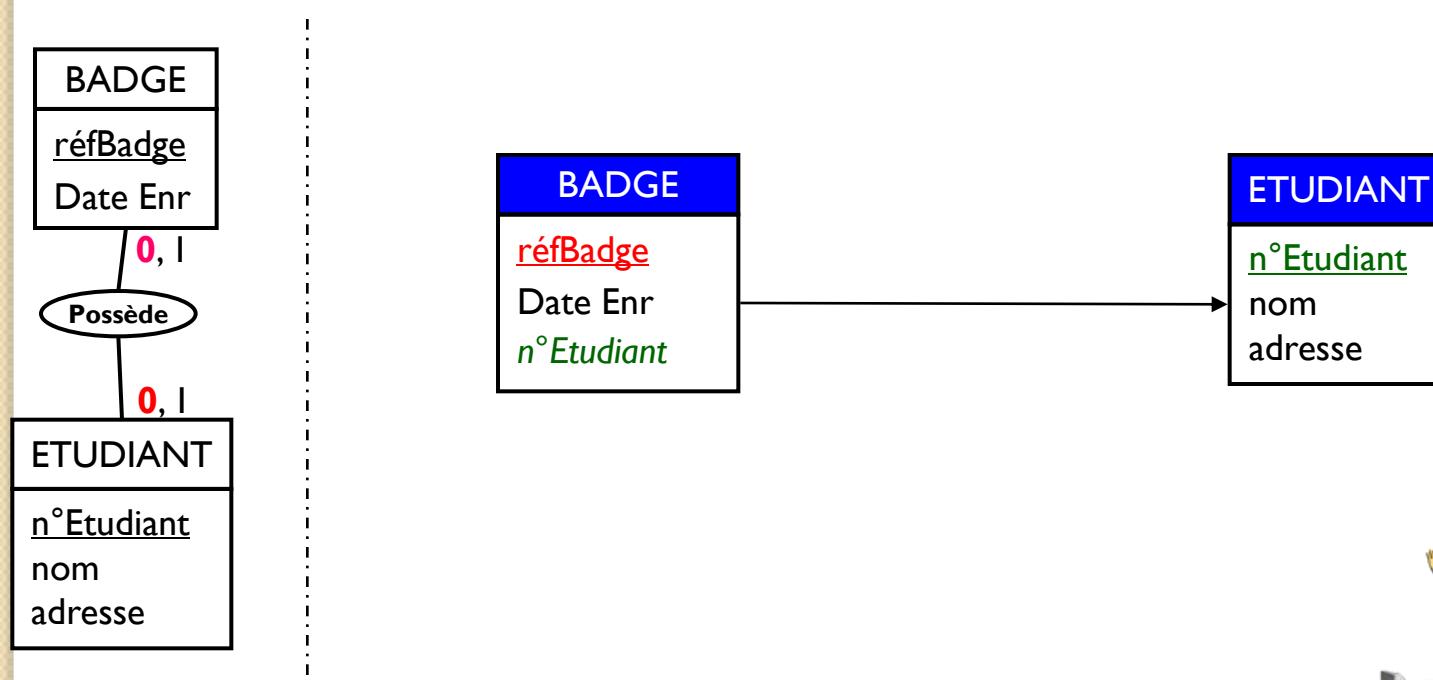
# Règle 3 – Relation binaire $(?,*)-(0,1)$

- Cas 3 – Relation binaire  $(0,1)-(0,1)$ 
  - Solution I



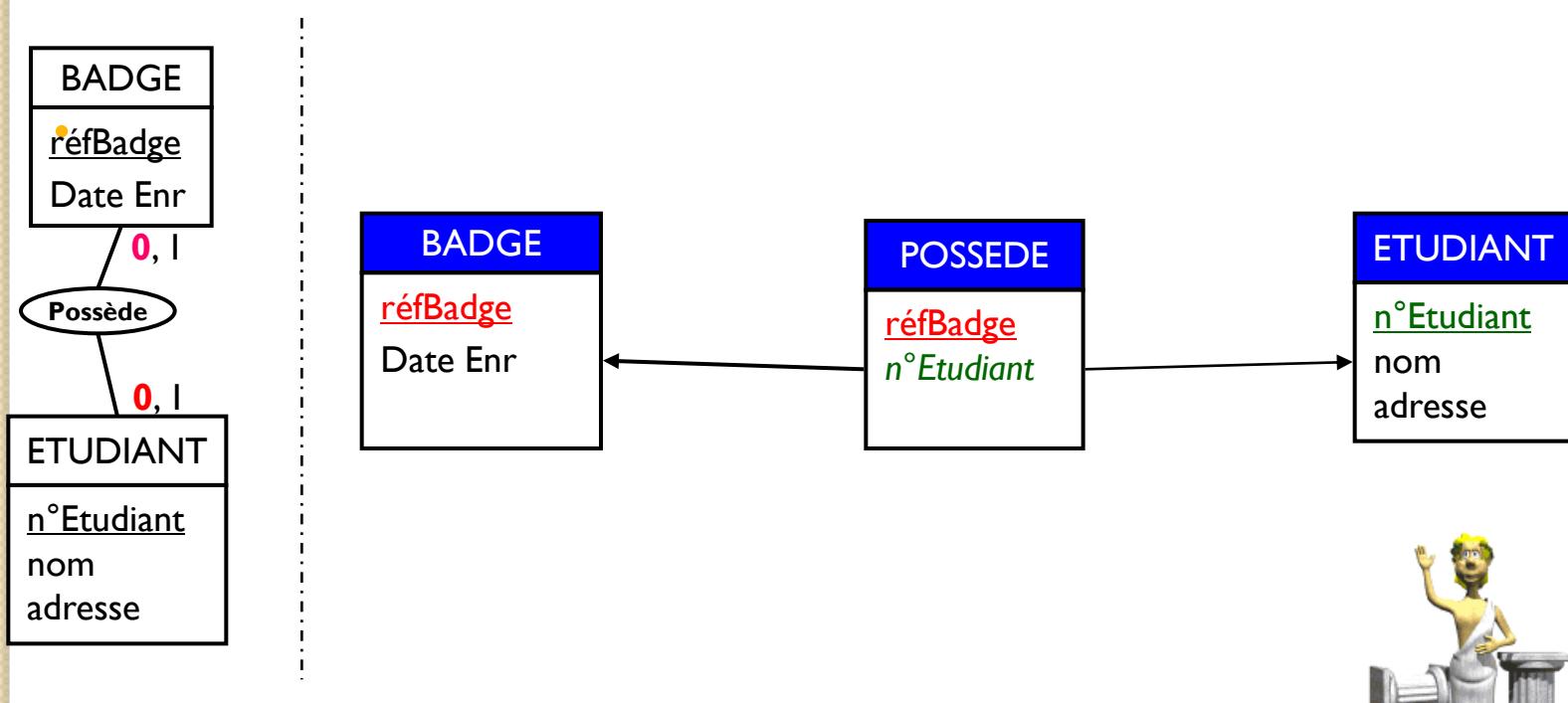
# Règle 3 – Relation binaire $(?,*)-(0,1)$

- Cas 3 – Relation binaire  $(0,1)-(0,1)$ 
  - Solution 2



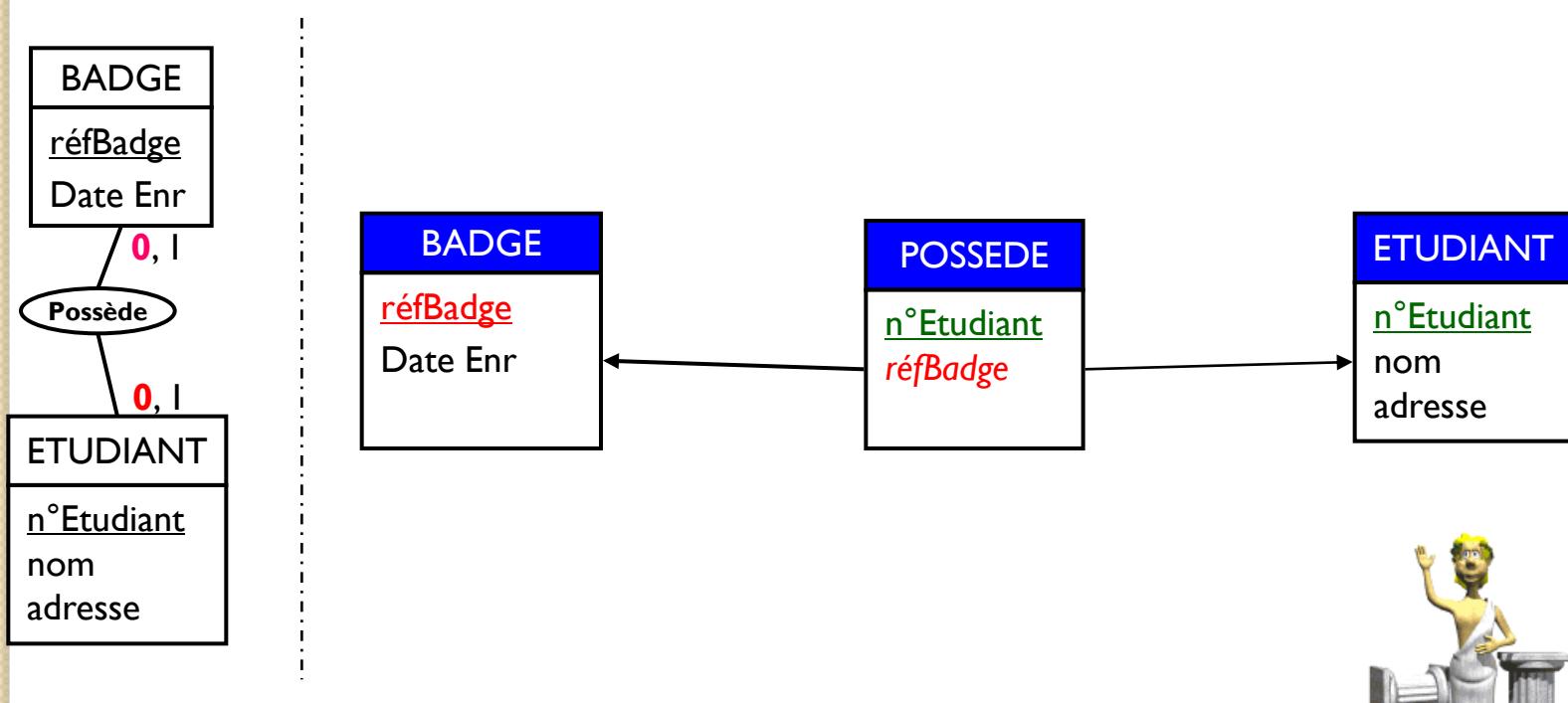
# Règle 3 – Relation binaire $(?,*)-(0,1)$

- Cas 3 – Relation binaire  $(0,1)-(0,1)$ 
  - Solution 3



# Règle 3 – Relation binaire $(?,*)-(0,1)$

- Cas 3 – Relation binaire  $(0,1)-(0,1)$ 
  - Solution 4



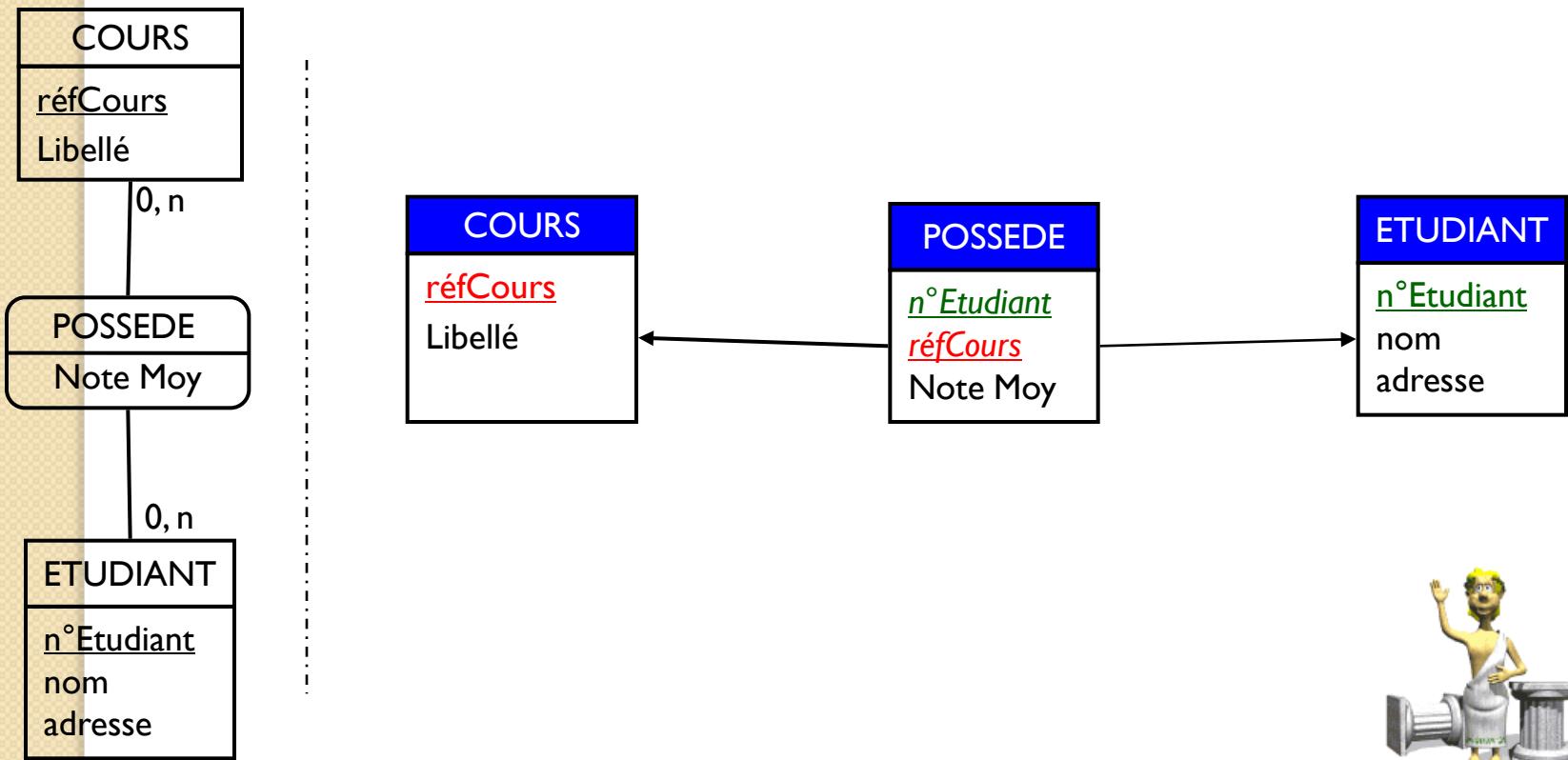
## Règle 4 – Relation binaire $(*,n)-(*,n)$

- Elle se transforme en table avec
  - Comme clé primaire : une clé composée des identifiants des entités participant à la relation. Cette clé composée est également clé externe
  - Comme champs: les propriétés portées par la relation



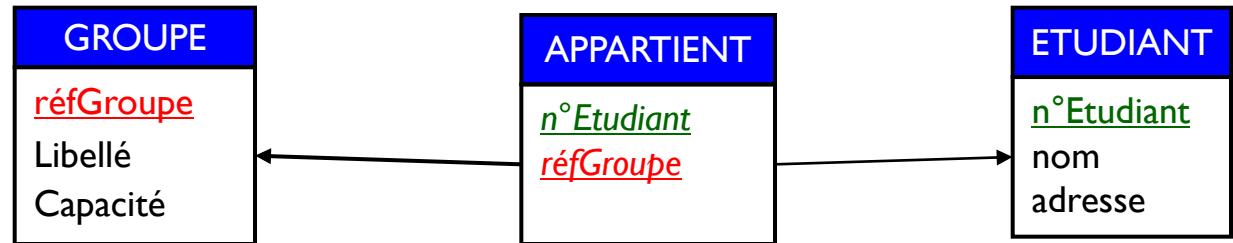
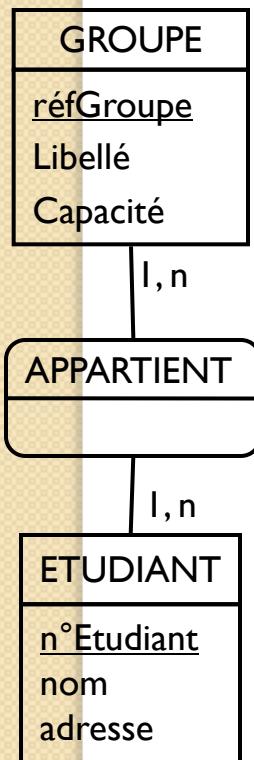
# Règle 4 – Relation binaire (\*,n)-(\*,n)

- Cas 3 – Relation binaire (0,n)-(0,n)



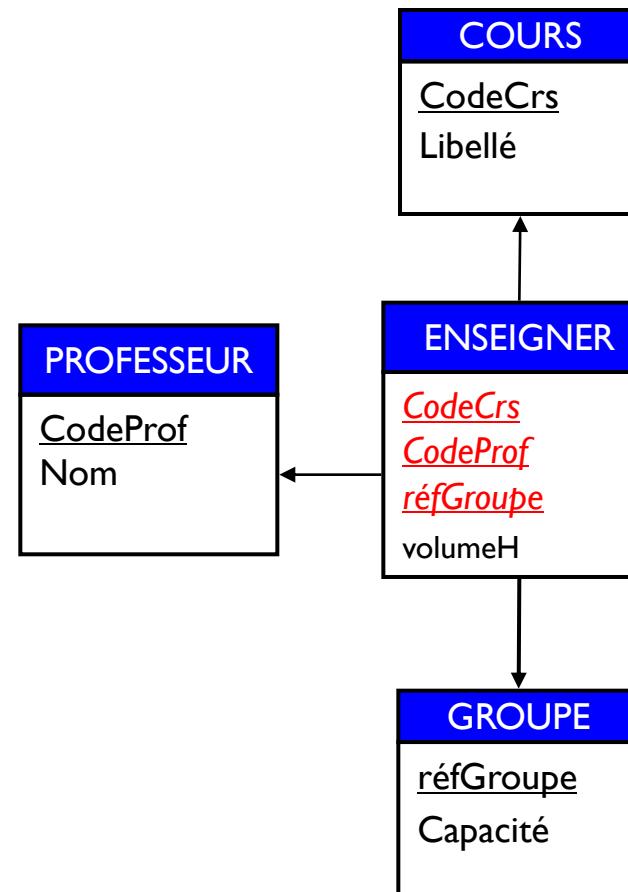
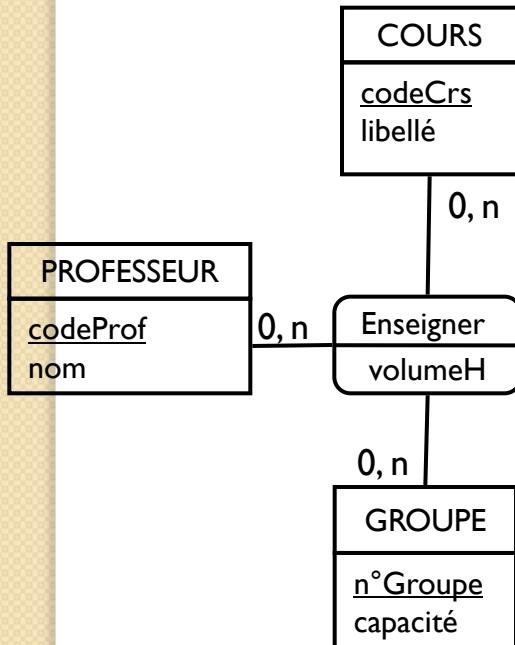
# Règle 4 – Relation binaire (\*,n)-(\*,n)

- Cas 3 – Relation binaire (1,n)-(1,n)



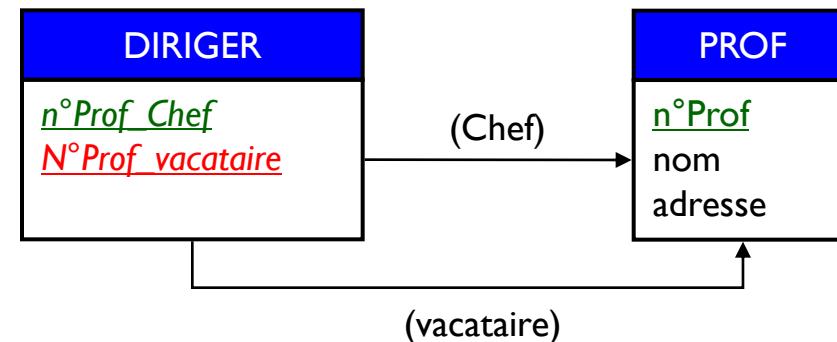
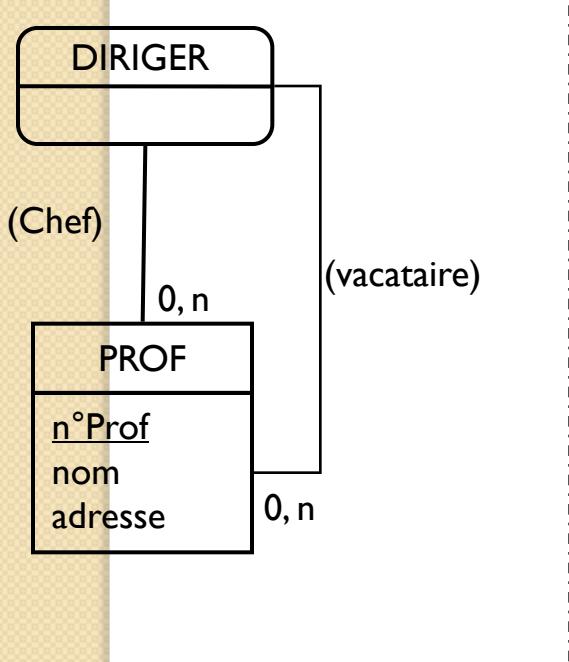
# Règle 5 – Relation ternaire ou +

- Cas I – Relation ternaire



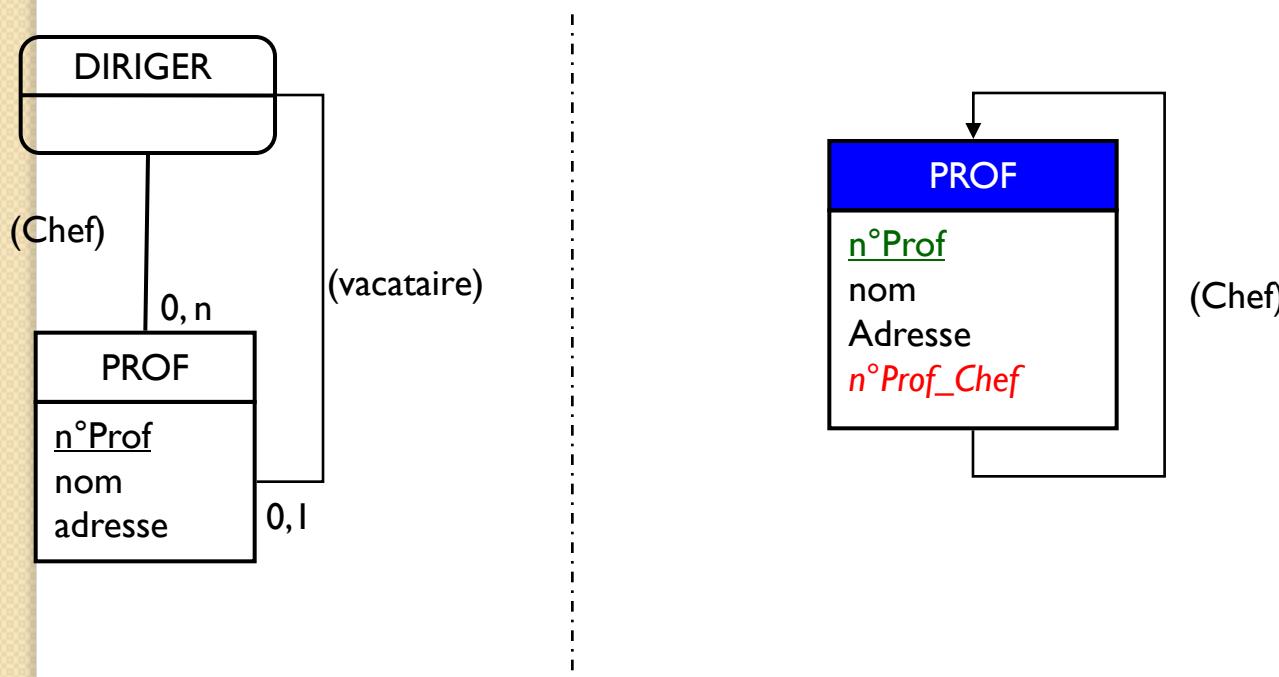
# Relation réflexive

- Cas I – Relation binaire (0,n)-(0,n)
  - Un vacataire est dirigé par plusieurs chefs



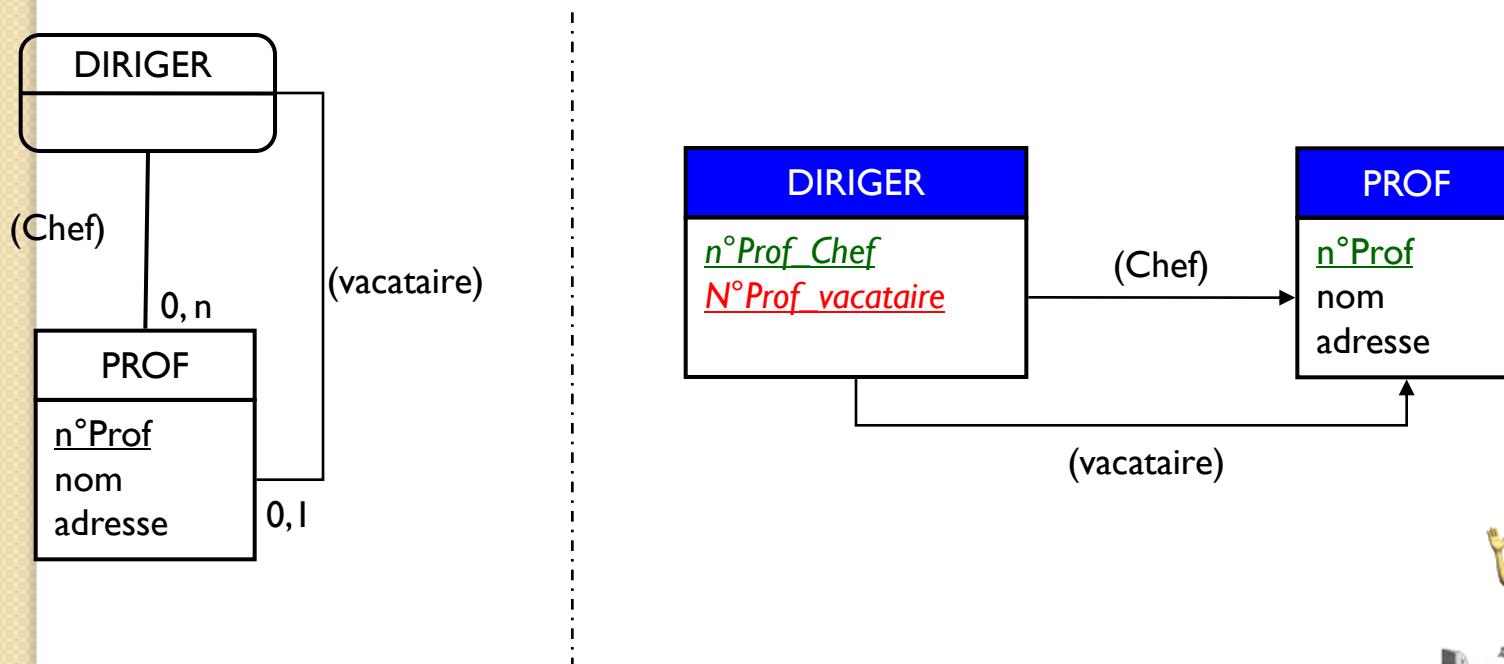
# Relation réflexive

- Cas 2 – Relation binaire  $(0,n)-(0,1)$ 
  - Un vacataire peut être dirigé par au max 1 chef



# Relation réflexive

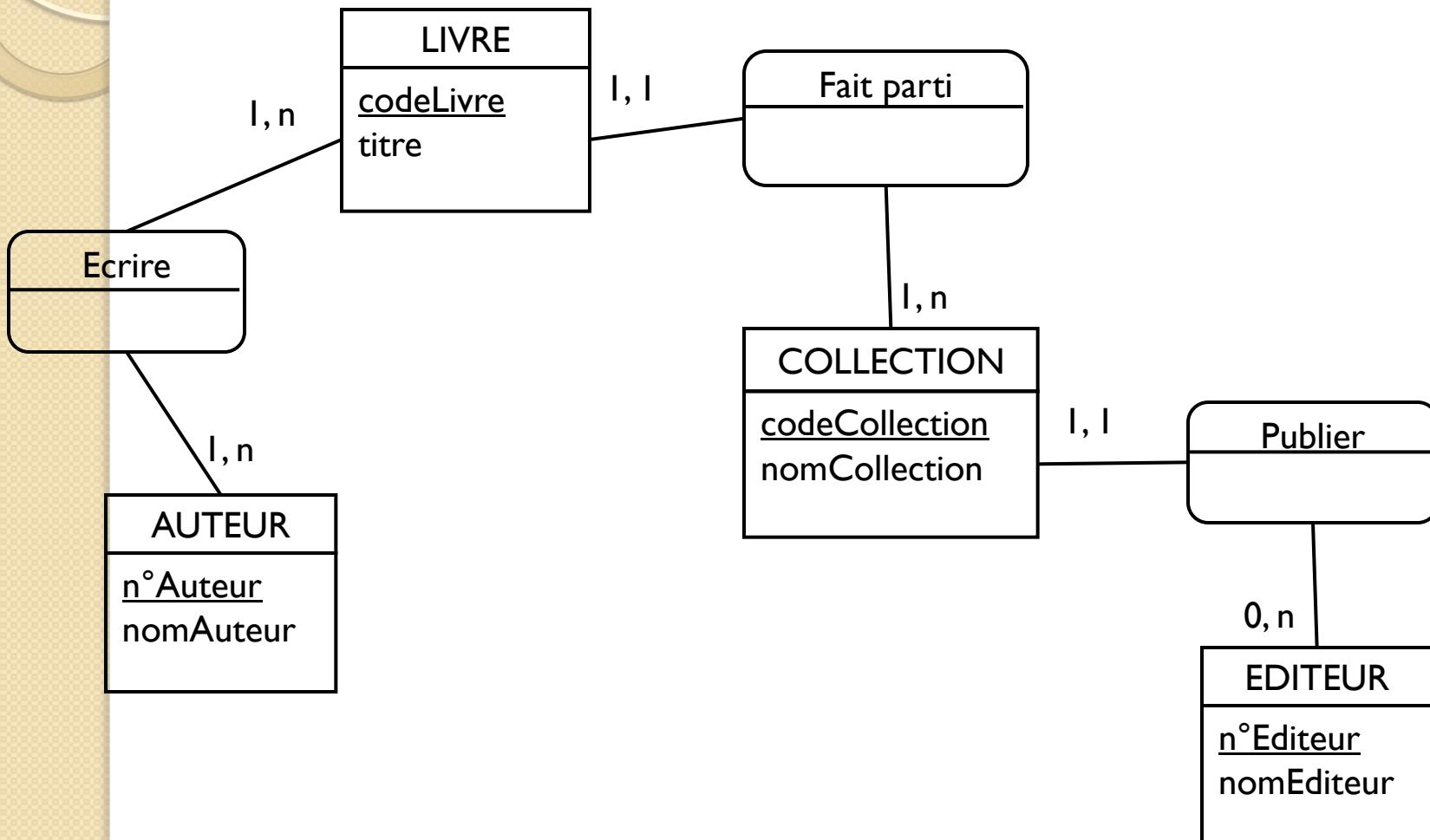
- Cas 2 – Relation binaire  $(0,n)-(0,1)$ 
  - Un vacataire peut être dirigé par au max 1 chef



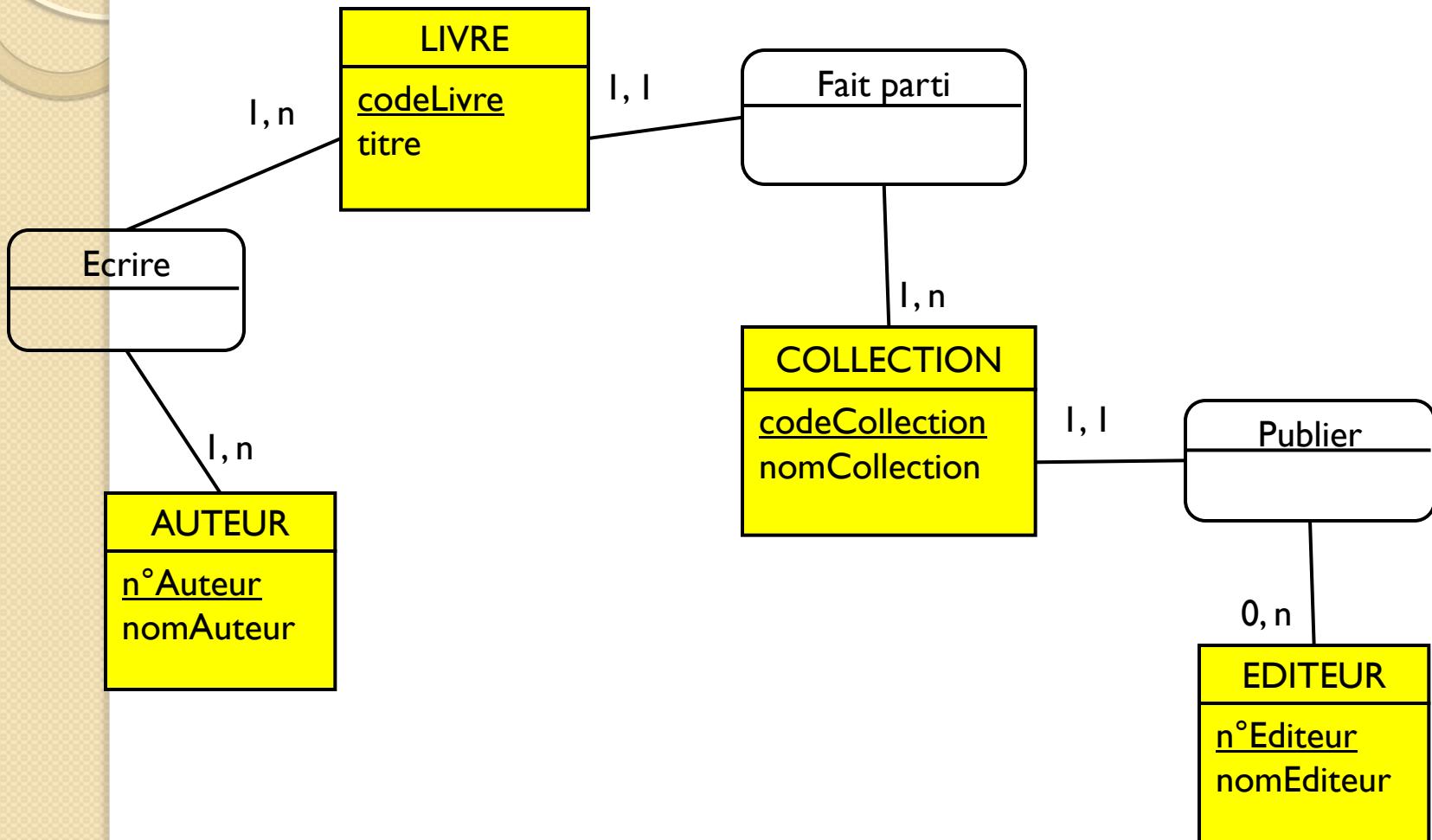


# **ETUDE DE CAS**

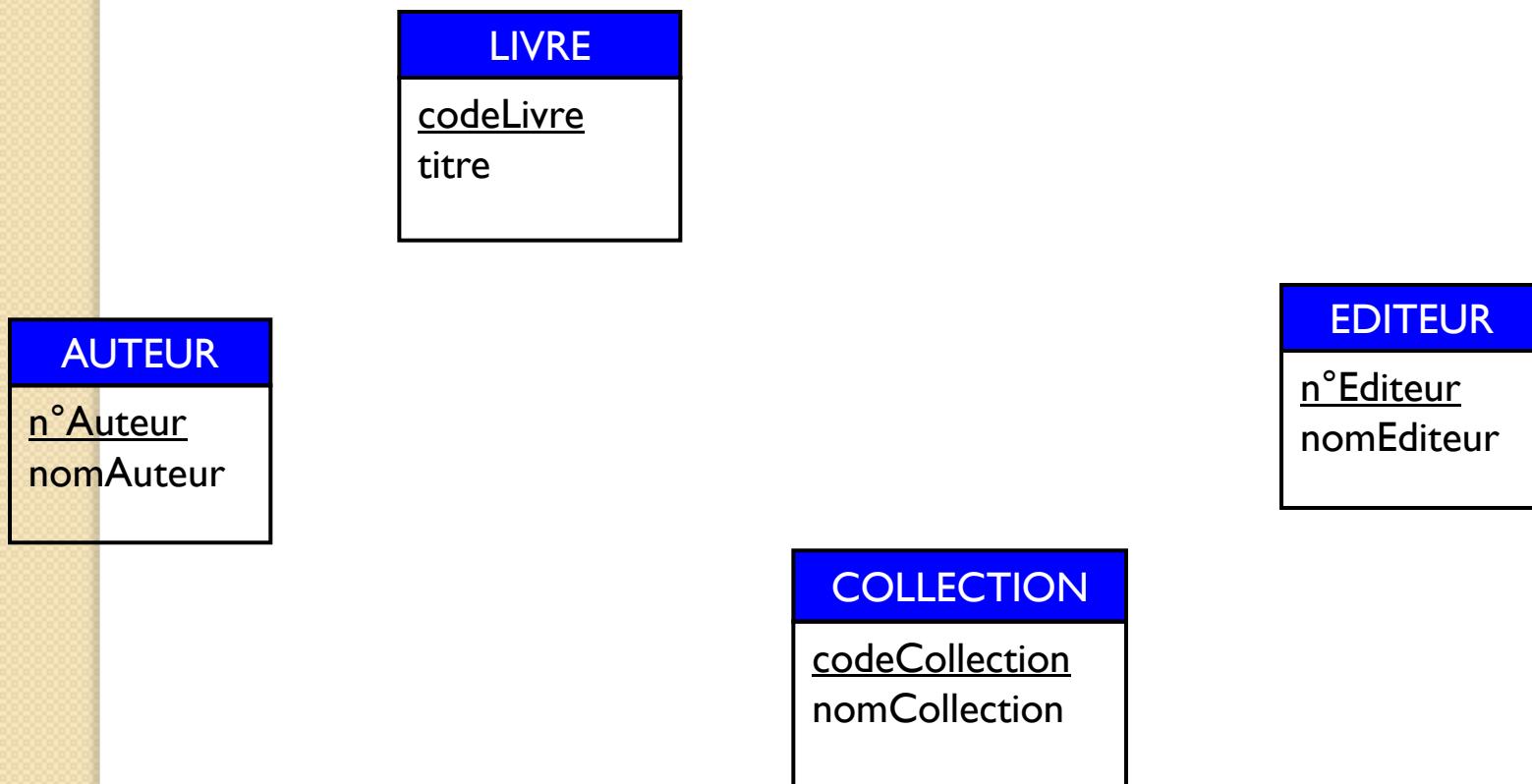
# Exemple (MCD)



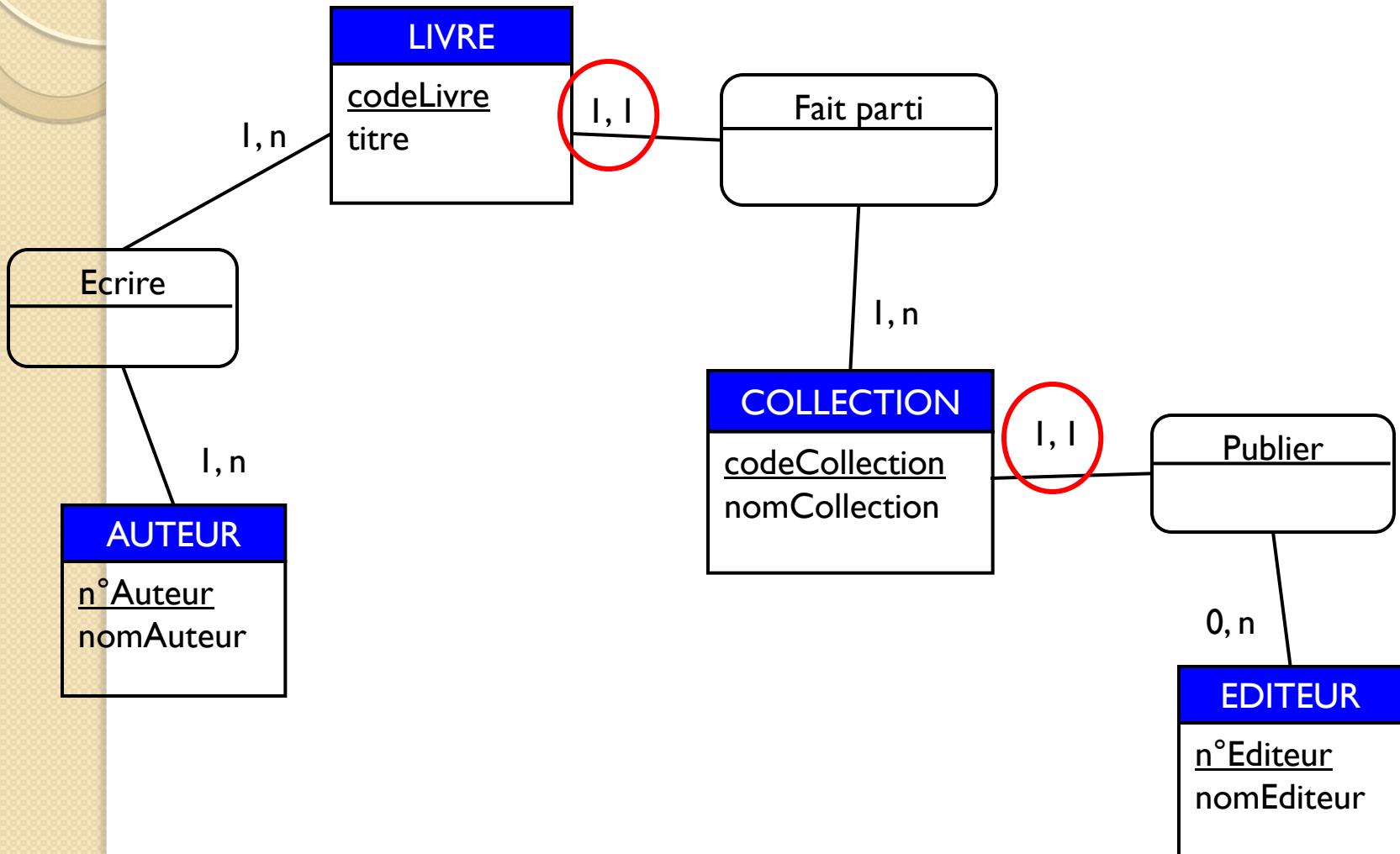
# Exemple



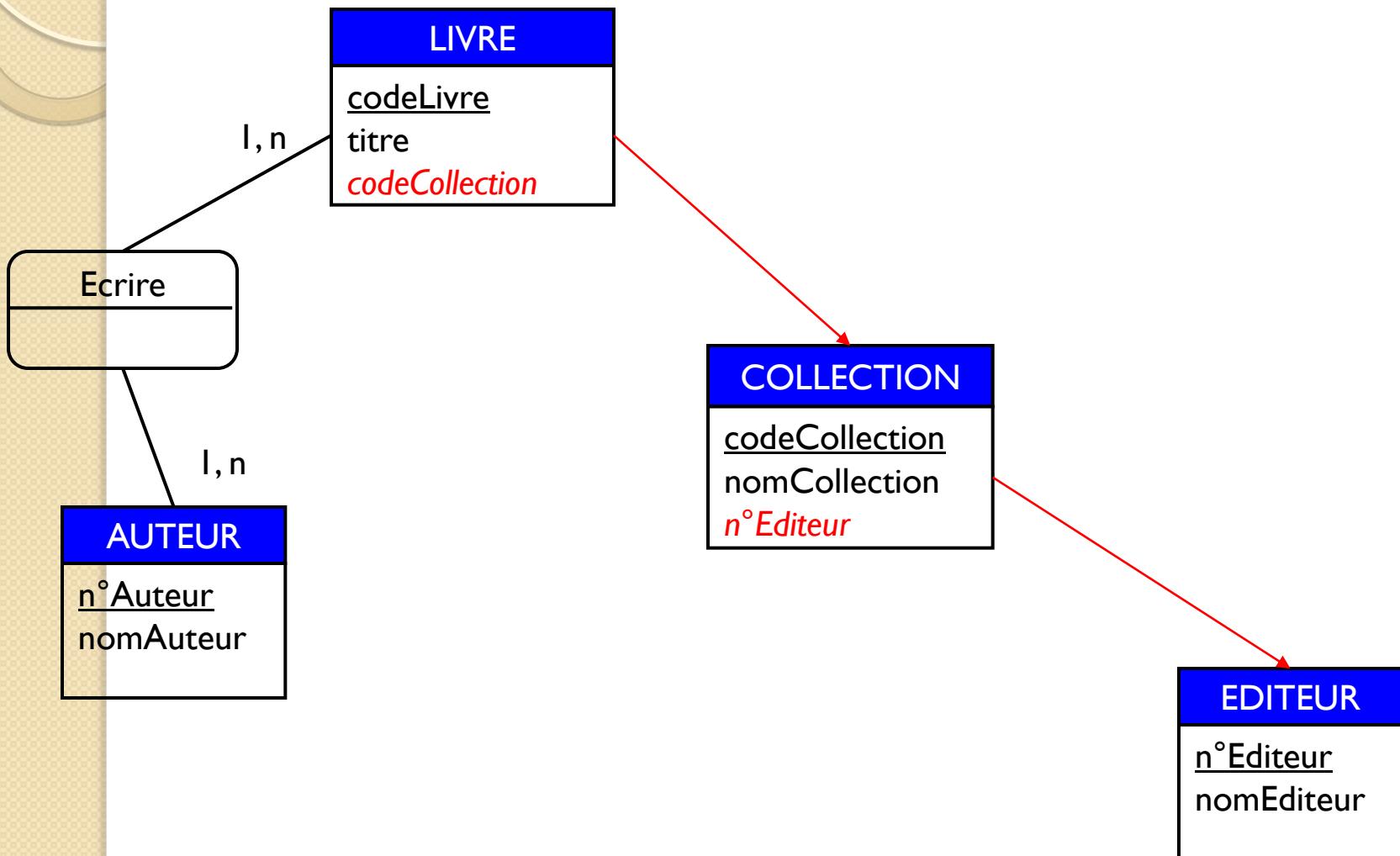
# Exemple



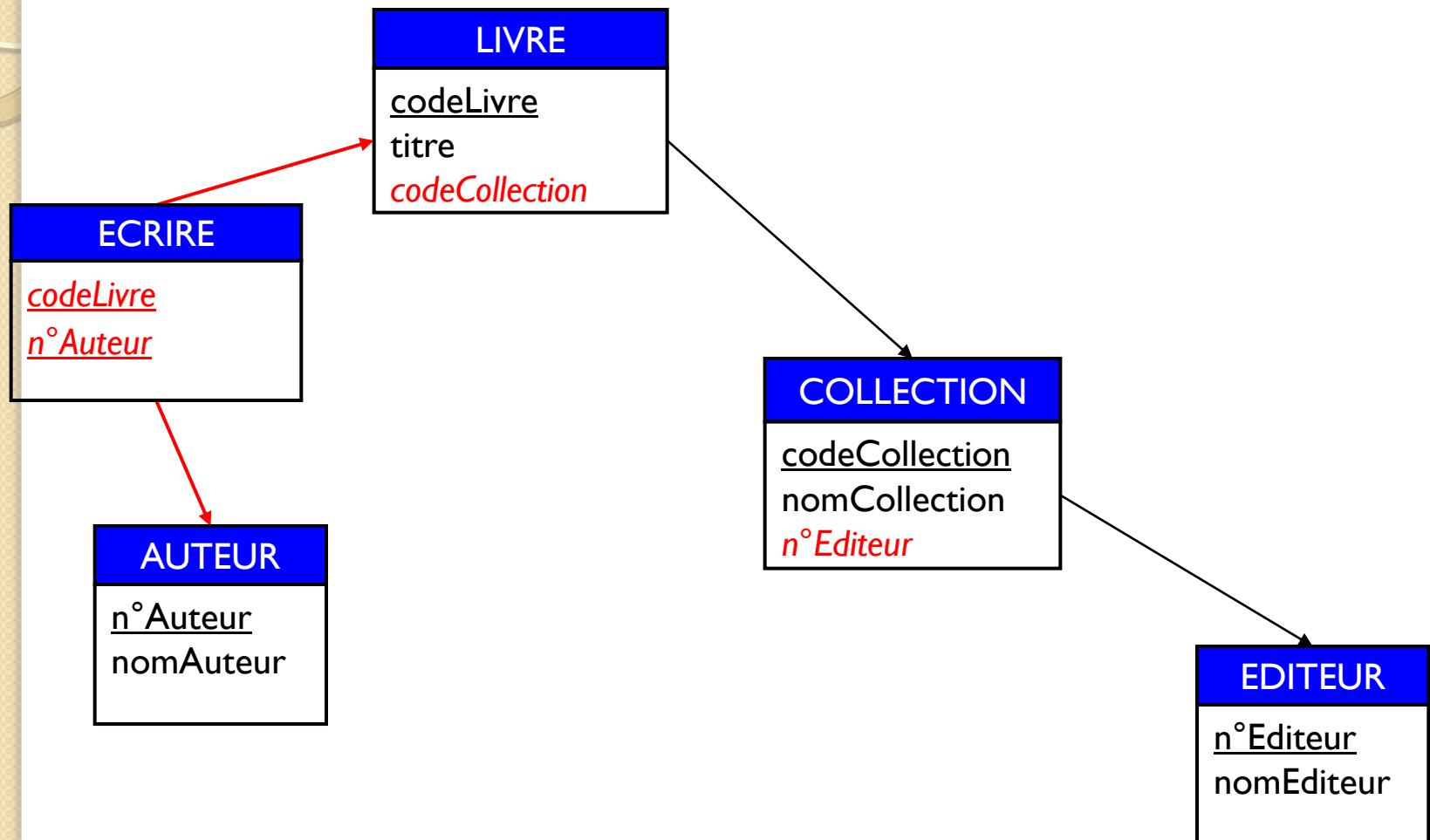
# Exemple



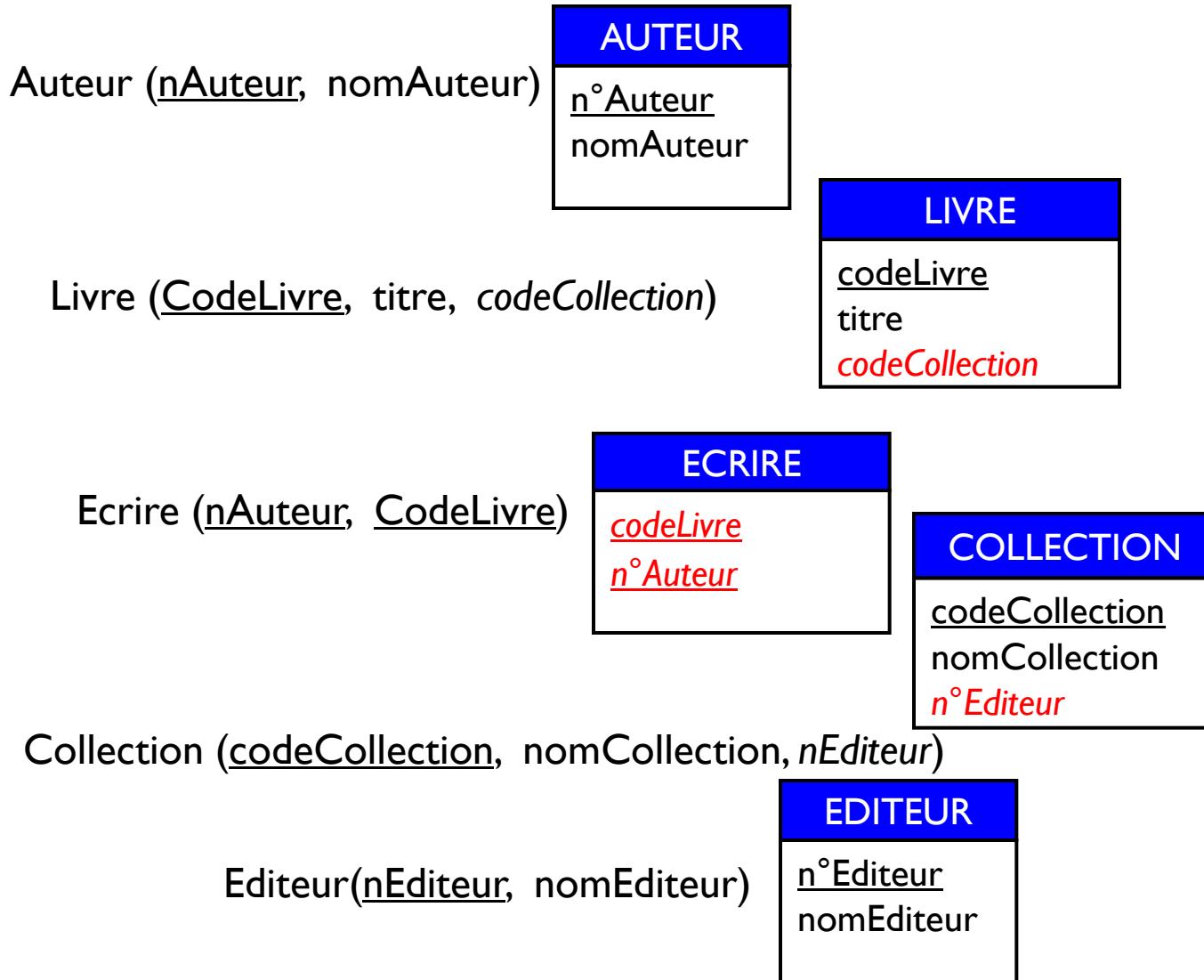
# Exemple



# Exemple (MLD- Graphique)



# Exemple (MLD Relationnel)



# SQL

```
CREATE TABLE Auteur (nAuteur INT NOT NULL AUTO_INCREMENT,  
nomAuteur VARCHAR(80),  
PRIMARY KEY(nAuteur));
```

```
CREATE TABLE Livre(codeLivre INT NOT NULL AUTO_INCREMENT,  
titre VARCHAR(100),  
codeCollection INT,  
PRIMARY KEY(codeLivre), FOREIGN KEY(codeCollection) REFERENCES  
Collection(codeCollection));
```

```
CREATE TABLE Collection (codeCollection INT NOT NULL  
AUTO_INCREMENT, nomCollection VARCHAR(100), PRIMARY KEY  
(codeCollection), etc.)
```



# **THÉORIE DE LA NORMALISATION**

# Introduction

## La théorie de la normalisation

Cette théorie est basée sur les "dépendances fonctionnelles" (DF). Les dépendances fonctionnelles traduisent des contraintes sur les données (par exemple, on décide que deux individus différents peuvent avoir même nom et prénom mais jamais le même numéro NN). Ces contraintes représentent la réalité et imposent des limites à la base. Les dépendances fonctionnelles et des propriétés particulières, définissent une suite de formes normales (FN). Elles permettent de décomposer l'ensemble des informations en diverses relations.

# Les dépendances fonctionnelles

- Définition : On dit qu'un attribut(ou ens d'attributs) Y dépend fonctionnellement d'un attribut( ou ens) X si, étant donné une valeur de X, il lui correspond une unique valeur de Y (quel que soit l'instance t considéré). On dit aussi que X détermine Y
- Soit  $U = \{A_1, A_2, \dots, A_n\}$  l'ensemble des n attributs de la relation R. On considère que la base de données est constituée de la seule relation R, dite relation universelle.
- Une dépendance fonctionnelle notée  $X \rightarrow Y$ , entre deux ensembles d'attributs X et Y ( $X \subset U$  et  $Y \subset U$ ), spécifie une contrainte entre deux n-uplets potentiels de R.  
Cette contrainte est :
  - $t_1, t_2 \in R, t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$

# Dépendances Fonctionnelles

même entité

Numéro INSEE



Nom

Prénom

ville

Immatriculation



Nom propriétaire

Prénom

entités différentes

# Dépendances Fonctionnelles

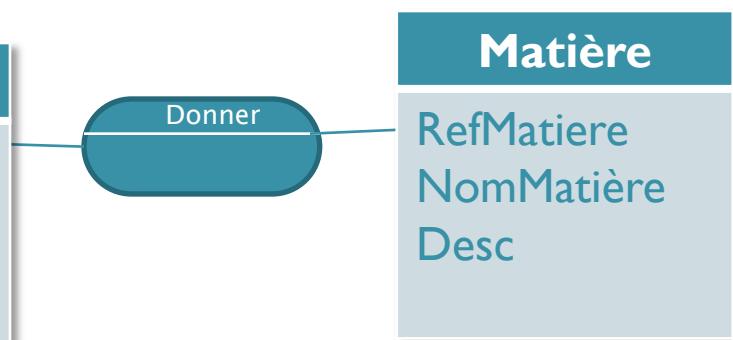
## Objectifs

Caractériser des entités pouvant être décomposées sans pertes d'Informations

Enseignant
RefEns
Nom
Prénom
<b>Matière</b>



Enseignant
RefEns
Nom
Prénom





# **LES FORMES NORMALES**

# Normalisation

- **Objectifs**

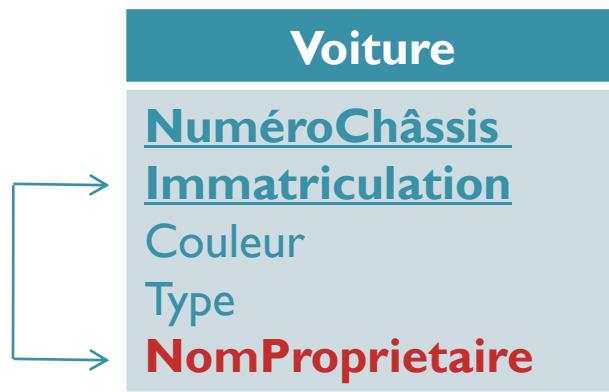
- Représenter des entités et des associations canoniques du monde réel
- Pouvoir décomposer les entités en se basant sur les dépendances fonctionnelles sans pertes d'informations et éliminant la redondance de données

# Première forme normale (IFN)

- Une entité est en première forme normale si et seulement si :
  - Toutes ses propriétés (attributs) sont élémentaires
  - Elle possède un identifiant

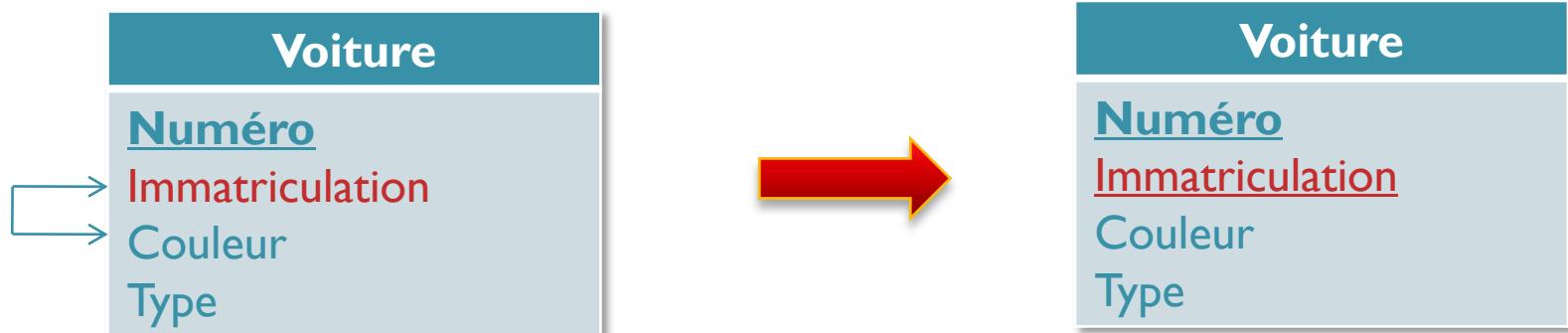
# Deuxième Forme Normale (2 FN)

- Une entité est en deuxième forme normale si et seulement si :
  - Elle est en première forme normale
  - Toute propriété n'appartenant pas à l'identifiant; ne dépend pas d'une partie de l'identifiant



# Troisième Forme Normale (3 FN)

- Une entité est en troisième forme normale si et seulement si :
  - Elle est en deuxième forme normale
  - Toute propriété n'appartenant pas à l'Identifiant ne dépend pas d'une autre propriété



**MCD doit être élaboré sous la troisième forme normale**



# **STRUCTURED QUERY LANGUAGE (SQL)**

# SQL

- Il a été standardisé par l'ANSI/ISO depuis 1986 Il a été intégré à Access, SQL Server, SQL/DS, DB2, puis ORACLE, INGRES, ...
- Il existe trois versions normalisées, du simple au complexe :
  - SQLI 86 : version minimale
  - SQLI 89 : ajout des contraintes de références (PRIMARY KEY, FOREIGN KEY)
  - SQL2 (92): langage complet à 3 niveaux
  - SQL 3 : les notions d'objets (TRIGGER, ROLE).
- La plupart des systèmes supportent SQLI complet

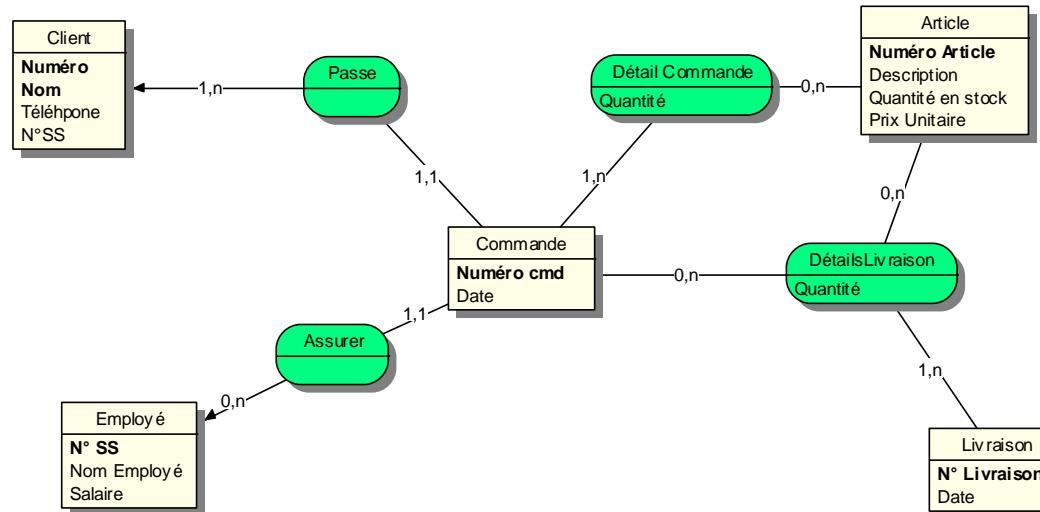
# SQL

## **SQL1 complet est composé de**

<b>SELECT INSERT UPDATE DELETE</b>	<b>Langage de manipulation des données (LMD)</b>
<b>CREATE ALTER DROP ...</b>	Langage de définition des données (LDD)
<b>GRANT REVOKE</b>	Langage de contrôle des données (LCD)

# Schéma utilisé

## Modèle Conceptuel de Données



## Modèle Logique de Données

Client (noClient, Nom, Téléphone, N°SS)

Article (N° Article, Description, P.U., Quantité en stock)

Commande (N° Com, Date, N° Client, N°SS\_Employé)

Détail commande (N° Com, N° Article, Quantité)

Employé (N°SS, Nom\_Employé, Salaire)

Livraison (N°Livraison, Date)

DétailsLivraison (N°Livraison, N°Cmd, N°Art, Quantité)

# Requêtes SQL simples

- **SELECT**

- La syntaxe de base est :

```
SELECT [ALL | DISTINCT] <clause de sélection1> [<clause de sélection2>...]  
FROM <Nom Table1> [ <Nom Table2> ...]  
[WHERE <condition1> [ [AND |OR] <condition2>....] ]
```

# Requêtes SQL simples

- **SELECT**

- La syntaxe de base est :

```
SELECT [ALL | DISTINCT] <clause de sélection1> [<clause de sélection2>...]  
FROM <Nom Table1> [ <Nom Table2> ...]  
[WHERE <condition1> [ [AND |OR] <condition2>....] ]
```

la clause de sélection = \* | nom\_table.\* | nom\_colonne, nom\_table.nom\_colonne

# Requêtes SQL simples

- **SELECT**

- La syntaxe de base est :

```
SELECT [ALL | DISTINCT] <clause de sélection1> [<clause de sélection2>...]  
FROM <Nom Table1> [ <Nom Table2> ...]  
[WHERE <condition1> [ [AND |OR] <condition2>....] ]
```

la clause where est facultative. Lorsqu'elle ne figure pas, il s'agit d'effectuer une projection simple

la condition est un booléen qui est vérifié pour les n-uplets retournés

# Requêtes SQL simples

- **SELECT**

- La syntaxe de base est :

```
SELECT [ALL | DISTINCT] <clause de sélection1> [<clause de sélection2>...]  
FROM <Nom Table1> [ <Nom Table2> ...]  
[WHERE <condition1> [ [AND |OR] <condition2>....] ]
```

Condition =

- | {expression {=|<|=|<=|=|>} expression}
- | expression BETWEEN expression AND expression
- | expression {IS NULL |IS NOT NULL}
- | expression {IN |NOT IN} listeConstantes
- | expression {LIKE |NOT LIKE} patron}

**AND est plus fort que OR**

On peut utiliser les parenthèses pour définir la priorité de résolution des conditions

# Requêtes SQL simples

- Afficher une table
  - Afficher toutes les *Commandes*

```
SELECT      *
FROM        Commande
```

Table <i>Commande</i>			
Nº Com	Date	Nº Client	NºSS Emp
1	01/06/2000	10	123
2	02/06/2000	20	456
3	02/06/2000	10	123
4	05/07/2000	10	456
5	09/07/2000	30	789
6	09/07/2000	20	789
7	15/07/2000	40	789
8	15/07/2000	40	123

Algèbre relationnelle : *Commande*

*ATTENTION, si vous sélectionnez tous les attributs de la table,  
il n'y a plus de projection*

# Requêtes SQL simples

- Projection d'une table
  - Afficher les *noClient* et *dateCommande* de toutes les *Commandes*

```
SELECT      noClient,  dateCommande  
FROM        Commande
```

noClient	dateCommande
10	01/06/2000
20	02/06/2000
10	02/06/2000
10	05/07/2000
30	09/07/2000
20	09/07/2000
40	15/07/2000
40	15/07/2000

Doublons !

# Requêtes SQL simples

- Projection d'une table
  - Afficher les *noClient* et *dateCommande* de toutes les *Commandes*

```
SELECT      ALL noClient, dateCommande  
FROM        Commande
```

noClient	dateCommande
10	01/06/2000
20	02/06/2000
10	02/06/2000
10	05/07/2000
30	09/07/2000
20	09/07/2000
40	15/07/2000
40	15/07/2000

Doublons !

La forme générale réalise une PROJECTION sans élimination des doublons

# Requêtes SQL simples

- Projection d'une table **sans doublons**
  - Afficher les *noClient* et *dateCommande* de toutes les *Commandes*

```
SELECT      DISTINCT noClient, dateCommande  
FROM        Commande
```

noClient	dateCommande
10	01/06/2000
20	02/06/2000
10	02/06/2000
10	05/07/2000
30	09/07/2000
20	09/07/2000
40	15/07/2000

Algèbre relationnelle :  $\pi_{noClient, dateCommande} (Commande)$

# Requêtes SQL simples

- Sélection dans une table (WHERE)
  - Afficher les *Articles* de numéro supérieur à 30 dont le prix est inférieur à 20 Euros

```
SELECT      *
FROM        Article
WHERE       prixUnitaire < 20 AND noArticle > 30
```

noArticle	description	prixUnitaire	Quantité
60	Erable argenté	15.99	10
70	Herbe à puce	10.99	10
95	Génévrier	15.99	10

Algèbre relationnelle :  $\sigma_{prixUnitaire < 20.00 \text{ ET } noArticle > 30} (Article)$

# Requêtes SQL simples

- Sélection dans une table (WHERE)
  - Afficher les *Articles* dont le numéro est supérieur à 30 et le prix est inférieur à 20 Euros, ou le numéro est inférieur à 20 et le prix est supérieur à 30.

```
SELECT      *
FROM        Article
WHERE       (prixUnitaire < 20 AND noArticle > 30)
OR          (prixUnitaire > 30 AND noArticle < 20)
```

```
SELECT      *
FROM        Article
WHERE       prixUnitaire < 20 AND noArticle > 30
OR          prixUnitaire > 30 AND noArticle < 20
```

# Requêtes SQL simples

- Sélection dans une table (WHERE)
  - Afficher les *Commandes* du mois de juin de l'année 2000

```
SELECT      *
FROM        Commande
WHERE       dateCommande BETWEEN 01/06/2000 AND 30/06/2000
```

```
SELECT      *
FROM        Commande
WHERE       dateCommande >= 01/06/2000 AND dateCommande <=30/06/2000
```

N° Com	Date	N° Client	N°SS Emp
1	01/06/2000	10	123
2	02/06/2000	20	456
3	02/06/2000	10	123

# Requêtes SQL simples

- Sélection dans une table (WHERE)
  - Afficher les *Commandes* du *Client* numéro 10 ou 40 ou 80

```
SELECT      *
FROM        Commande
WHERE       noClient IN (10; 40; 80)
```

```
SELECT      *
FROM        Commande
WHERE       noClient = 10 OR noClient = 40 OR noClient = 80
```

# Requêtes SQL simples

- Sélection dans une table (WHERE)
  - Afficher les *Clients* dont le *nomClient* commence par *D*

```
SELECT *
FROM Client
WHERE nomClient LIKE "D*"
```

N°	Nom_Client	Téléphone	N°SS Emp
20	Dollard Cash	(888)888-8888	456
80	Dollard Cash	(333)333-3333	123

# Requêtes SQL simples

- Sélection dans une table (WHERE)
  - Afficher les *Clients* dont le *nomClient* contient le mot "*Dol*"

```
SELECT *
FROM Client
WHERE nomClient LIKE "*Dol*"
```

N°	Nom_Client	Téléphone	N°SS Emp
20	Dollard Cash	(888)888-8888	456
80	Dollard Cash	(333)333-3333	123

# Requêtes SQL simples

- Sélection dans une table (WHERE)
  - Afficher les *Clients* dont la 3<sup>ème</sup> lettre du *nomClient* est *L* et dernière lettre est un *h*

```
SELECT  *
FROM      Client
WHERE     nomClient LIKE "??L*h"
```

N°	Nom_Client	Téléphone	N°SS Emp
20	Dollard Cash	(888)888-8888	456
80	Dollard Cash	(333)333-3333	123

# Requêtes SQL simples

- Sélection dans une table (WHERE)
  - Afficher les *Clients* dont le téléphone a été saisi

```
SELECT      *
FROM        Client
WHERE       Téléphone IS NOT NULL
```

N°	Nom_Client	Téléphone	N°SS Emp
10	Hugh Paycheck	(999)999-9999	123
20	Dollard Cash	(888)888-8888	456
30	Ye San Le Sou	(777)777-7777	123
40	Le Comte Hasek	(666)666-6666	456
50	Hafedh Lajoie	(555)555-5555	789
60	Comtesse Hasek	(666)666-6666	789
70	Coco McPoulet	(444)444-4419	789
80	Dollard Cash	(333)333-3333	123

# Requêtes SQL simples

- Sélection-projection sur une table
  - Afficher les *noClient* et *dateCommande* des *Commandes* saisies après le *05/07/2000*

```
SELECT      noClient, dateCommande  
FROM        Commande  
WHERE       dateCommande > 05/07/2000
```

noClient	dateCommande
30	09/07/2000
20	09/07/2000
40	15/07/2000
40	15/07/2000

Algèbre relationnelle :  $\pi_{noClient, dateCommande} (\sigma_{dateCommande > 05/07/2000} (Commande))$

# Requêtes SQL simples

- Recherche des valeurs nulles
  - On utilise l'expression **IS [NOT] NULL**
  - **IS NULL** est vraie si et seulement si la valeur associée est absente (valeur nulle)
  - Exemple
    - Donnez les noms des employés qui n'ont pas de salaire fixe

```
SELECT      N°SS, Nom  
FROM        Employé  
WHERE       Salaire IS NULL
```

# Requêtes SQL simples

## Présentation du résultat

Il est possible de trier le résultat d'une requête en utilisant :

**ORDER BY**      nom-de-colonne [ **ASC |DESC** ]  
                  [, nom-de-colonne [ **ASC |DESC** ] ...]

Où

**ASC** = ordre ascendant et **DESC** = ordre descendant  
Par défaut, c'est ASC

# Requêtes SQL simples

- Sélection-projection-triée

- Afficher le *nom* et le *téléphone* des *Clients* dont le numéro est différent de 30 par ordre alphabétique

```
SELECT      Nom, Téléphone
FROM        Client
WHERE       Numéro <> 30
ORDER BY    NOM ASC, Téléphone ASC
```

Nom	Téléphone
Coco McPoulet	(444)444-4419
Comtesse Hasek	(666)666-6666
Dollard Cash	(888)888-8888
Dollard Cash	(333)333-3333
Hafedh Lajoie	(555)555-5555
Hugh Paycheck	(999)999-9999
Le Comte Hasek	(666)666-6666

Nom	Téléphone
Coco McPoulet	(444)444-4419
Comtesse Hasek	(666)666-6666
Dollard Cash	(333)333-3333
Dollard Cash	(888)888-8888
Hafedh Lajoie	(555)555-5555
Hugh Paycheck	(999)999-9999
Le Comte Hasek	(666)666-6666

Ordre de saisie  
est le suivant



# Conflits de noms d'attribut

On peut utiliser des ALIAS pour changer l'affichage des attributs. La syntaxe est :

nom-de-colonne [ [AS] **Nouveau\_nom** ]

Le nouveau nom doit être entre **deux crochets** sous ACCESS

# Requêtes SQL simples

- Projection avec ALIAS
  - Afficher le *numéro de sécurité sociale* et le *téléphone* de tous les *Clients*

```
SELECT      Nom, Téléphone, N°SS AS [Num Sécurité Sociale]  
FROM        Client
```

Nom_Client	Téléphone	Num sécurité sociale
Hugh Paycheck	(999)999-9999	951
Dollard Cash	(888)888-8888	753
Ye San Le Sou	(777)777-7777	486
Le Comte Hasek	(666)666-6666	197
Hafedh Lajoie	(555)555-5555	123
Comtesse Hasek	(666)666-6666	164
Coco McPoulet	(444)444-4419	188
Dollard Cash	(333)333-3333	133

Algèbre relationnelle :  $\pi_{nom, Téléphone, \rho_{N°SS \rightarrow [Num\ sécurité\ Sociale]}}(Client)$

# Requêtes SQL simples

- Avec paramètres

- Afficher le téléphone et le nom du client dont le numéro INSEE est saisi par l'utilisateur

```
SELECT      Nom, Téléphone, N°SS AS [Num sécurité sociale]
FROM        Client
WHERE       N°SS = [Saisissez le Num Sécurité Sociale]
```



Nom_Client	Téléphone	Num sécurité sociale
Hugh Paycheck	(999)999-9999	951

Algèbre relationnelle :  $\pi_{nom, Téléphone} (\sigma_{N°SS = [Saisissez le Num sécurité Sociale]} (Client))$

# Opérateurs ensemblistes

- Union
- Intersect (Pas implémenté dans Access)
- Except ou Minus (Pas implémenté dans Access)

Tous les SELECT doivent avoir le même nombre de colonnes sélectionnées, et leurs types doivent être un à un identiques

# Opérateurs ensemblistes

- Union
- Intersect (Pas implémenté dans Access)
- Except ou Minus (Pas implémenté dans Access)

Les doublons sont éliminés (DISTINCT implicite) sauf si l'on spécifie l'opérateur ALL

# Opérateurs ensemblistes

- Union
- Intersect (Pas implémenté dans Access)
- Except ou Minus (Pas implémenté dans Access)

Les noms des colonnes à l'affichage sont ceux du premier **SELECT**

# Opérateurs ensemblistes

## Syntaxe:

table-expression

```
{ UNION | EXCEPT |INTERSECT } [ALL] [CORRESPONDING  
[BY (attributs)] ]
```

table-expression

# Opérateurs ensemblistes

- Exemple
  - Union
    - Afficher le N°SS et le nom de tous les clients et de tous les employés

```
(SELECT      N°SS as N°INSEE, nomClient as nomPersonne  
FROM        Client)  
UNION ALL  
(SELECT      N°SS, nom_Employé  
FROM        Employé)
```

N° INSEE	NomPersonne
951	Hugh Paycheck
753	Dollard Cash
486	Ye San Le Sou
197	Le Comte Hasek
123	M. Barth
164	Comtesse Hasek
188	Coco McPoulet
133	Dollard Cash
456	M. Chbeir
789	Mme Kacimi
123	M. Barth

# Opérateurs ensemblistes

- Exemple
  - Union
    - Afficher le N°SS et le nom de tous les clients et de tous les employés

```
(SELECT      N°SS as N°INSEE, nomClient as nomPersonne  
FROM        Client)  
UNION  
(SELECT      N°SS, nom_Employé  
FROM        Employé)
```

N° INSEE	NomPersonne
951	Hugh Paycheck
753	Dollard Cash
486	Ye San Le Sou
197	Le Comte Hasek
123	M. Barth
164	Comtesse Hasek
188	Coco McPoulet
133	Dollard Cash
456	M. Chbeir
789	Mme Kacimi

# Opérateurs ensemblistes

- Exemple
  - Intersect
    - Afficher le N°SS et le nom de tous les clients qui sont également employés

```
(SELECT      N°SS as N°INSEE, nomClient as nomPersonne  
FROM        Client)  
INTERSECT  
(SELECT      N°SS, nom_Employé  
FROM        Employé)
```

N° INSEE	NomPersonne
123	M. Barth

L'intersection est faite sur la position des colonnes

# Opérateurs ensemblistes

- Exemple
  - Intersect
    - Afficher le N°SS et le nom de tous les clients qui sont également employés

```
(SELECT      N°SS, nomClient as nomPersonne  
FROM        Client)  
INTERSECT CORRESPONDING BY (N°SS)  
(SELECT      N°SS, nom_Employé  
FROM        Employé)
```

N° SS	NomPersonne
123	M. Barth

L'intersection est faite sur la colonne N°SS

# Opérateurs ensemblistes

- Exemple
  - Intersect
    - Afficher le N°SS et le nom de tous les clients qui sont également employés

```
(SELECT      *
  FROM        Client)
INTERSECT CORRESPONDING
(SELECT      *
  FROM        Employé)
```

N° SS	NomPersonne
123	M. Barth

L'intersection est faite sur toutes les colonnes communes

# Produit cartésien

- Définition
  - Il suffit de citer dans une requête SELECT sans WHERE les relations qui participent au produit après la **clause FROM**

SQL 1

```
SELECT      *
FROM        R1 ,  R2
```

SQL 2

```
SELECT      *
FROM        R1 CROSS JOIN R2
```

Algèbre relationnelle :  $R1 \times R2$

# Produit cartésien

- Exemple
  - Afficher toutes les combinaisons possibles de lignes de Client et de Commande

```
SELECT      *
FROM        Client, Commande
```

Algèbre relationnelle : *Client × Commande*

# Conflits de noms d'attributs

- Pour lever les ambiguïtés ou pour faciliter la compréhension des requêtes, deux techniques sont utilisées dans SQL
  - Préfixage
  - Renommage

# Conflits de noms d'attributs

- Préfixer

- les attributs peuvent être préfixés par les noms de relation dans les diverses clauses
- Syntaxe

Nom\_table . nom\_colonne

- Exemple

```
SELECT Client.Nom, ...
FROM Client, ...
```

# Conflits de noms d'attributs

- Renommer les relations
  - Il est parfois utile de renommer les relations
    - Nom long d'une relation
    - Auto jointure
  - Il suffit de mettre le nouveau nom après la relation à renommer (avec ou sans le mot-clé AS)

# Conflits de noms d'attributs

- Renommer les relations
  - Exemple : Afficher les coordonnées des Clients qui ont effectué des commandes

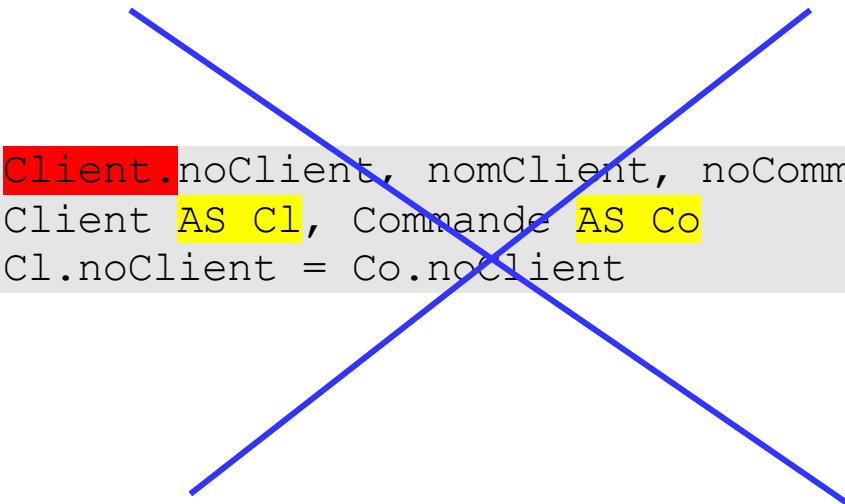
```
SELECT      Client.noClient, nomClient, noTéléphone, noCommande,  
           dateCommande  
FROM        Client, Commande  
WHERE       Client.noClient = Commande.noClient
```

```
SELECT      Cl.noClient, nomClient, noTéléphone, noCommande,  
           dateCommande  
FROM        Client AS Cl, Commande AS Co  
WHERE       Cl.noClient = Co.noClient
```

# Conflits de noms d'attributs

- Attention
  - Quand on renomme une relation, on ne peut plus préfixer les attributs avec l'ancien nom

```
SELECT Client.noClient, nomClient, noCommande  
FROM Client AS Cl, Commande AS Co  
WHERE Cl.noClient = Co.noClient
```



# Thêta-produit (thêta-jointure)

- Exemple

- Afficher toutes les combinaisons possibles de lignes de Client numéro 10 et de Commande

```
SELECT      Client.noClient, nomClient, noTéléphone, noCommande
FROM        Client, Commande
WHERE       Client. noClient = 10
```

```
SELECT      Client.noClient, nomClient, noTéléphone, noCommande
FROM        Client JOIN Commande
ON         Client.noClient = 10
```

```
SELECT      Client.noClient, nomClient, noTéléphone, noCommande
FROM        Client JOIN Commande
USING     (Client.noClient = 10)
```

Algèbre relationnelle :  $\pi_{Client.noClient, nomClient, noTéléphone, noCommande}$   
 $(Client \bowtie_{noClient=10} Commande)$

# Jointure naturelle (\*)

- Exemple
  - Afficher toutes les informations au sujet des *Clients* et de leurs *Commandes*

```
SELECT      Client.noClient, nomClient, noTéléphone, noCommande  
FROM        Client, Commande  
WHERE       Client.noClient = Commande.noClient
```

```
SELECT      Client.noClient, nomClient, noTéléphone, noCommande  
FROM        Client NATURAL JOIN Commande
```

La jointure avec **NATURAL JOIN** est faite sur les colonnes de même nom (noClient)

$$\pi_{Client.noClient, nomClient, noTéléphone, noCommande}$$
$$(\sigma_{Client.noClient = Commande.noClient} (Client \times Commande))$$

# Jointure naturelle (\*)

- Exemple
  - Afficher toutes les informations au sujet des *Clients* et de leurs *Commandes*

```
SELECT      Client.noClient, nomClient, noTéléphone, noCommande  
FROM        Client JOIN Commande ON  
           Client.noClient = Commande.numeroClient
```

Quand les noms des colonnes sont différents, on peut utiliser JOIN avec une condition

# Jointure naturelle (\*)

- Exemple
  - Afficher le nom des *Clients* qui ont passé des *Commandes le 01/07/2002*

```
SELECT      nomClient
FROM        Client Cl, Commande Co
WHERE       Cl.noClient = Co.noClient
AND Co.date = 01/07/2002
```

```
SELECT      nomClient
FROM        Client
WHERE       noClient IN
           (SELECT noClient
            FROM  COMMANDE
            WHERE Date = 01/07/2002)
```

# Jointure naturelle de plusieurs tables

- Exemple
  - Afficher le *nom* des *Clients* qui ont commandé au moins un article *Imprimante*

```
SELECT      nomClient
FROM        Client, Commande, DétailsCommande, Article
WHERE       description = 'Imprimante' AND
           Client.noClient = Commande.noClient AND
           Commande.noCommande = DétailsCommande.noCommande AND
           DétailsCommande.noArticle = Article.noArticle
```

$$\pi_{nomClient} (\sigma_{description = "Imprimante"} (Client * Commande * DétailsCommande * Article))$$

# Jointure naturelle de plusieurs tables

- Exemple
  - Afficher le *nom* des *Clients* qui ont commandé au moins un article *Imprimante*

```
SELECT      nomClient  
FROM        Client  
WHERE       noClient IN
```

```
SELECT noClient  
FROM Commande  
WHERE noCommande IN
```

```
SELECT noCommande  
FROM DétailsCommande  
WHERE noArticle IN
```

```
SELECT noArticle  
FROM Article  
WHERE description = 'Imprimante'
```

# Auto-Jointure

- Exemple
  - Afficher les *Clients* qui ont le même numéro de téléphone

```
SELECT      Client.noClient, Client2.noClient
FROM        Client, Client AS Client2
WHERE       Client.noTéléphone = Client2.noTéléphone
AND         Client.noClient <> Client2.noClient
```

$$\pi_{Client.noClient, Client2.noClient} (\sigma_{Client.noTéléphone = Client2.noTéléphone} (Client \times \rho_{Client2} (Client)))$$

L'utilisation d'ALIAS est indispensable

# Auto-Jointure

- Exemple
  - Afficher les *Clients* qui ont le même numéro de téléphone

```
SELECT      noClient, noClient2
FROM        Client JOIN Client AS Client2
            ON Client.noTéléphone = Client2.noTéléphone
```

# Auto-Jointure

- Exemple
  - Afficher les *Clients* qui ont le même numéro de téléphone

```
SELECT      noClient, noClient2
FROM        Client NATURAL JOIN
           Client AS Client2(noClient2, nomClient2, noTéléphone, N°SS2)
```

$$\pi_{noClient, noClient2} (Client^* \rho_{Client2(noClient2, nomClient2, noTéléphone, N°SS2)} (Client))$$

La jointure avec **NATURAL JOIN** est faite sur les colonnes de même nom

# Expressions générales sur les colonnes

- Exemple
  - Afficher la liste des articles avec le *prix Unitaire actuel et le prix taxé (augmenté de 15%)*

```
SELECT noArticle, prixUnitaire, prixUnitaire*1.15 AS prixPlusTaxe  
FROM Article
```

noArticle	prixUnitaire	prixPlusTaxe
10	10.99	12.64
20	12.99	14.94
40	25.99	29.89
50	22.99	26.44
60	15.99	18.39
70	10.99	12.64
80	26.99	31.04
81	25.99	29.89
90	25.99	29.89
95	15.99	18.39

# Expressions générales sur les colonnes

- Exemple
  - Afficher le détail de chacun des *Articles* commandés incluant le prix total avant et après la taxe de 15%

```
SELECT L.noArticle, quantité, prixUnitaire, prixUnitaire*quantité AS total,  
       prixUnitaire*quantité*1.15 AS totalPlusTaxe  
FROM      DétailsCommande AS L, Article AS A  
WHERE     L.noArticle = A.noArticle
```

# Expression sur colonne du WHERE

- Exemple
  - Afficher les *Articles* dont le *prixUnitaire* augmenté de 15% est inférieur à 16

```
SELECT noArticle, prixUnitaire  
FROM Article  
WHERE prixUnitaire*1.15 < 16
```

# Fonctions

- Dans SQL, on peut utiliser plusieurs types de fonctions
  - Fonctions dépendantes du SGBD
    - Fonctions sur chaînes de caractères
    - Fonctions sur des entiers
    - Fonctions sur des dates
  - Fonctions génériques
    - Fonctions arithmétiques
    - Fonctions ensemblistes

# Fonctions dépendantes du SGBD

- Exemple
  - Afficher les *Commandes de la journée*

```
SELECT *
FROM      Commande
WHERE     dateCommande = Maintenant()
```

# Fonctions arithmétiques et ensemblistes

- Elles peuvent être utilisées
  - dans la clause **SELECT**
  - dans la clause **WHERE**
- Elles agissent sur un ensemble de valeurs d'une colonne
- Elles donnent toujours une seule valeur

# Fonctions arithmétiques et ensemblistes

## • FONCTIONS ARITHMETIQUES

- +, \*, /, -

## • FONCTIONS ENSEMBLISTES

- **MAX** : Fournit la valeur maximale
- **MIN** : Fournit la valeur minimale
- **COUNT** : Fournit la cardinalité d'un ensemble
- **SUM** : Somme de toutes les valeurs
- **AVG** : Moyenne de toutes les valeurs

Valeurs numériques,  
caractère et date

Valeurs Numériques

# Fonctions ensemblistes

- Si l'argument est l'ensemble vide
  - COUNT donne 0
  - Les autres fonctions donnent NULL
- L'argument de ces fonctions ne peut pas être une fonction
  - **SELECT AVG(SUM(Salaire)) FROM ... est invalide**
  - Mais on peut faire:
    - **SELECT AVG(x) FROM (SELECT SUM(Salaire) AS x FROM ..)**
- Les valeurs nulles sont éliminées avant l'application de la fonction, sauf pour COUNT(\*)

# Fonctions ensemblistes

- **COUNT(\*)**

- **COUNT(\*)** compte toutes les lignes d'une table ou d'un résultat d'une sélection (sans éliminer les doublons)
- **COUNT(DISTINCT attribut)** compte le nombre d'attributs sans doublons
- **COUNT(DISTINCT \*) est invalide**

# Fonctions ensemblistes

- **Exemple**

- Afficher le salaire moyen des employés

```
SELECT AVG(Salaire)  
FROM Employé
```

- Afficher les employés qui gagnent plus que le salaire moyen

```
SELECT Nom  
FROM Employé  
WHERE Salaire >= (SELECT AVG(Salaire)  
                   FROM Employé)
```

# Fonctions ensemblistes

- **Exemple**

- Afficher le nombre de salariés qui ont un salaire fixe

```
SELECT COUNT(*)  
FROM      Employé  
WHERE     Salaire IS NOT NULL
```

- Afficher le nombre d'employés qui ne gagnent pas le même salaire

```
SELECT COUNT(DISTINCT Salaire)  
FROM      Employé
```

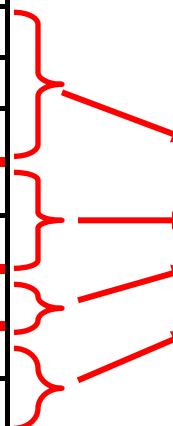
# Partition d'une table avec GROUP BY

- Exemple

- Afficher **le nombre de Commandes** de chaque *Client* (qui a passé au moins une *Commande*)

```
SELECT      noClient, COUNT (*) AS nombreCommandes  
FROM        Commande  
GROUP BY    noClient
```

Table <i>Commande</i>		
noCommande	dateCommande	noClient
1	01/06/2000	10
3	02/06/2000	10
4	05/07/2000	10
2	02/06/2000	20
6	09/07/2000	20
5	09/07/2000	30
7	15/07/2000	40
8	15/07/2000	40

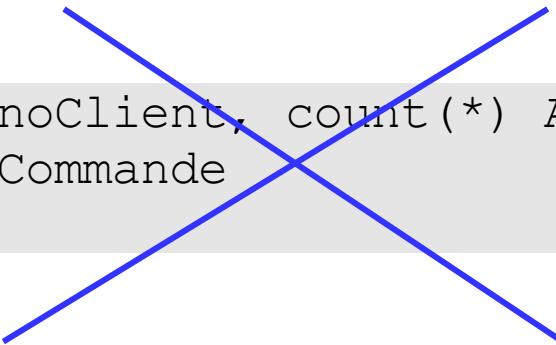


noClient	nombreCommandes
10	3
20	2
30	1
40	2

# Partition d'une table avec GROUP BY

- Il faut dire que le GROUP BY exige

```
SELECT      noClient, count(*) AS nombreCommandes  
FROM        Commande
```



```
SELECT      noClient, Count(*) AS nombreCommandes  
FROM        Commande  
GROUP BY    noClient
```

```
SELECT      COUNT(*) AS nombreCommandes  
FROM        Commande
```

# Partition d'une table avec GROUP BY

## • Exemple

- Pour chacune des *détailsLivraison* (pour lesquelles au moins une *Livraison* a été effectuée), affichez le *noCommande* et *noArticle*, la quantité totale livrée et le nombre de *Livraisons* effectuées

```
SELECT      noCommande, noArticle, SUM(quantitéLivrée) AS totalLivré,  
           COUNT(*) AS nombreLivraisons  
FROM        DétailLivraison  
GROUP BY    noCommande, noArticle
```

Table <i>DétailLivraison</i>			
noLivraison	noCommande	noArticle	quantitéLivrée
100	1	10	7
101	1	10	3
100	1	70	5
102	2	40	2
102	2	95	1
100	3	20	1
103	1	90	1
104	4	40	1
105	5	70	2

noCommande	noArticle	totalLivré	nombreLivraisons
1	10	10	2
1	70	5	1
1	90	1	1
2	40	2	1
2	95	1	1
3	20	1	1
4	40	1	1
5	70	2	1

# Clause HAVING

- Exemple
  - Afficher le nombre de *Commandes* de chaque *Client* qui a passé deux *Commandes* ou plus

Table <i>Commande</i>		
noCommande	dateCommande	noClient
1	01/06/2000	10
3	02/06/2000	10
4	05/07/2000	10
2	02/06/2000	20
6	09/07/2000	20
5	09/07/2000	30
7	15/07/2000	40
8	15/07/2000	40

Diagram illustrating the aggregation process:

- The original table has 8 rows.
- Red curly braces group rows by noClient value:
  - noClient 10: Rows 1, 2, 3, 4 (total 4 rows)
  - noClient 20: Rows 5, 6 (total 2 rows)
  - noClient 30: Row 7 (total 1 row)
  - noClient 40: Rows 8, 9 (total 2 rows)
- Red arrows point from each group to the corresponding row in the result table.
- The result table shows the count of commands for each client who placed at least 2 commands.

noClient	nombreCommandes
10	3
20	2
30	1
40	2

# Clause HAVING

- Exemple

- Afficher le nombre de *Commandes* de chaque *Client* qui a passé deux *Commandes* ou plus

```
SELECT      noClient, COUNT(*) AS nombreCommandes
FROM        Commande
GROUP BY    noClient
WHERE       COUNT(*) >= 2
```

Mais la clause WHERE ne peut pas être utilisée après GROUP BY

```
SELECT      noClient, COUNT(*) AS nombreCommandes
FROM        Commande
GROUP BY    noClient
HAVING     COUNT(*) >= 2
```

# Clause HAVING

- Exemple

- Afficher le nombre de *Commandes* de chaque *Client* passées après le 02/06/2000

```
SELECT      noClient, COUNT(*) AS nombreCommandes
FROM        Commande
WHERE       dateCommande > '02/06/2000'
GROUP BY    noClient
```

Table <i>Commande</i>		
noCommande	dateCommande	noClient
1	01/06/2000	10
3	02/06/2000	10
4	05/07/2000	10
2	02/06/2000	20
6	09/07/2000	20
5	09/07/2000	30
7	15/07/2000	40
8	15/07/2000	40

noClient	nombreCommandes
10	1
20	1
30	1
40	2

# Clause HAVING

- Exemple

- Afficher le nombre de *Commandes* de chaque *Client* qui a passé deux Commandes ou plus après le 02/06/2000

```
SELECT      noClient, COUNT(*) AS nombreCommandes
FROM        Commande
WHERE       dateCommande > '02/06/2000'
GROUP BY    noClient
HAVING     COUNT(*) >= 2
```

Table <i>Commande</i>		
noCommande	dateCommande	noClient
1	01/06/2000	10
3	02/06/2000	10
4	05/07/2000	10
2	02/06/2000	20
6	09/07/2000	20
5	09/07/2000	30
7	15/07/2000	40
8	15/07/2000	40

noClient	nombreCommandes
10	1
20	1
30	1
40	2

# SELECT imbriqué

- Exemple
  - Afficher toutes les informations concernant les *Clients* qui ont passé au moins une *Commande*

```
SELECT      noClient, nomClient, noTéléphone
FROM        Client
WHERE       noClient IN
            (SELECT  noClient
             FROM    Commande)
```

```
SELECT      Client.noClient, nomClient, noTéléphone
FROM        Client, Commande
WHERE       Client.noClient = Commande.noClient
```

# SELECT imbriqué

- Exemple
  - Afficher les *LigneCommandes* pour lesquelles au moins une *Livraison* a été effectuée

```
SELECT      *
FROM        LigneCommande
WHERE       (noCommande, noArticle) IN
            (SELECT      noCommande, noArticle
             FROM        DétailLivraison)
```

# SELECT imbriqué

- Exemple

- Afficher les *Commandes* de M. Bob

```
SELECT      *
FROM        Commande
WHERE       noClient =
            (SELECT      noClient
             FROM        Client
             WHERE       nomClient = 'Bob' )
```

Attention si plusieurs lignes sont renvoyées par la SELECT imbriquée

# SELECT imbriqué corrélé

- Exemple

- Afficher les informations au sujet des *Clients* qui ont passé au moins une *Commande*

```
SELECT      *
FROM        Client
WHERE       0 <
           (SELECT      COUNT(*)
                FROM       Commande
                WHERE      noClient = Client.noClient)
```

Référence à une colonne non locale

POUR chaque ligne de *Client*

Exécuter le SELECT suivant :

```
(SELECT      COUNT(*)
           FROM       Commande
           WHERE      noClient = Client.noClient)
```

SI le compte retourné > 0

Placer la ligne de *Client* dans la table du résultat à retourner

FIN SI

FIN POUR

# Condition ALL

- Exemple

- Afficher les employés qui touchent un salaire supérieur à n'importe quel autre employé (le plus élevé)

```
SELECT      *
FROM        Employé
WHERE       Salaire >= ALL (
                      SELECT      Salaire
                      FROM        Employé)
```

# Condition ALL

- Exemple

- Afficher les *Commandes* qui ont été passées après la dernière *Livraison* (c'est-à-dire à une date ultérieure)

```
SELECT * FROM Commande  
WHERE      dateCommande > ALL  
          (SELECT dateLivraison  
           FROM   Livraison)
```

# Condition ANY (SOME)

- Exemple

- Afficher les employés qui touchent un salaire supérieur à au moins un salaire d'un employé dont le nom commence par c

```
SELECT      *
FROM        Employé
WHERE       Salaire > ANY (
                      SELECT      Salaire
                      FROM        Employé
                      WHERE Nom_employé like "c*")
```

# Condition ANY

- Exemple
  - Afficher les *Commandes* qui ont été passées après au moins une des *Livraisons*

```
SELECT * FROM Commande  
WHERE dateCommande > ANY  
      (SELECT dateLivraison  
       FROM Livraison)
```

# Test d'ensemble vide EXISTS

- Exemple
  - Afficher les *Clients* qui ont passé au moins une *Commande*

```
SELECT      *
FROM        Client
WHERE       EXISTS
           (SELECT      *
            FROM        Commande
            WHERE       noClient = Client.noClient)
```

Référence à une colonne non locale

POUR chaque ligne de *Client*

Exécuter le SELECT suivant :

```
(SELECT      *
  FROM        Commande
  WHERE       noClient = Client.noClient)
```

SI le résultat de la sélection n'est pas vide

Placer la ligne de *Client* dans la table du résultat à retourner

FIN SI

FIN POUR

# Test d'ensemble vide EXISTS

- Exemple
  - Afficher les *Clients* qui ne sont pas en même temps des *employés*

```
SELECT      *
FROM        Client e
WHERE       NOT EXISTS
           (SELECT      *
            FROM        Employé
            WHERE       N°SS_Employé = e.N°SS)
```

# Test de double (UNIQUE)

- Exemple
  - Afficher les clients qui portent le même nom

```
SELECT nomClient  
FROM Client e  
WHERE NOT UNIQUE  
      (SELECT nomClient  
       FROM Client  
       WHERE noClient=e.noClient)
```



# CRÉATION DES VUES

# Définition

- Une vue est une vision partielle ou particulière des données d'une ou plusieurs tables de la base
- La définition d'une vue est donnée par un SELECT qui indique les données de la base qui seront vues
- Seule la définition de la vue est enregistrée dans la base, et pas les données de la vue. On peut parler de table virtuelle.

# Commande de création d'une vue

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

*CREATE VIEW nom\_vue (col<sub>1</sub>, col<sub>2</sub>...) AS SELECT ...*

- Pour supprimer une vue il suffit d'exécuter la commande suivante

*DROP VIEW vue*

# Exemple

- Crédation d'une vue de la table Article

**CREATEVIEW article\_v AS (SELECT noArticle, description, Quantité FROM article )**

**Article**

noArticle	description	prixUnitaire	Quantité
60	Erable argenté	15.99	10
70	Herbe à puce	10.99	10
95	Génévrier	15.99	10

**Article\_v**

noArticle	description	Quantité
60	Erable argenté	10
70	Herbe à puce	10
95	Génévrier	10

La spécification des noms des colonnes de la vue est facultative : par défaut, les colonnes de la vue ont pour nom les noms des colonnes résultats du SELECT. Si certaines colonnes résultats du SELECT sont des expressions sans nom, il faut alors obligatoirement spécifier les noms de colonnes de la vue.

# Exemple

- Cration d'une vue de la table Article

**CREATEVIEW article\_v (v1,v2) AS (SELECT noArticle, description FROM article )**

**Article**

noArticle	description	prixUnitaire	Quantite
60	Erable argente	15.99	10
70	Herbe a puce	10.99	10
95	Genevrier	15.99	10

**Article\_v**

v1	v2
60	Erable argente
70	Herbe a puce
95	Genevrier

# Utilité des vues

- les vues permettent de dissocier la façon dont les utilisateurs voient les données du découpage en tables
- Séparation de l'aspect externe (ce que voit un utilisateur particulier de la base) de l'aspect conceptuel (comment a été conçu l'ensemble de la base)

# Utilité des vues

- Une vue peut aussi être utilisée pour restreindre les droits d'accès à certaines colonnes et à certaines lignes d'une table
- Une vue peut également simplifier la consultation de la base en enregistrant des SELECT complexes.

# Exemple (droits limités)

- **CREATE VIEW** article\_v **AS** (SELECT noArticle, description, Quantité FROM article )
- **GRANT** select **ON** article\_v **TO** user

# Exemple – (simplification)

Si on crée la vue suivante:

```
CREATE VIEW salarie_sal AS  
SELECT nom, salaire + NVL(comission, 0) GAINS, nom_departement  
FROM salarie NATURAL JOIN departement)
```

la liste des salariés avec leur rémunération totale et le nom de leur département sera obtenue simplement par :

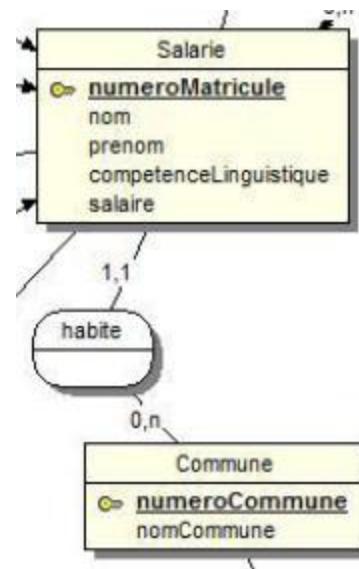
```
SELECT * FROM salarie_sal
```

# Mise à jour d'une vue

- Sous certaines conditions, il est possible d'effectuer des DELETE, INSERT et des UPDATE à travers des vues.
- Les conditions suivantes doivent être remplies :
  - Pour effectuer un DELETE, le select qui définit la vue ne doit pas comporter de jointure, de group by, de distinct, de fonction de groupe ;
  - Pour un UPDATE, en plus des conditions précédentes, les colonnes modifiées doivent être des colonnes réelles de la table sous-jacente ;
  - Pour un INSERT, en plus des conditions précédentes, toute colonne « not null » de la table sous-jacente doit être présente dans la vue.

# Exercice

- Créez une vue qui permet de grouper les salariés qui ont un salaire plus grand que 2000 ainsi que les noms des communes où ils habitent



# Solution

```
CREATEVIEW sal_com as (select nom,  
salaire, nomcommune FROM salarie  
NATURAL JOIN commune WHERE  
salaire > 2000 )
```

# Mise à jour d'une vue

- Une vue peut créer des données qu'elle ne pourra pas visualiser

*CREATE TABLE t1 (a INT);*

*CREATE VIEW v1 AS SELECT \* FROM t1 WHERE a < 2*

*Insert into v1 values (3)*

- Si l'on veut éviter cela il faut ajouter WITH CHECK OPTION dans l'ordre de création de la vue après l'interrogation définissant la vue

*CREATE VIEW v1 AS SELECT \* FROM t1 WHERE a < 2*

*WITH CHECK OPTION*



# **PROCÉDURES ET FONCTIONS STOCKÉES**

# Procédure stockée

- Une procédure stockée est un programme qui comprend des instructions SQL précompilées et qui est enregistré dans la base de données
- Le plus souvent le programme est écrit dans un langage spécial qui contient à la fois des instructions procédurales et des ordres SQL

# Procédure stockée

- Malheureusement, la notion de procédure stockée n'est standardisé et elle est liée fortement à chacun des SGBD
- Oracle offre ainsi le langage PL/SQL qui se rapproche de la syntaxe qui inclut des ordres SQL
- Les procédures stockées d'Oracle peuvent aussi être écrites en Java et en langage C

# Utilité des procédures stockées

- Le trafic sur le réseau est réduit car les clients SQL ont seulement à envoyer l'identification de la procédure et ses paramètres au serveur sur lequel elle est stockée
- Les procédures sont précompilées une seule fois quand elles sont enregistrées. De plus les erreurs sont repérées dès la compilation et pas à l'exécution
- Les développeurs n'ont pas à connaître les détails de l'exécution des procédures
- La gestion et la maintenance des procédures sont facilitées car elles sont enregistrées sur le serveur et ne sont pas dispersées sur les postes clients

# Création d'une procédure stockée

- En Oracle, on peut créer une nouvelle procédure stockée par la commande

```
CREATE OR REPLACE PROCEDURE <nom>[ (liste de paramètres) ]  
AS|IS  
<zone de déclaration de variables>  
BEGIN  
<corps de la procédure>  
EXCEPTION  
<traitement des exceptions>  
END;
```

# Création d'une procédure stockée

- Nous souhaitons créer une procédure stockée sous Oracle qui prend en paramètre
  - un numéro de département
  - un pourcentage,
  - augmente tous les salaires des employés de ce département de ce pourcentage

# Création d'une procédure stockée

```
CREATE OR REPLACE PROCEDURE aug
(unDept in integer, pourcentage in decimal) is
begin
update salarie
set salaire = salaire * (1 + pourcentage / 100)
where iddept = unDept;
end;
```

# Paramètres d'une procédure

<b>IN</b>	<b>OUT</b>	<b>IN OUT</b>
DEFAUT	A PRECISER	A PRECISER
VALEUR PASSEE A UN SOUS PROGRAMME	RETOURNEE A L 'ENVIRONNEMENT D 'APPEL	PASSEE A UN SOUS PROGRAMME ET RETOURNEE
LITERAL EXPRESSION VARIABLE INITIALISEE	VARIABLE	VARIABLE

# PROCEDURES / PARAMETRES

```
SQL> CREATE OR REPLACE PROCEDURE query_emp
```

```
(v_id IN emp.empno%TYPE,  
 v_name OUT emp.ename%TYPE,  
 v_salary OUT emp.sal%TYPE,  
 v_comm OUT emp.comm%TYPE)
```

```
IS
```

```
BEGIN
```

```
    SELECT ename,sal,comm  
    INTO v_name,v_salary,v_comm  
    FROM emp  
    WHERE empno = v_id;
```

```
END query_emp;
```

```
/
```

```
Procedure created.
```

Une variable en entrée du même type que l'attribut *empno*

Une variable de sortie du même type que l'attribut *comm*

# PROCEDURES / PARAMETRES

- TEST ET VISUALISATION SOUS SQLPLUS

```
SQL> variable g_name      varchar2(15)
SQL> variable g_salary    number
SQL> variable g_comm      number
```

```
SQL> EXECUTE query_emp (7389,:g_name,:g_salary,:g_comm)
```

```
SQL> PRINT g_name
```

# Remarques

- Sous Oracle, on peut avoir une description des paramètres d'une procédure stockée par la commande

*DESC nom-procédure*

- Les erreurs de compilation des procédures stockées peuvent être vues sous Oracle par la commande

*show errors* dans SQL\*PLUS

# Fonction Stockée

- La différence entre une fonction et une procédure est qu'une fonction renvoie une valeur
- Le code est semblable au code d'une procédure stockée mais la déclaration de la fonction indique le type de la valeur renournée
- Seuls les paramètres de type IN sont acceptés

# FONCTIONS / CREATION

CREATE OR REPLACE FUNCTION <nom>[(parametres)] RETURN <type du resultat>

```
CREATE OR REPLACE FUNCTION get_sal (v_id IN emp.empno%TYPE)
  RETURN NUMBER
IS
  v_salary emp.sal%TYPE := 0;
BEGIN
  SELECT sal
    INTO v_salary
   FROM emp
  WHERE empno = v_id;
RETURN (v_salary);
END get_sal;
/
```