



# BASES DE DONNÉES

L3/L3 new/L3 INT

*Abdelkrim LAHLOU*

*Abdelkrim.lahlou@intervenants.efrei.net*

# Déroulement

- **Volume horaire :**
  - Cours : 06h (4 séances de 1h30)
  - TD : 10h30 (3 séances de 3h30)
  - TP : 10h30 (3 séances de 3h30)
  - PRoJet : 03h30 (1 séances de 3h30)
- **Evaluation :**
  - Note de TP : 20 % (TP 80% + 20 % TD)
  - Note Projet : 30 %
  - Note DE : 50%

**Objectifs** : Etre capable de bien concevoir,  
implémenter et manipuler une base de données

# Plan

- **Partie I : Conception de bases de données**
  - Modèle conceptuel de données (MCD)
  - Modèle logique de données relationnel (MLD)
  - Traduction MCD vers MLD et reverse-engineering
- **Partie II : Dépendances fonctionnelles et normalisation de bases de données**
  - Dépendances fonctionnelles
  - Formes normales et normalisation de bases de données
- **Partie III : Concepts avancés en bases de données**
  - Vues et droits d'accès
  - Transactions, résistance aux pannes et concurrence d'accès
  - Procédures stockées et déclencheurs
- **Partie IV : Interopérabilité des bases de données et applications**
  - Bases de données et Java
  - API JDBC (Java DataBase Connectivity)

# Références

## □ Quelques livres utiles:

- [Database Systems: The Complete Book](#) : Hector Garcia-Molina  
Jeffrey D. Ullman Jennifer Widom, Pearson Prentice Hall  
(<https://people.inf.elte.hu/miiqaai/elektroModulatorDva.pdf>)
- [Bases de données](#) : Georges Gardarin, Edition Eyrolles
- [PostGreSQL par la pratique](#). John C. Worsley, Joshua D. Drake  
(O'Reilly – 2002 ISBN : 2-84177-211-X )
- Lien du site de Jeff ULLMAN : <http://infolab.stanford.edu/~ullman/>

# INTRODUCTION

# INTRODUCTION

Une **base de données** (correcte) est un ensemble **structuré**, **cohérent** (non contradictoire) et **pertinent** (représentatif du problème traité et sans redondances) d'informations fortement liées.

Il en découle la nécessité d'appliquer une **méthode de conception** stricte, rendant possible la conception globale d'une base de données (c-à-d éviter d'avoir un ensemble de points de vue partiels), permettant des maintenances aisées, impliquant une documentation complète et des notions de modularité.

**Il est impératif que les données soient séparées des traitements.**

# INTRODUCTION

Un **Système de Gestion de Bases de Données** (SGBD) est un logiciel qui permet d'interagir avec une base de données (définition des données, consultation, mises à jour) avec les objectifs suivants :

- > **Relation entre les données** : assurer la corrélation des données;
- > **Cohérence** : contrôle d'intégrité de la base et gestion des autorisations de modifications;
- > **Souplesse (confort), économie de place mémoire et rapidité d'accès** : langage de haut niveau SQL (Structured Query Language) qui permet des requêtes en un langage presque naturel du genre "Je veux la liste de tous les étudiants aux yeux bleus qui portent des lunettes";
- > **Sécurité** : sauvegardes périodiques;
- > **Partage des données** : gérer l'accès à des données communes pour des applications différentes;
- > **Indépendance des données** : idéalement (ce n'est pas tout à fait le cas), l'utilisateur n'aurait pas à savoir comment elles sont implantées physiquement.

# INTRODUCTION

Un **SGBD** est constitué de trois niveaux :

-> **Niveau physique** : implantation des données en machine;

    Ce niveau étant le problème des informaticiens purs et durs.

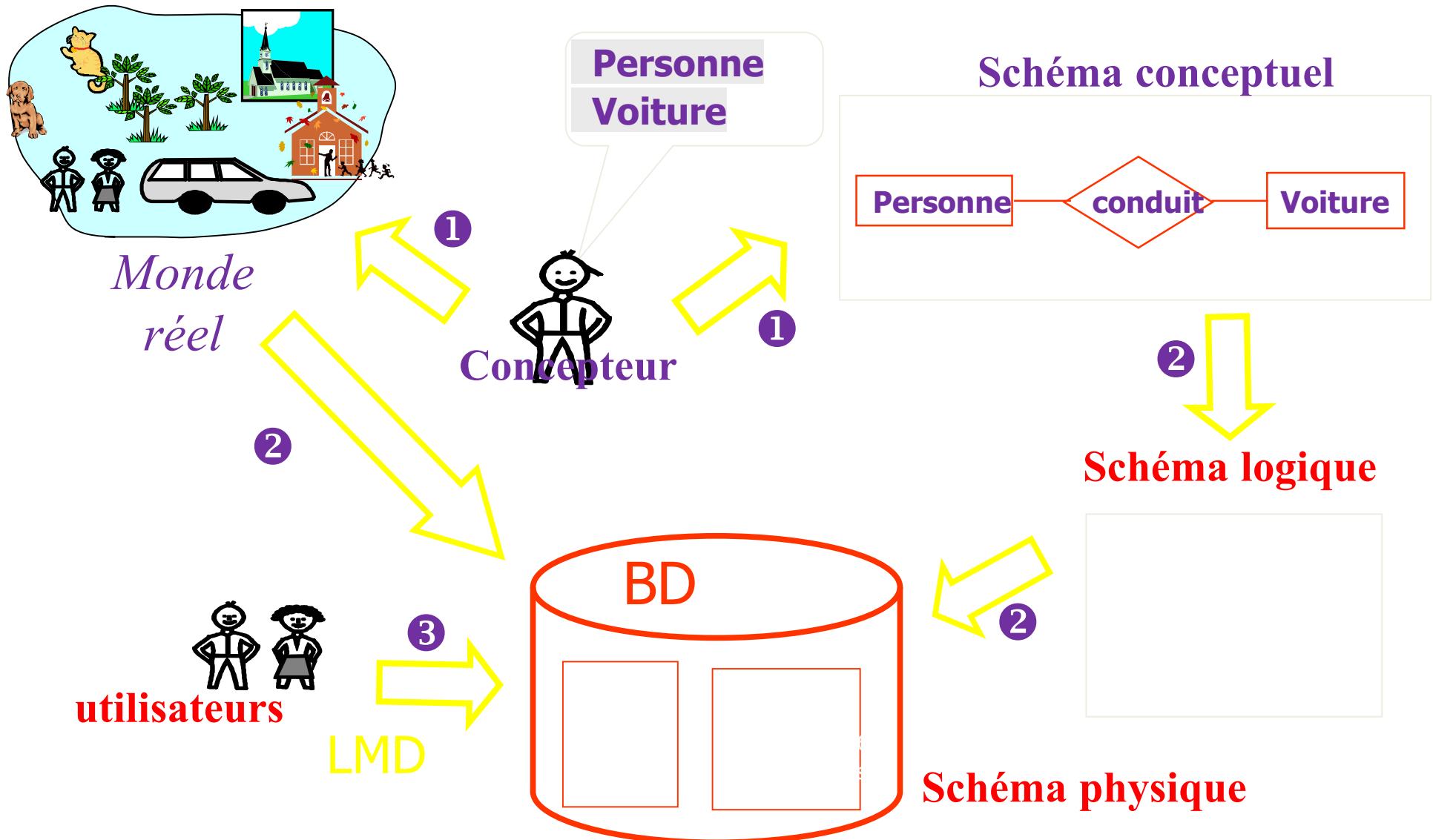
-> **Niveau conceptuel** : représentation logique du système étudié;

    C'est le cœur de nos préoccupations aujourd'hui

-> **Niveau externe** : ce que chaque utilisateur a le droit de faire et de savoir.

    Ces considérations étant annexes et « faciles », elles ne seront pas abordées ici.

# Cycle de vie d'une base de données



# **MODÈLE CONCEPTUEL DES DONNÉES (MCD)**

# MCD : Introduction

Connu sous les noms :

Modèle ou schéma **Entités/Associations (E/A)**

ou

Modèle **Conceptuel de Données (MCD, méthode MERISE)**

Il existe aussi d'autres normes qui sont utilisées, comme le diagramme de classe UML.

**En entreprise : on s'adapte à sa tradition.**

# MCD : Introduction

- > Cette méthode s'applique quelque soit le type de SGBD.
  - > Elle est non seulement vitale pour les gros problèmes, mais aussi très utile pour cerner les petits problèmes que l'on croît avoir compris (programmation).
  - > Elle est totalement indépendante du langage de programmation et de son implantation physique.
- => IL S'AGIT D'UNE **ETAPE CONCEPTUELLE** DANS LAQUELLE ON RECENSE LES **INTERACTIONS LOGIQUES ENTRE LES DONNÉES.**

# MCD : Recensement des informations

Il s'agit d'élaborer un **dictionnaire des informations** à prendre en compte.

C'est une démarche sans méthode stricte ni modèle réellement définis :

**=>on récolte de l'information et on classe.**

Deux moyens :

-> **Discuter avec les responsables et les intervenants** du domaine à considérer

=> savoir faire parler les gens ET LES ECOUTER  
(Technique de conduite d'entretiens)

-> **Lire des documents** : savoir lire entre les lignes;

Astuce : A chaque affirmation, se poser deux questions :

-> et alors ?

-> et sinon ?

*Remarque :* A chaque étape significative, il faut rédiger un document récapitulatif et, bien que cela ne soit pas toujours facile, essayer de le faire signer par la personne qui a fourni l'information ("Machin à Truc pour accord ...") et signer soi-même.

# MCD : Recensement des informations

On obtient alors une **liste d'informations** qu'il va falloir épurer :

- > On précisera les **polyséries** ("masse" signifie "poids" ou "gros marteau" ?);
  - > On éliminera les **synonymies**;
  - > On rencontrera des **informations élaborées** : il faudra alors énoncer la règle d'élaboration (pour être sûr qu'il s'agit effectivement d'informations élaborables);  
on conservera, ou non, ces informations selon l'urgence et la fréquence de leur emploi (on va peut-être conserver les informations nécessitant 3 heures de traitement !).
  - > On gardera les **informations élémentaires** (non élaborées) dans le modèle E/A.
- => *On attaque donc le modèle E/A avec une liste significative et épurée d'informations à prendre en compte.*

# MCD : Définitions

## **Entité :**

quelque chose de **générique** (pouvant prendre un ensemble de valeurs) dotée d'une existence propre. Elle peut être de deux types :

**concrète** : un livre, une personne, une pièce mécanique,...

**abstraite** : généralement l'explication d'un code.

**Remarque** : un livre est une entité générique dans le sens où il peut s'agir de

"Les Misérables" de Victor Hugo, de "Germinal" de Zola,...

# MCD : Définitions

## ***Association :***

représentation d'une association entre entités. On peut avoir à faire à :

- > 1 seule entité : un étudiant est en binôme avec un étudiant;
- > 2 entités : un étudiant a emprunté un livre;
- > 3 entités : un étudiant a emprunté un livre dans une certaine bibliothèque (si on tient compte de plusieurs bibliothèques).

Au-delà de 3 entités, il faut essayer de casser l'association en plusieurs petites :

"Un professeur fait cours dans telle matière dans telle salle pour telle promotion »

devient

"cette matière est enseignée dans telle salle pour telle promotion »

et

"un professeur fait cours dans telle salle"

à condition qu'ayant la salle et la promotion on ait la matière.

Le nombre d'entités est la *dimension de l'association*.

# MCD : Définitions

## **Attribut :**

donnée élémentaire relative à une entité ou une association, tel le nom de l'étudiant par exemple.

## **Instance (ou occurrence) d'une entité :**

combinaison unique de valeurs prises par les attributs d'une entité. Deux instances d'étudiants : Jacques Durand et Georges Martin. Autrement dit, une entité est une machine à cake, une instance est un cake. Encore dit d'une autre manière :

une entité est un formulaire vide, une instance un formulaire rempli.

## **Identifiant (ou clé primaire) :**

un ou plusieurs attributs (souvent artificiels comme un numéro) tels qu'il existe une bijection entre l'occurrence de l'entité et l'identifiant.

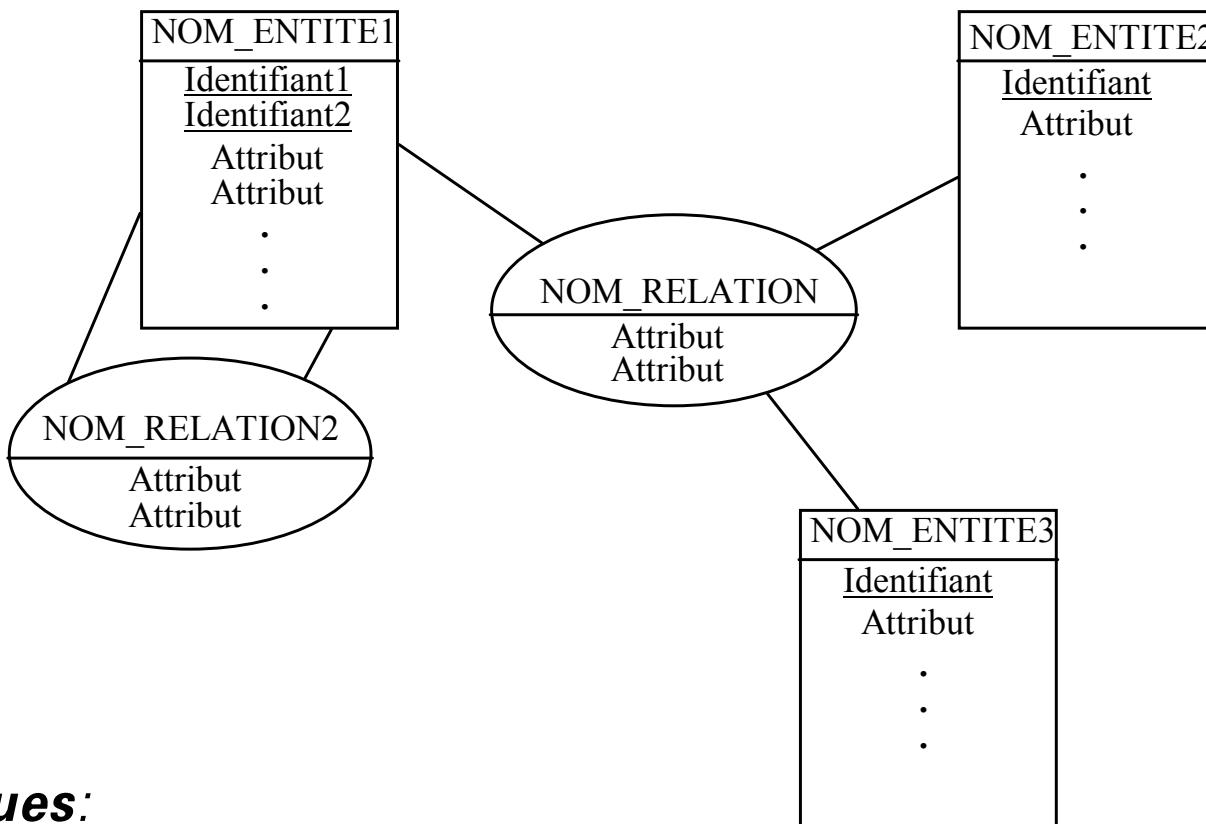
Je m'explique : la valeur de l'identifiant ne peut se retrouver qu'une seule fois parmi les occurrences de l'entité.

## **Occurrence d'une association :**

combinaison unique des occurrences intervenant dans l'association et de ses attributs propres. Ainsi nous avons trois occurrences :

- Martin a emprunté "Les Misérables" le 15-07-89;
- Durand a emprunté « Le seigneur des anneaux » le 25-12-93;
- Martin a emprunté "Les Misérables" le 07-08-94;

# MCD : Symbolisme



## ***Remarques:***

- l'entité NOM\_ENTITE1 a un identifiant composé de la réunion de deux attributs.
- NOM\_RELATION2 implique deux fois une seule entité.
- choisir des noms (d'entités, d'associations, d'attributs) significatifs, c-à-d éviter les termes trop généraux tels "est un" ou "a".

# MCD : règles

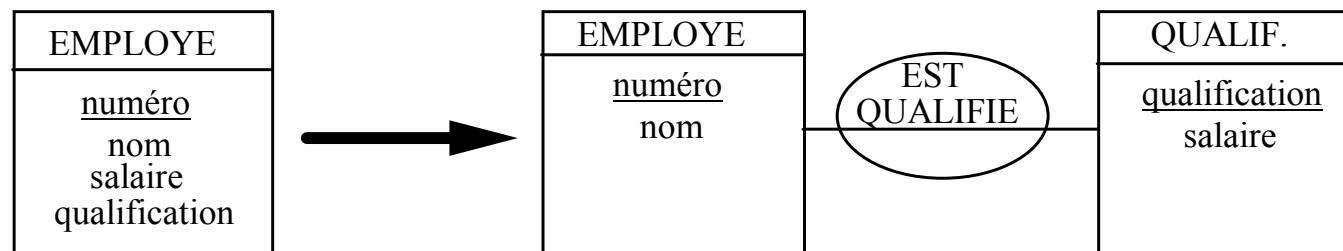
a) L'identifiant doit réellement être un identifiant; chaque entité doit avoir un identifiant.

La règle sur laquelle tout repose :

b) Un attribut est un scalaire, i.e. ce n'est pas une liste, ce n'est pas un sous objet.

chaque attribut doit une valeur pour toute occurrence, i.e. pas de case vide

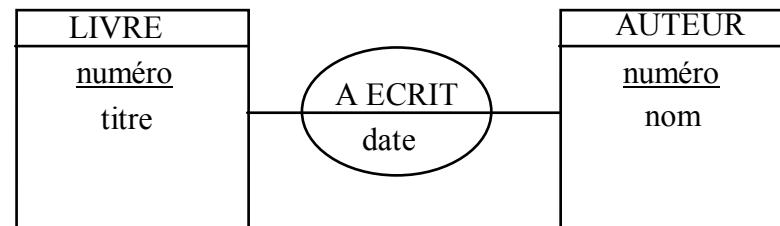
c) Les dépendances transitives entre attributs sont interdites :



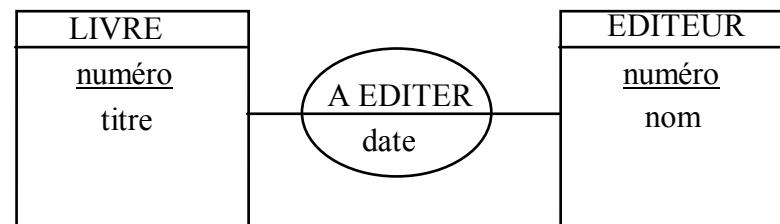
*à condition que la qualification implique un salaire.*

# MCD : règles

- d) Une association doit toujours mettre en relation le même nombre d'entités (il est interdit d'avoir une relation entre A, B et C qui, de temps en temps, ne relierait que A et B)
- e) Si les attributs d'une association peuvent être mis dans une entité supplémentaire en relation avec l'association, le faire.
- f) Si un attribut de l'association n'a de sens que pour un sous-ensemble des entités concerné, alors cet attribut fait partie du sous-ensemble concerné et non de l'association complète.



"date" ne doit pas être dans l'association : la date ne qualifie que le livre (un livre n'a été écrit qu'une fois). Par contre :



a un sens puisqu'un livre peut être édité plusieurs fois par le même éditeur à des dates différentes

# MCD : Cardinalités

Elles indiquent pour **chaque couple entité-association** le **nombre minimum et maximum d'occurrences de l'association** pouvant exister à un instant quelconque pour une occurrence de l'entité.

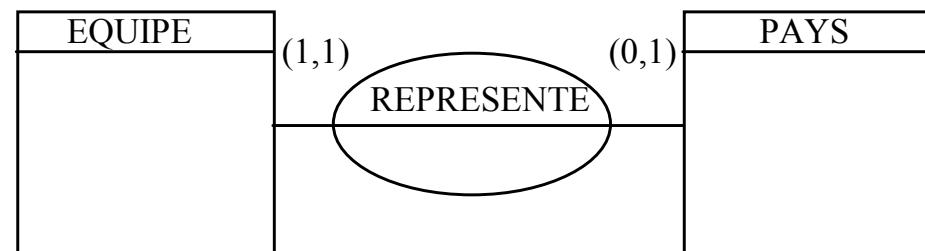
Elle se note (**mini, maxi**) sur chaque "branche" de l'association.

**Mini** : au moins

**Maxi** : au plus

Elles peuvent être du type (0,1), (0,n) (1,1) ou (1,n) mais jamais du type (n,n).

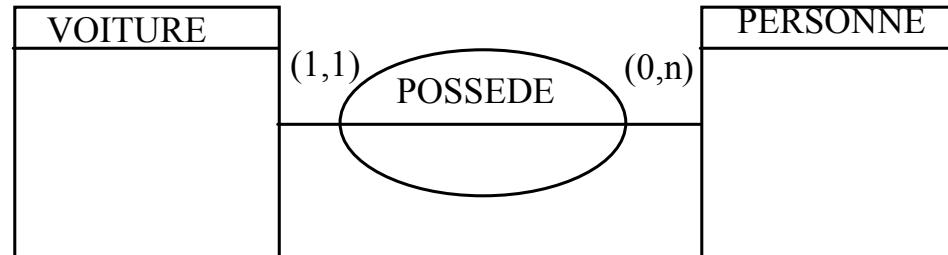
Exemple :



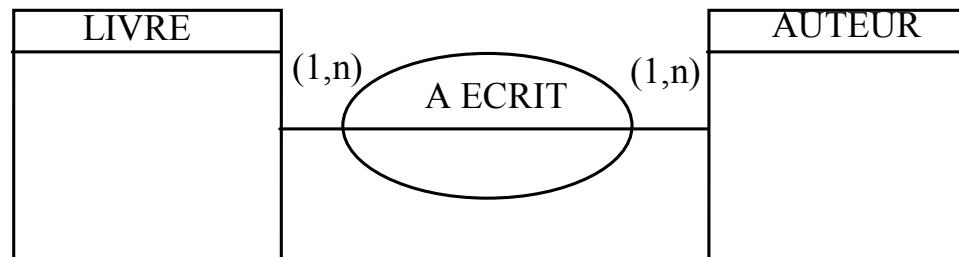
Une équipe représente **un et un seul** pays; un pays est représenté par **aucune ou une** équipe.

*Remarquer l'emploi du passif pour fixer les cardinalités*

# MCD : Cardinalités



Une voiture appartient à **une seule** personne, une personne peut avoir **aucune ou plusieurs** voitures.



Un livre a été écrit par **un ou plusieurs** auteurs, un auteur peut avoir écrit **un ou plusieurs** livres.

# MCD : recommandations

- 1) Répertorier l'ensemble des informations.
- 2) Epurer cet ensemble.
- 3) Trouver une entité concrète.
- 4) Y attacher tous les attributs.
- 5) Déterminer l'identifiant.
- 6) Si on trouve une information qui se rattache à cette entité, mais qui n'est pas un attribut alors on a une relation et éventuellement une nouvelle entité, on repart alors à l'étape 4 avec cette nouvelle relation et/ou entité.
- 7) Essayer de retourner en 3 avec les informations restantes.
- 8) Attacher au modèle toutes les informations restantes devant y être incluses (entités abstraites: explication de codage).
- 9) Toute information élaborée est-elle alors facilement élaborable ?
- 10) Fixer les cardinalités.
- 11) Les traitements prévus vont-ils pouvoir s'effectuer ?

**Remarque très importante :** Vérifier et revérifier le modèle E/A car une fois celui-ci réalisé, le reste du travail se fait quasi automatiquement.

# **MODÈLE LOGIQUE DE DONNÉES (MLD)**

# MLD ou le modèle relationnel

Le **modèle relationnel (ou MLD en Merise)** et les systèmes qui les utilisent s'imposent aujourd'hui sur le marché des bases de données : Oracle, SQLServer, Postgresql, MySQL, Access, ...

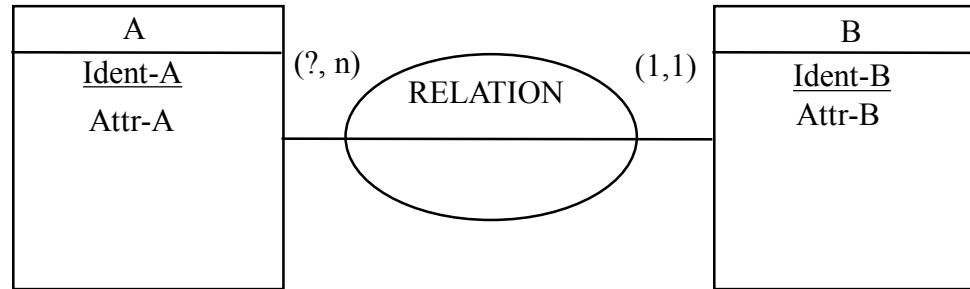
**C'est un modèle qui priviléie les RELATIONS.**

La transformation modèle E/A au modèle relationnel s'appuie sur

- > les associations,
- > leurs cardinalités,
- > sur le nombre d'entités concernées.

# Une ou deux entités

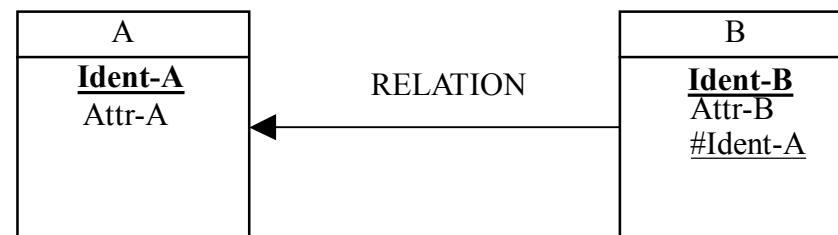
## a) cardinalités $(?, n)$ - $(1, 1)$



Connaissant B, je peux connaître directement le A qui est en relation avec lui puisqu'il n'y en a qu'un;  
par contre, connaissant A, pour connaître tous les B qui lui sont associés il faut que je parcoure toutes les occurrences de B.

Modélisation : A va donner une copie de son identifiant à B qui en fera un de ses attributs. On dit que A est le propriétaire de la relation et B le membre.

La relation est alors symbolisée par une flèche.

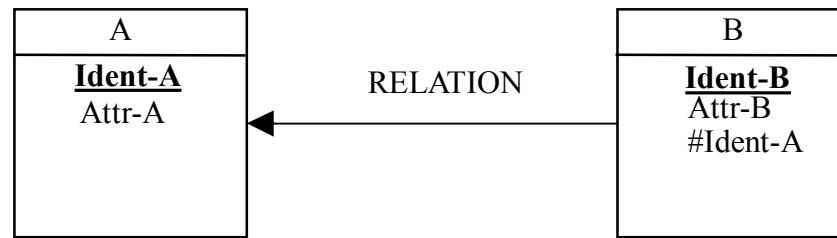


# Une ou deux entités

On peut en déduire le schéma relationnel avec une **représentation algébrique**, de la forme :

**Nom\_de\_l'entité(identifiant, attribut1, attribut2, #clé\_étrangère)**

Par exemple :



*devient*

A(Ident-A, Attr-A)

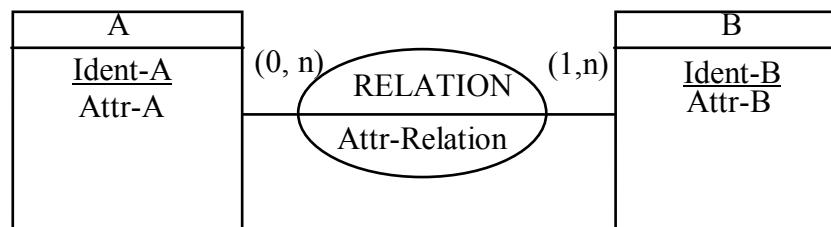
B(Ident-B, Attr-B, #Ident-A)

# Une ou deux entités

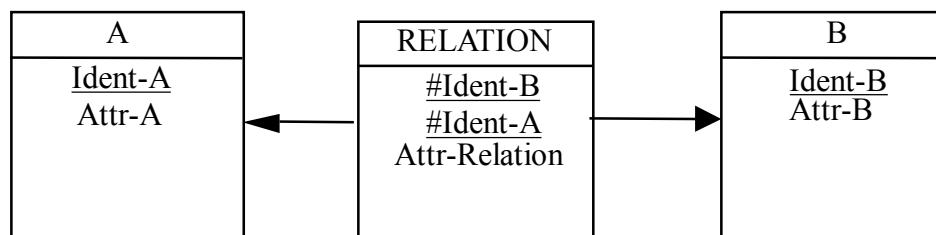
## b) cardinalités $(?,n)$ - $(0,1)$

Idem.

## c) cardinalités $(?,n)$ - $(?,n)$



On ne peut plus passer directement de B vers A. La relation devient une *entité-lien*. Elle conserve ses attributs et reçoit la copie des identifiants des entités concernées. L'identifiant de la relation devient alors l'ensemble de ses attributs.

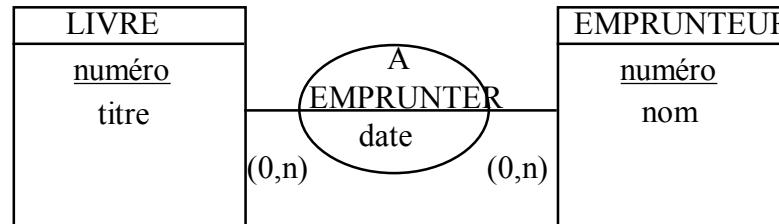


A(Ident-A, Attr-A)

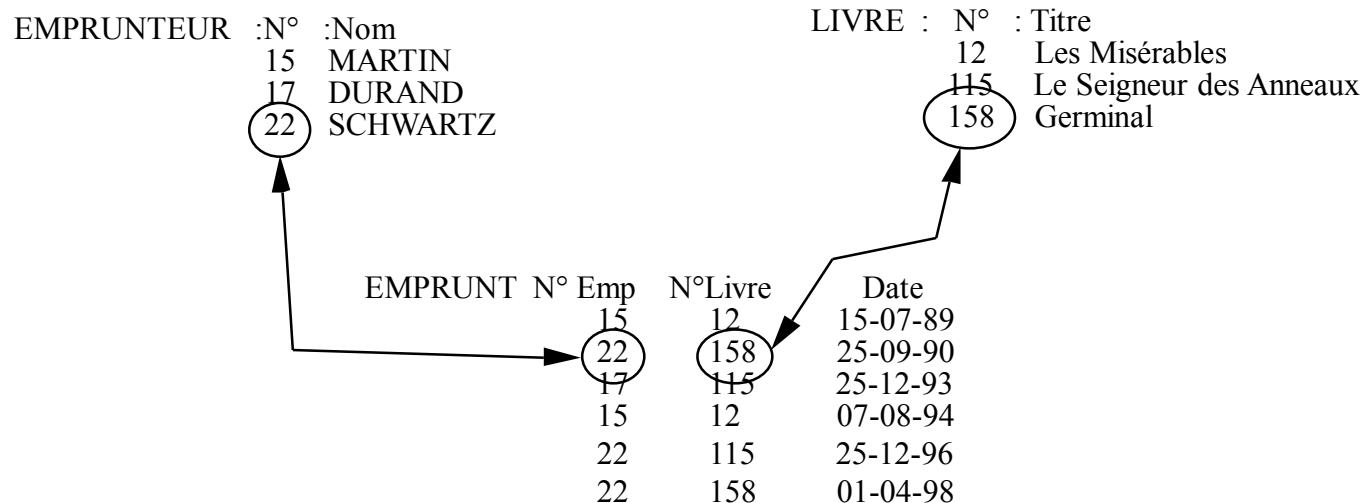
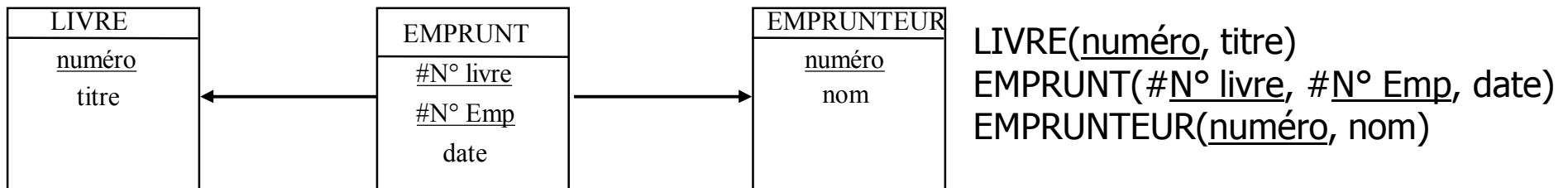
RELATION(#Ident-B, #Ident-A, Attr-Relation)

B(Ident-B, Attr-B)

# Une ou deux entités

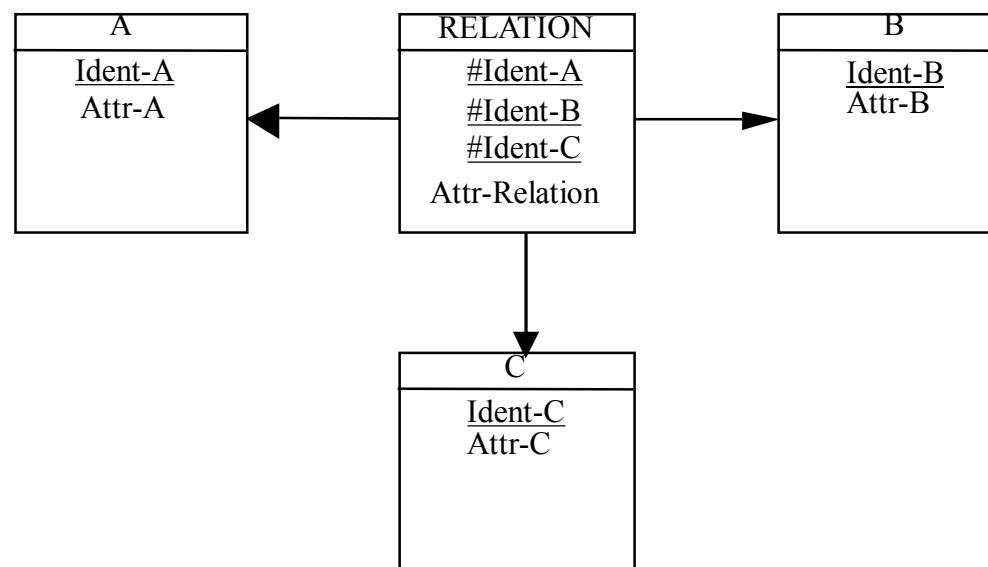
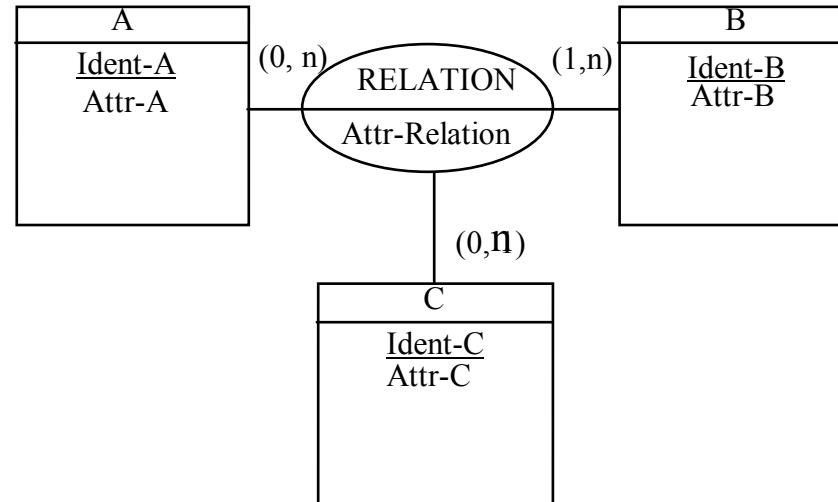


Généralement, on essaye de mettre le substantif du verbe de la relation



# Relation à plus de deux entités

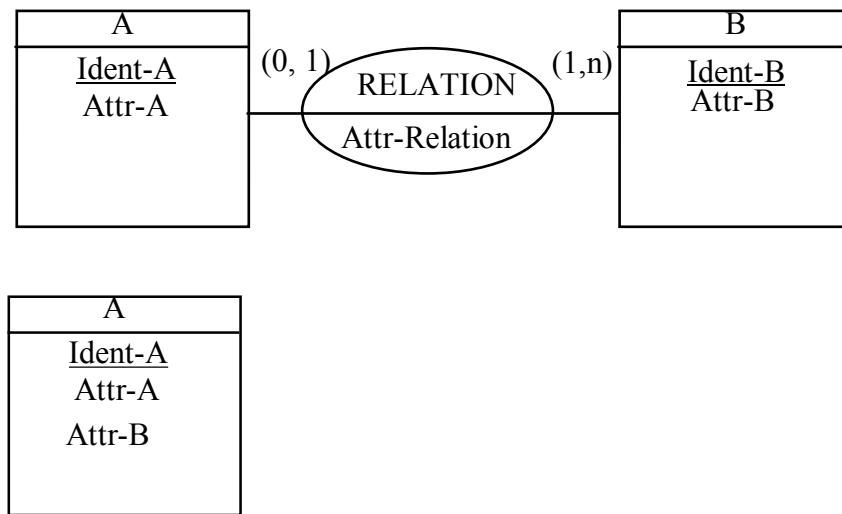
On procède comme pour le cas à cardinalité  $(?,n) - (?,n)$ ; c-à-d la relation devient une entité-lien et récupère les identifiants des entités concernées.



# Cas de simplification

Dans le cas suivant où une entité est toute petite, on peut s'autoriser à inclure les attributs de cette entité dans une autre.

Ex : B contient le nom marital, bien que tous les A n'en n'aient pas, on va préférer laisser à vide souvent cet attribut plutôt que d'avoir à gérer deux entités.

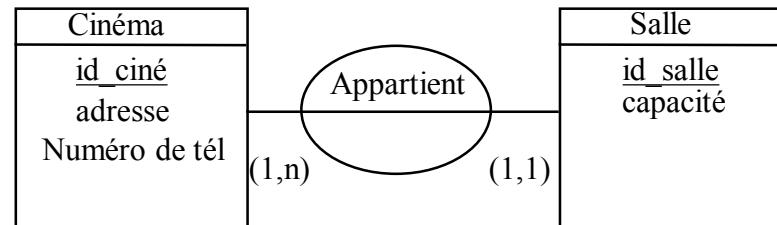


## Implantation :

Le modèle relationnel établi, chaque entité et entité-lien devient une table dont la structure de ses éléments est constituée de l'identifiant et des attributs de l'entité.

# Entité Faible

Il existe des cas où une entité ne peut exister qu'en étroite association avec une autre, et est identifiée relativement à cette autre entité. On parle alors **d'entité faible**.



Il est difficilement imaginable de représenter une salle sans qu'elle soit rattachée à son cinéma. C'est en effet au niveau du cinéma que l'on va trouver quelques informations générales comme l'adresse ou le numéro de téléphone.

On peut considérer alors qu'il est beaucoup plus naturel **de numérotter les salles par un numéro interne à chaque cinéma**.

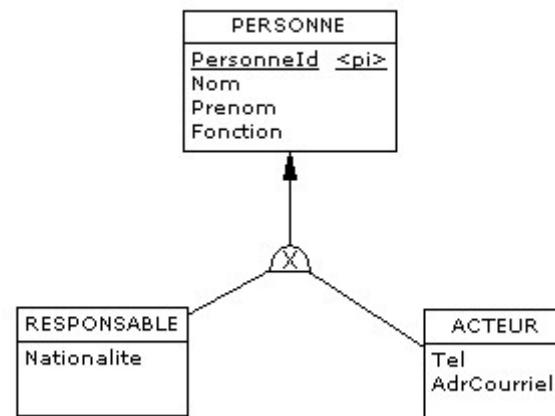
La clé d'identification d'une salle :

- la **clé de Cinéma**, qui indique dans quel cinéma se trouve la salle ;
  - le **numéro de la salle** au sein du cinéma.
- > l'entité salle ne dispose pas d'une *identification absolue*, mais d'une *identification relative* à une autre entité. Bien entendu cela force la salle à toujours être associée à **un et un seul** cinéma.

# Héritage

Les **sous-classes** RESPONSABLE et ACTEUR feront l'objet elles aussi d'entités-types, associées à l'entité-type PERSONNE par une **relation d'héritage**. Les entités-types RESPONSABLE et ACTEUR possèdent les mêmes attributs que les **sous-classes**.

*Exemple d'héritage avec un MCD*



Il y a plusieurs possibilités pour implémenter un héritage → au programme de Advanced DataBases en M1



# BASES DE DONNÉES - TI603

## INTRODUCTION À LA NORMALISATION RELATIONNELLE

Cours 2

*Karim LAHLOU*  
[abdelkrim.lahlou@efrei.fr](mailto:abdelkrim.lahlou@efrei.fr)

# Plan

- I. Introduction
- II. Dépendance fonctionnelle
- III. Formes normales (1FN – 2FN – 3FN)
- IV. Algorithmes de normalisation
- V. Conclusion

# I – Introduction

- La théorie de la normalisation permet de définir une méthode de conception de « bonnes » tables, c'est-à-dire sans redondance et sans perte d'information
- Exemple :

<u>NumPropriétaire</u>	Nom	Ville	<u>NumVéhicule</u>	Marque	Date
1000	AAAAA	PARIS	90FE75	PEUGEOT	10-sep-89
1500	BBBBB	NANTES	43XY97	RENAULT	02-fev-96
1000	AAAAA	PARIS	56GT98	FIAT	06-mar-91
1350	CCCCC	NICE	43ZT88	RENAULT	28-dec-87
1500	BBBBB	NANTES	57TG92	PEUGEOT	26-jui-91

- *Redondance* : on dit 2 fois que le propriétaire N°1500 a pour nom BBBBB et habite à Nantes

# *Pourquoi la normalisation ?*

- pour éliminer les redondances
- pour mieux comprendre les relations sémantiques entre les données
- pour éviter les incohérences de mise à jour
- pour éviter, autant que possible, les valeurs nulles

*Insertion d'une personne sans voiture  $\Rightarrow$  introduction de valeurs nulles*

- Pour éviter la perte d'information

*Suppression de la dernière voiture possédée par une personne  $\Rightarrow$  perte d'information*

# Exemple

- *Relation COURS*

Nomprof	Ville	Département	Nometud	Age	Nomcours	Note
Dupont	Lille	59	Alfred	22	Math	12
Dupont	Lille	59	Arthur	25	Math	05
Martin	Arras	62	Alfred	22	Anglais	18
Martin	Arras	62	Pierre	23	Anglais	11
Dupont	Lille	59	Pierre	23	Anglais	13
Charles	Lille	59	Pierre	23	Anglais	12

- *des données redondantes* : Dupont à Lille (59)
- *des risques d'incohérence* : déménagement de Dupont à Marseille
- *des valeurs nulles* : représenter un prof qui n'a pas d'étudiant entraînent des anomalies à l'interrogation
  - **Problème du choix des relations**

# Comment normaliser un schéma relationnel ?

- Approche par décomposition :
  - on part d'une table contenant tous les attributs
  - et on décompose jusqu'à ce qu'il n'y ait plus de redondances
- Approche par synthèse :
  - à partir de l'ensemble des attributs
  - et des dépendances fonctionnelles
  - on constitue les tables

# Exemple de table non normalisée

<u>NumPropriétaire</u>	Nom	Ville	<u>NumVéhicule</u>	Marque	Date
1000	AAAAA	PARIS	90FE75	PEUGEOT	10-sep-89
1500	BBBBB	NANTES	43XY97	RENAULT	02-fev-96
1000	AAAAA	PARIS	56GT98	FIAT	06-mar-91
1350	CCCCC	NICE	43ZT88	RENAULT	28-dec-87
1500	BBBBB	NANTES	57TG92	PEUGEOT	26-jui-91

# Exemple de normalisation par décomposition en utilisant les Dépendances Fonctionnelles

<u>NumPropriétaire</u>	Nom	Ville
1000	AAAAA	PARIS
1500	BBBBB	NANTES
1350	CCCCC	NICE

<u>NumVéhicule</u>	Marque
90FE75	PEUGEOT
43XY97	RENAULT
56GT98	FIAT
43ZT88	RENAULT
57TG92	PEUGEOT

<u>NumPropriétaire</u>	<u>NumVéhicule</u>	Date
1000	90FE75	10-sep-89
1500	43XY97	02-fev-96
1000	56GT98	06-mar-91
1350	43ZT88	28-dec-87
1500	57TG92	26-jui-91

# Décomposition sans perte

## Jointure

Soient R (A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>) et S (B<sub>1</sub>, B<sub>2</sub>, ..., B<sub>p</sub>) deux relations

La jointure de R et S est la relation T qui a pour attributs l'union des attributs de R et S et pour tuples l'ensemble des tuples construits à partir de R et S sur les valeurs identiques des attributs communs

On note T = R  $\bowtie$  S

## Exemple

R	Nom	Salaire
Dupond	10000	
Durand	5400	
Martin	12000	

S	Nom	Adresse
Dupond	Issy	
Durand	Sète	
Martin	Sète	

T	Nom	Salaire	Adresse
Dupond	10000	Issy	
Durand	5400	Sète	
Martin	12000	Sète	

## Définition

La décomposition de R en R<sub>1</sub>, R<sub>2</sub>, ..., R<sub>n</sub> est sans perte si, pour toute extension de R, on a :

$$R_1 \bowtie R_2 \bowtie \dots \bowtie R_n = R$$

## II – Dépendance fonctionnelle

- Soient
  - $R_1(A_1, \dots, A_n)$  un schéma relationnel
  - $X$  et  $Y$  sont deux sous-ensembles de  $\{A_1, \dots, A_n\}$
- On dit que :
  - $Y$  dépend fonctionnellement de  $X$  ou bien  $X$  détermine  $Y$ , on note  $(X \rightarrow Y)$ , si quelle que soit l'instance de  $R$ , pour tout tuples  $T_1, T_2$  de  $R$ , on a :

$$T_1[X] = T_2[X] \Rightarrow T_1[Y] = T_2[Y]$$

avec  $T_i[X]$  la valeur de  $X$  pour le tuple  $T_i$ .

# Exemple

- ◆ NumPropriétaire → Nom
- ◆ NumPropriétaire → Ville
- ◆ NumVéhicule → Marque
- ◆ NumPropriétaire, NumVéhicule → Date

Remarques :

- Une DF s'applique sur toutes les instances possibles
- Une DF doit être déclarée.....

# Axiomes d'Armstrong

- Propriétés des dépendances fonctionnelles
  1. Réflexivité  $Y \subseteq X \Rightarrow X \rightarrow Y$
  2. Augmentation  $X \rightarrow Y \Rightarrow X, Z \rightarrow Y, Z$
  3. Transitivité  $X \rightarrow Y$  et  $Y \rightarrow Z \Rightarrow X \rightarrow Z$
  - ...
- Conséquences
  4. Union  $X \rightarrow Y$  et  $X \rightarrow Z \Rightarrow X \rightarrow Y, Z$
  5. Pseudo-transitivité  $X \rightarrow Y$  et  $W, Y \rightarrow Z \Rightarrow W, X \rightarrow Z$
  6. Décomposition  $X \rightarrow Y$  et  $Z \subseteq Y \Rightarrow X \rightarrow Z$

# Dépendance fonctionnelle élémentaire

- Dépendance fonctionnelle élémentaire  
 $x \rightarrow A$  telle que
  - 1)  $A$  est un attribut unique
  - 2)  $A \notin x$
  - 3) Il n'existe pas  $x' \subseteq x$  tel que  $x' \rightarrow A$
- Dans la recherche des DF, on peut se limiter sans restriction aux DFs élémentaires

# Exemple

Table COURS (NOMPROF, VILLE, DEPARTEMENT, NOMETUDIANT, AGE, COURS, NOTE)

*Dépendances fonctionnelles valides :*

NOMPROF  $\rightarrow$  VILLE  
NOMPROF NOMETUDIANT  $\rightarrow$  VILLE  
NOMETUDIANT  $\rightarrow$  AGE

*Dépendances fonctionnelles invalides :*

AGE  $\rightarrow$  NOMETUD

*Dépendances fonctionnelles élémentaires*

NOMPROF  $\rightarrow$  VILLE  
VILLE  $\rightarrow$  DEPARTEMENT  
NOMPROF  $\rightarrow$  DEPARTEMENT  
NOMETUD  $\rightarrow$  AGE  
NOMETUD NOMCOURS  $\rightarrow$  NOTE  
NOMCOURS  $\rightarrow$  NOMPROF

..

# Autres définitions

$X \rightarrow Y$  est triviale

$Y$  est un sous-ensemble de  $X$

$A, B \rightarrow B$  est une DF triviale

$X \rightarrow Y$   
est élémentaire  
et  
non triviale

$Y \subset X$  et  $\forall X' \subset X, X' \neq Y$

Ex : NumPropriétaire, NumVéhicule  $\rightarrow$  Date

# Autres définitions

- Fermeture d'un ensemble  $F$  de DFs :
  - Ensemble  $F'$  de DFs obtenu par applications successives des axiomes d'inférence
- Fermeture transitive d'un ensemble  $F$  de DFs :
  - Ensemble  $F^+$  de DFs élémentaires obtenues par application des axiomes de transitivité et de pseudo-transitivité
- Couverture minimale d'un ensemble  $F$  de DFs :
  - Plus petit ensemble de DFs permettant d'obtenir, par applications successives des axiomes d'inférence, la fermeture transitive de  $F$

# Exemple

*Exemple : Relation COURS*

(NOMPROF, VILLE, DEPARTEMENT, NOMETUDIANT,  
AGE, COURS, NOTE)

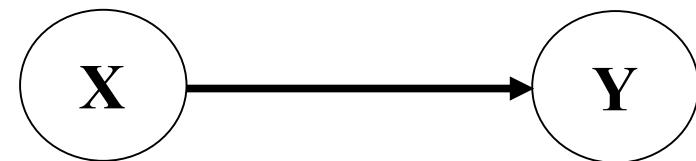
1. NOMPROF → VILLE
  2. VILLE → DEPARTEMENT
  3. NOMPROF → DEPARTEMENT
  4. NOMETUDIANT → AGE
  5. NOMETUDIANT, COURS → NOTE
  6. COURS → NOMPROF
- } n'est pas minimal car 3 est redondante

{1, 2, 4, 5, 6} est une couverture minimale

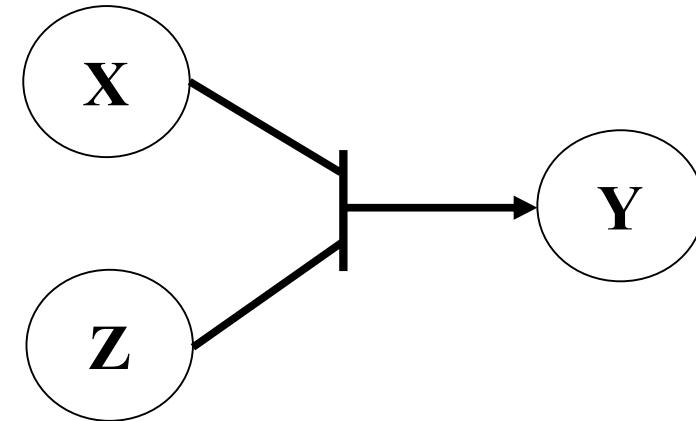
{1, 3, 4, 5} n'est pas une couverture

# Graphe de dépendance

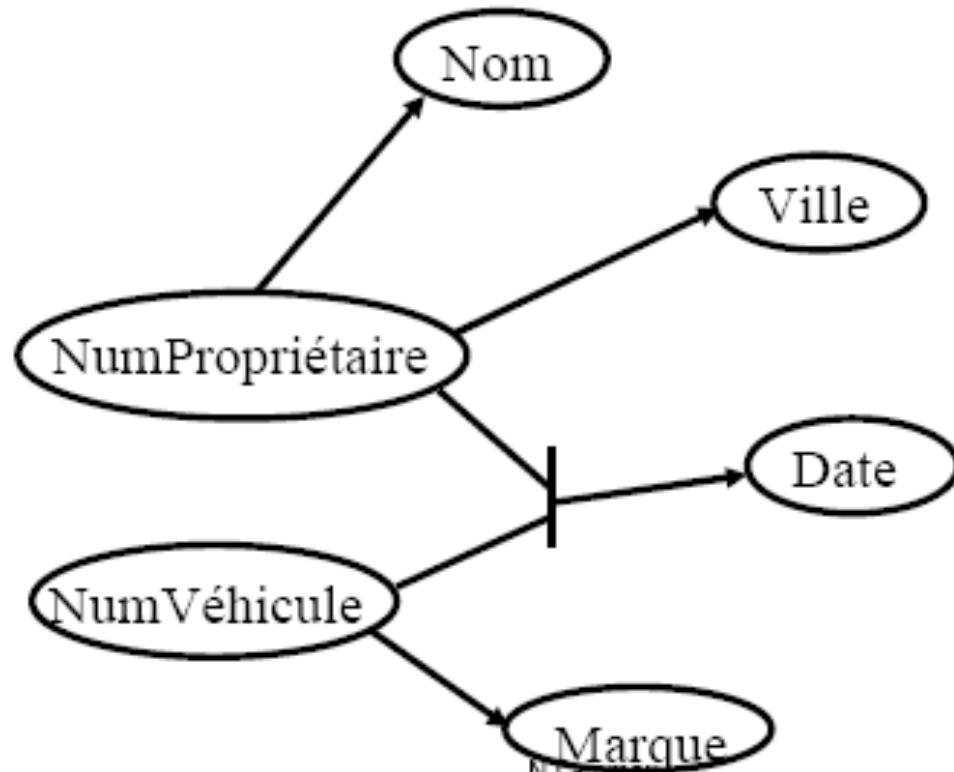
- $X \rightarrow Y$



- $X, Z \rightarrow Y$



# Exemple



# Dépendance fonctionnelle et Clé

◆ Soient :

$R(A_1, A_2, \dots, A_n)$

$X \subseteq \{A_1, A_2, \dots, A_n\}$

◆ On dit que  $X$  est une clé candidate de  $R$  ssi :

-  $X \rightarrow A_1, A_2, \dots, A_n$

-  $\forall Y \subset X, Y \not\rightarrow A_1, A_2, \dots, A_n$

# Exemple

R (NumPropriétaire, NumVéhicule, Nom, Ville, Marque, Date)

- NumPropriétaire → Nom
- NumPropriétaire → Ville
- NumVéhicule → Marque
- NumPropriétaire, NumVéhicule → Date



**{NumPropriétaire, NumVéhicule} est la seule clé pour R**

# Les Formes Normales

- 1ère Forme Normale (1FN)
- 2ème Forme Normale (2FN)
- 3ème Forme Normale (3FN)
- Etc, ....

# Première forme normale (1FN)

- *Une relation est en 1ère Forme Normale (1FN) si et seulement si tous ses attributs sont atomiques (non composés et mono-valués)*
- *Contre-exemples :*
  1. PERSONNE (NOM, PRENOMS)  
Mise en 1FN : PERSONNE1 (NOM, PRENOM1, PRENOM2)
  2. PERSONNE (NOM, PRENOM, ADRESSE)  
Mise en 1FN : PERSONNE2 (NOM, PRENOM, N°RUE, RUE, CODEPOSTAL, VILLE)

# Exemple

**ETUDIANT ( Matricule, Nom ,...., DIPLOMES)**

01	A	....	{Bac, BTS}
02	B	....	{Bac, Deug}
03	C	....	{Bac}

**ETUDIANT n'est pas en 1FN**

**ETUDIANT ( Matricule, Nom ,...., DIPLOME)**

01	A	....	Bac
01	A	....	BTS
02	B	....	Bac
02	B	....	Deug
03	C	....	Bac

**ETUDIANT est en 1FN**

## Deuxième forme normale (2FN)

- *Une relation est en deuxième forme normale (2FN) si :*
  1. elle est en 1 FN
  2. tout attribut n'appartenant pas à la clé dépend uniquement de la totalité de la clé
- *Exemple :*
  - **R (A, B, C, D) en 1FN et A → C ⇒ R n'est pas en 2FN**

# Exemples de relations non 2FN

R (NumPropriétaire, NumVéhicule, Nom, Ville, Marque, Date)

NumPropriétaire → Nom

NumPropriétaire → Ville

NumVéhicule → Marque

NumPropriétaire, NumVéhicule → Date

## **R n'est pas en 2FN**

*Exemple :* COURS (NOMPROF, VILLE, DEPARTEMENT, NOMETUD, AGE, NOMCOURS, NOTE)

1 seule clé (NOMETUD, NOMCOURS)

*Les DF :*

1. NOMPROF ↣ VILLE	4. NOMETUD, NOMCOURS ↣ NOTE
2. VILLE ↣ DEPARTEMENT	5. NOMCOURS ↣ NOMPROF
3. NOMETUD ↣ AGE	

Problème pour les attributs NOMPROF, VILLE, DEPARTEMENT et AGE

COURS (NOMETUD NOMCOURS NOTE)

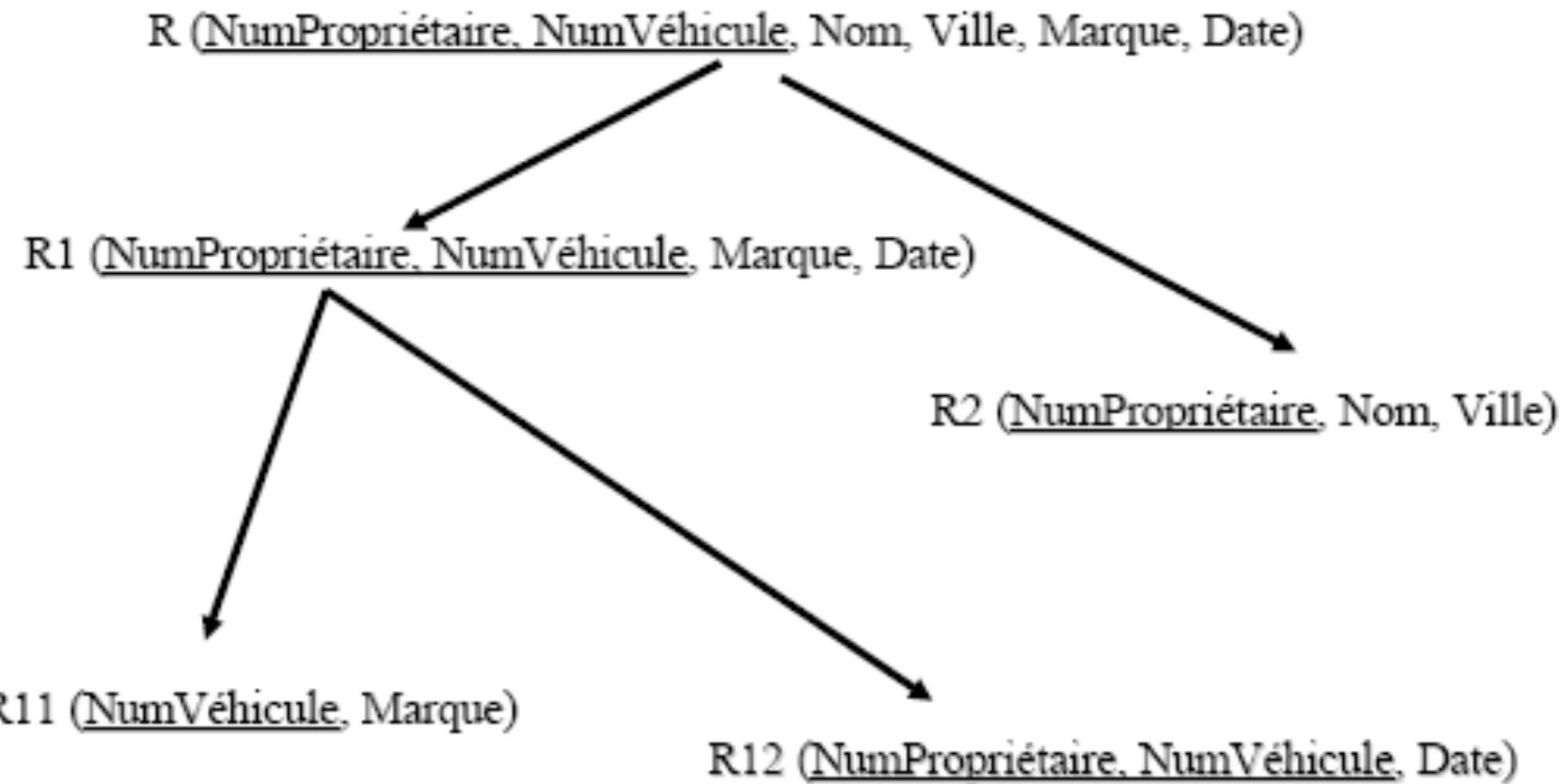
R1 (NOMCOURS NOMPROF VILLE DEPARTEMENT)

R2 (NOMETUD AGE)

}

sont en 2 FN

# Exemple 2FN par décomposition



# Troisième forme normale (3FN)

- Une relation est en troisième forme normale (3FN) si:
  1. elle est en 2FN
  2. tout attribut n'appartenant pas à une clé ne dépend pas d'un attribut non clé (pas de dépendance fonctionnelle entre attributs non clés)
- ‘ R (A, C, D) en 2FN et C → D ⇒ R n'est pas en 3FN

# Exemple

PRODUIT (NunProduit, Désignation, CodeTVA, TauxTVA)

CodeTVA → TauxTVA



PRODUIT (NunProduit, Désignation, CodeTVA)

TVA (CodeTVA, TauxTVA)

# Algorithme de mise sous 3 FN

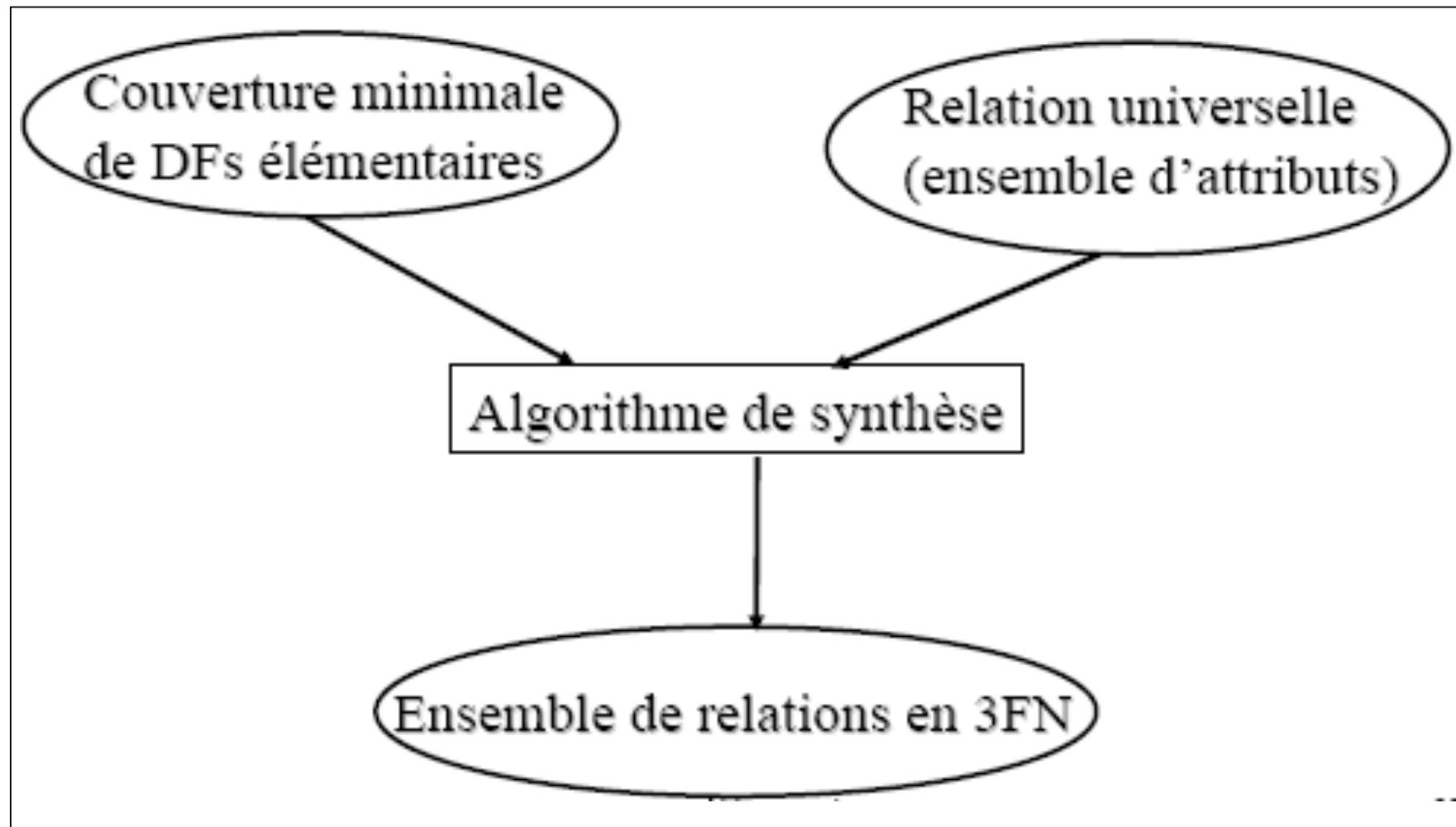
- $0FN \Rightarrow 1FN$  : mise sous forme atomique des attributs
- $1FN \Rightarrow 2FN$  : pour chaque partie X de clé déterminant des attributs non clés  $Y_1, \dots, Y_n$ 
  1. on crée une relation supplémentaire avec X pour clé et  $Y_1, \dots, Y_n$  comme attributs non clés
  2. on retire  $Y_1, \dots, Y_n$  de la relation initiale
- $2FN \Rightarrow 3FN$  : pour chaque attribut non clé Y déterminant des attributs non clés  $Z_1, \dots, Z_n$ 
  1. on crée une relation  $R'$  supplémentaire avec Y comme clé et  $Z_1, \dots, Z_n$  comme attributs non clés
  2. on retire  $Z_1, \dots, Z_n$  de la relation initiale

*$R'$  n'est pas nécessairement en 3 FN. Si c'est le cas, réitérer le processus sur  $R'$ .*

# Propriétés

- Dans une décomposition d'une relation en plusieurs autres, on dit que la décomposition préserve une dépendance fonctionnelle s'il reste, après décomposition, une relation contenant tous les attributs de la DF
- Propriété :  
Toute relation a au moins une décomposition en 3 FN qui :
  - préserve une couverture minimale de DF
  - est sans perte

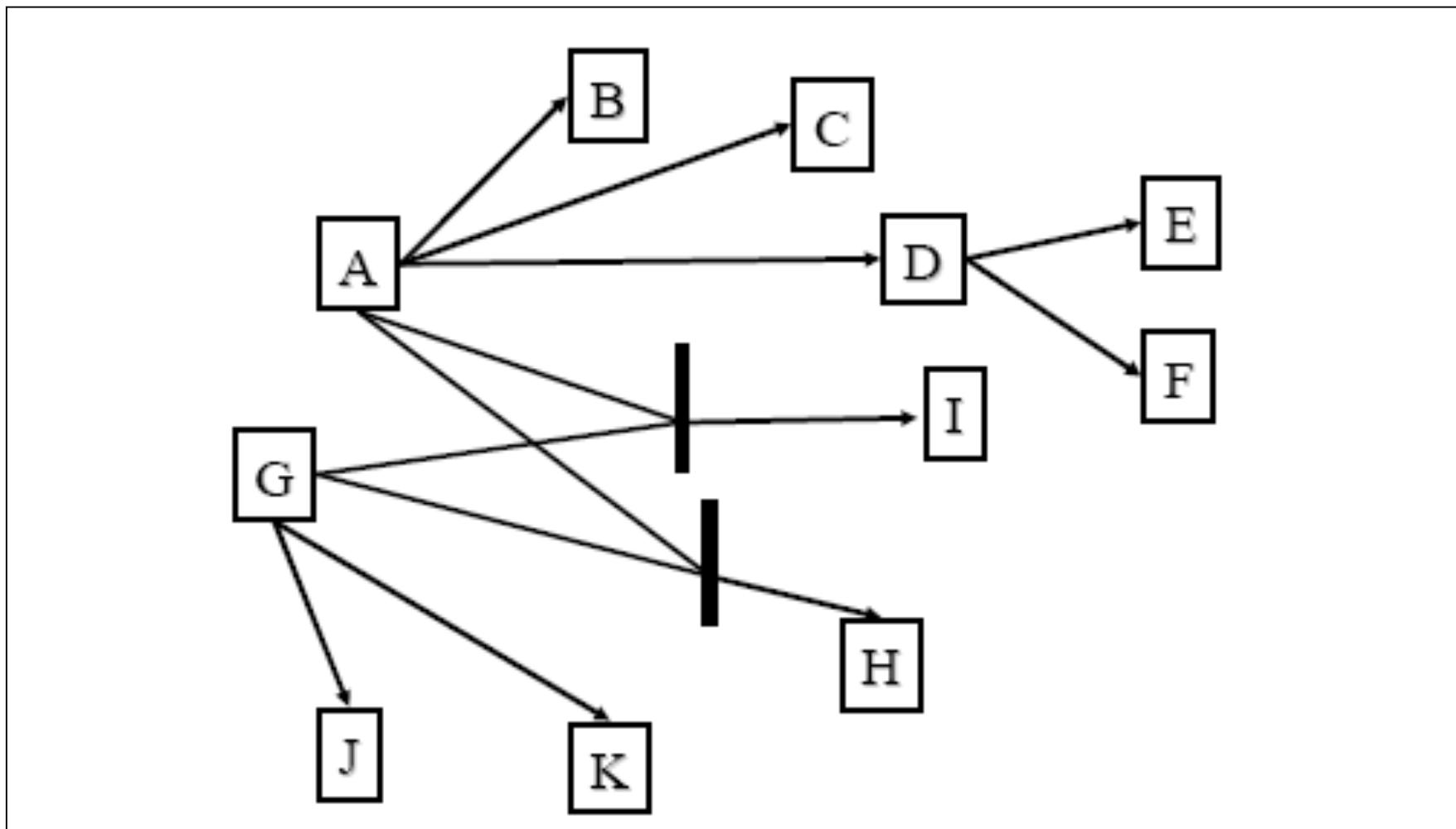
# Autre algorithme de normalisation



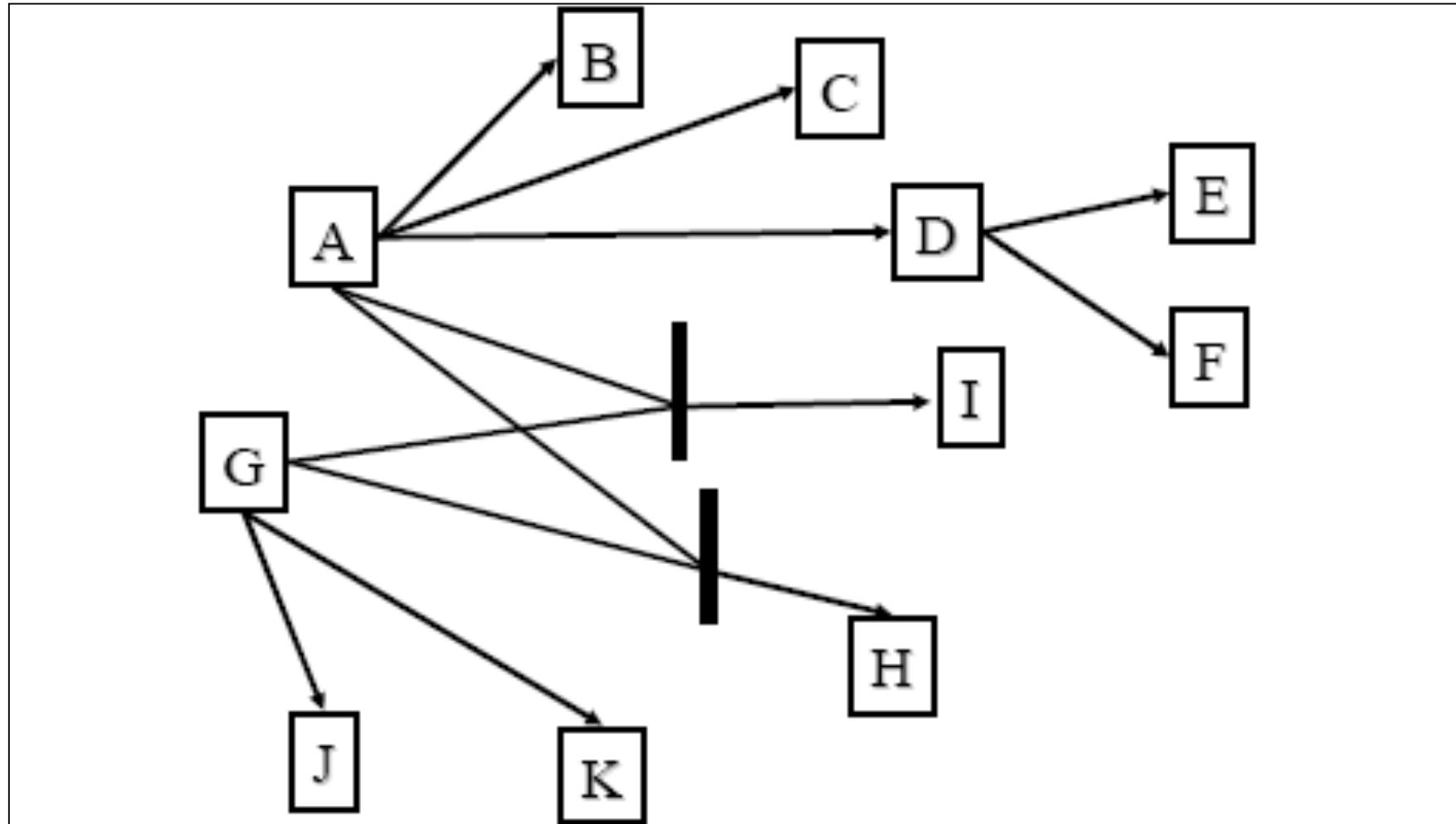
# Etapes de l'Algorithme de Synthèse

1. Regroupement des dépendances de même partie gauche
2. Construction d'une relation pour chaque ensemble
  - Chacune des relations a pour clé le groupe d'attributs en partie gauche

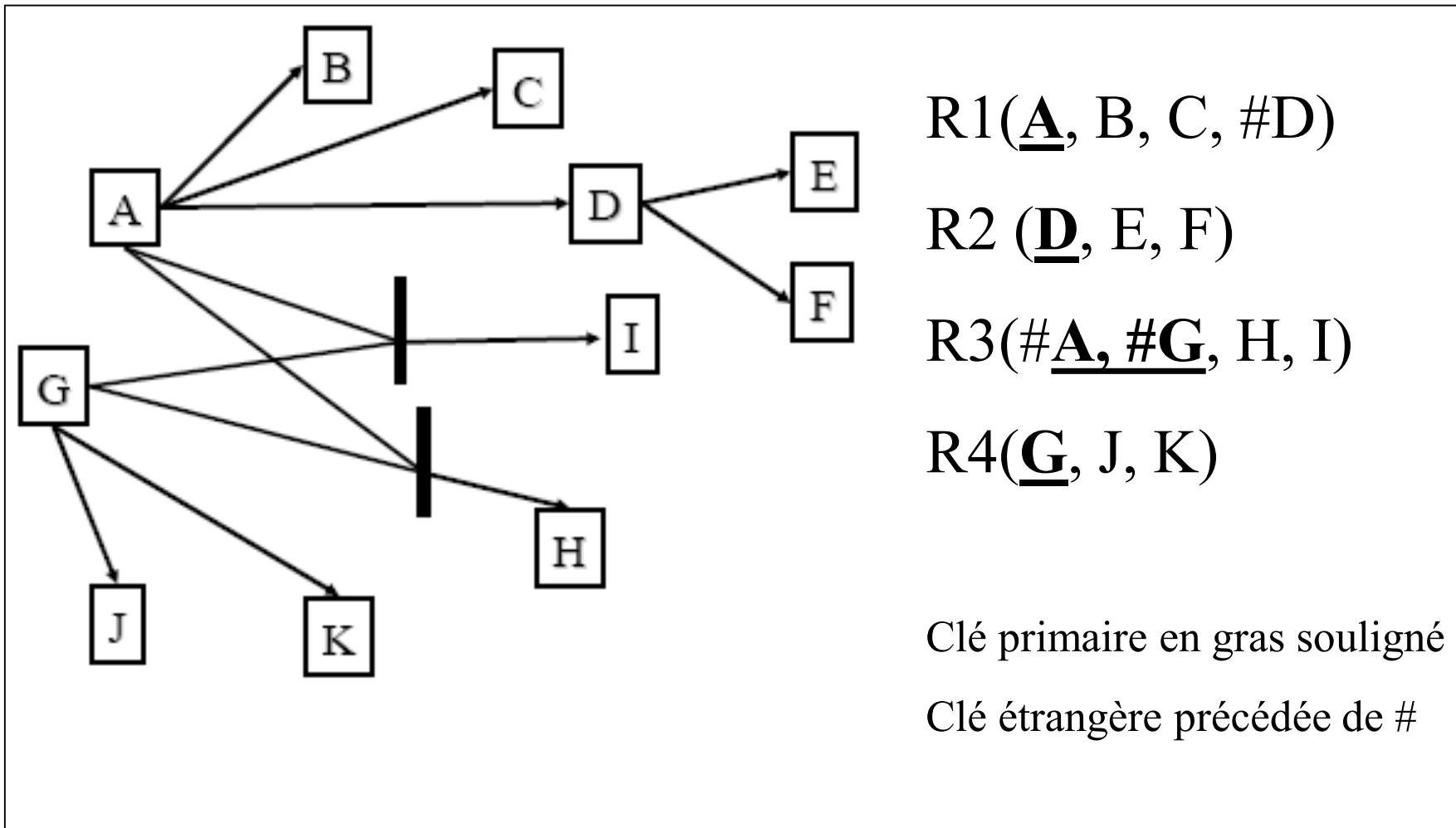
# Exemple



# Application de l'étape 1 de l'algorithme



# Application de l'étape 2 de l'algorithme



# Comparaison des 2 algorithmes

- ◆ Algorithme de décomposition :
  - préserve le contenu
  - Conduit à des relations en au moins 3FN
- ◆ Algorithme de synthèse
  - préserve les DFs
  - conduit à des relations en 3FN
- ◆ NB : une décomposition de R en R1, R2, ...Rn préserve le contenu ssi la jointure des relations de R1, R2, ...Rn est égale à la relation R

# CONCLUSION

- La normalisation permet de :
  - Construire des tables sans redondance
  - Vérifier la bonne conception des tables issues de la modélisation conceptuelle
  - Restructurer une base existante

## Partie III : Concepts avancés en bases de données

- Vues et droits d'accès
- Transactions, résistance aux pannes et concurrence d'accès
- Procédures stockées et déclencheurs



## VUES ET DROITS D'ACCÈS

VUES



# Vue : Définition

- Une vue est une **table virtuelle** au sens où ses instances n'existent pas physiquement.
- Une vue est une **table logique** pointant sur une table physique.
- Un utilisateur peut suivre l'évolution d'une table physique via une vue.
- Chaque appel à une vue correspond à l'exécution d'une requête SELECT.

# Vue : Avantages

- **Optimisation** : donner un nom à une requête longue pour l'utiliser souvent.
- **Simplification** : réduire des tables complexes à des ensembles de vues plus simples.
- **Sécurité et confidentialité** : masquer certaines données (lignes ou colonnes) aux utilisateurs.

# Vue : Type

Les vues sont manipulées, interrogées et mises à jour comme n'importe quelle BD conceptuelle (tables), mais cela dépend de l'implémentation choisie :

- 1. Vues virtuelles**
- 2. Vues matérialisées**

# Vues virtuelles

- Les relations de la vue ne sont pas stockées seule sa définition est stockée
- Le SGBD doit traduire les requêtes et mises à jour sur la vue en requêtes et mises à jour sur la BD conceptuelle

# Vues matérialisées

- Stockées physiquement (ex: entrepôts de données)
- Pour des raisons de performances, on peut avoir intérêt à volontairement enregistrer le résultat de la vue, on parle alors de vue matérialisée.

*CREATE MATERIALIZED VIEW...*

- Attention à la taille des vues matérialisées qui peut être conséquente (en présence de jointure)

# Création d'une vue

*CREATE [OR REPLACE] [TEMP] VIEW nom  
[(nom\_colonne [...])]  
AS requête  
[WITH [CASCADED | LOCAL] CHECK OPTION]*

- **temp** : supprimée en fin de session
- **check option** : les conditions de la création doivent être respectées lors des INSERT et des UPDATE
- **local** : uniquement sur cette vue
- **cascaded** : sur toutes les vues-filles

# Exemple simple

**CREATE VIEW** comedies **AS**

**SELECT \***

**FROM** films

**WHERE** genre = 'Comédie';

**SELECT \***

**FROM** comedies

**WHERE** sortie = 2010;

# Exemple avec jointure

**CREATE VIEW tous AS**

**SELECT e.nom as Employe, d.nom as departement**

**FROM Employes e, Departement d**

**WHERE e.departement = d.id**

**SELECT \* FROM tous;**

# Exemple avec héritage

**CREATE VIEW tous AS**

**SELECT e.nom as Employe,**

**d.nom as Departement**

**FROM Employes e, Departement d**

**WHERE e.departement = d.id**

**CREATE VIEW les\_dupont AS**

**SELECT \***

**FROM tous**

**WHERE Employe = 'Dupont';**

# Modification d'une vue

- la vue doit être sans jointure
- pas de renommage des colonnes
- pas d'opérateur d'agrégation

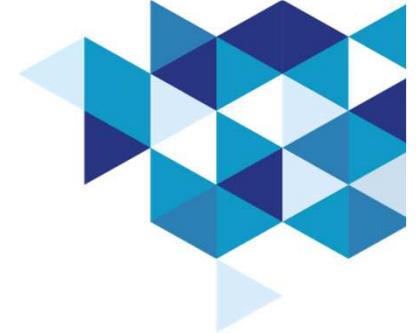
# Suppression d'une vue

*DROP VIEW nom [...]  
[CASCADE | RESTRICT]*

- **Cascade** : supprime aussi les objets qui dépendent de la vue
- **restrict** : refuse de supprimer la vue si un objet en dépend (valeur par défaut)

# Renommer une vue

```
RENAME ancien_nom  
TO nouveau_nom;
```



# VUES ET DROITS D'ACCÈS

droits d'accès

# Pourquoi des droits ?

- **De nombreuses menaces**
  - Omniprésence des bases de données
  - Informations de valeur croisées ou non
  - Utilisées par des prestataires externes
- **Des négligences**
  - protection des bases de données
  - les SGBD sont les 1er cibles d'attaques
  - 49% des attaques sont internes

# Droits et SGBD

Dans les SGBD, il existe un système d'autorisation et de protection de la BD contre des accès non autorisés.

- Le système de gestion de droits à deux principes :
  - Accorder des droits
  - Révoquer des droits

# Droits / Privilèges

- Niveau objets
  - objet = table, vue, fonction, procédures
  - des droits sur les objets (privilèges)
- Niveau utilisateurs (cf. systèmes) :
  - Utilisateur **connect** :
  - utiliser les tables de la BD,
  - créer des vues,
  - transmettre des droits
  - Utilisateur **resources** :
  - connect + créer des tables
  - Utilisateur **DBA** :
  - resources + créer des utilisateurs

# Droits sur les objets

	Tables	Vues	Fonctions
SELECT	X	X	
UPDATE	X	X	
DELETE	X	X	
INSERT	X	X	
ALTER	X		
REFERENCES	X		
EXECUTE			X

# Droits sur les tables

## **GRANT**

```
{SELECT | INSERT | UPDATE | DELETE |  
RULE | REFERENCES | TRIGGER | ALL  
[PRIVILEGES]}  
ON [TABLE] ma_table [...]  
TO {user | GROUP name | PUBLIC} [...]  
[WITH GRANT OPTION]
```

# Exemples simples

**GRANT**

**SELECT**

**ON client TO PUBLIC**

**GRANT**

**INSERT**

**ON TABLE client TO Alice**

**GRANT**

**UPDATE, DELETE**

**ON TABLE client TO Bob**

**WITH GRANT OPTION # Héritage de droits**

# Exemples fins

**GRANT**

INSERT, UPDATE (nom, adresse)

**ON TABLE** client **TO** Alice

*# La clé doit être insérée par un déclencheur*

*# Pas de privilèges sur une colonne via SELECT*

**CREATE VIEW** personne **AS**

SELECT nom, adresse

FROM client

**GRANT SELECT ON** personne **TO** Bob

# Droit de créer des tables

**GRANT {**

{CREATE | TEMPORARY | TEMP } [,...]

| ALL [PRIVILEGES]}

**ON DATABASE ma\_base [,...]**

**TO {user | GROUP name | PUBLIC } [,...]**

[WITH GRANT OPTION]

# Droit super-utilisateur

- Utilisateur de niveau **DBA**
- Dans la plupart des systèmes, le super-utilisateur par défaut est celui qui a créé la base de donnée.
- Un super-utilisateur a deux droits :
  - Créer des utilisateurs
  - Créer des bases de données

# Création d'utilisateur

**CREATE USER nom [[WITH] option [...]]**

**Options :**

SYSID uid (*choisir l'identifiant*)

- | CREATEDB | NOCREATEDB
- | CREATEUSER | NOCREATEUSER
- | IN GROUP nomgroupe [,...]
- | [ENCRYPTED | UNENCRYPTED] PASSWORD  
mdp
- | VALID UNTIL temps\_absolu

# Exemples

- Un utilisateur sans mot de passe

CREATE USER Alice;

- Un utilisateur avec un mot de passe

CREATE USER Paul WITH PASSWORD 'jw8s0F4';

- Un utilisateur avec un mot de passe valide jusqu'à la fin 2018 (*après 1 seconde dans 2019, il est invalidé*)

CREATE USER Claire WITH PASSWORD 'jw8s0F4'  
VALID UNTIL '2018-01-01';

# Exemples avancés

- Un utilisateur pouvant créer des bases de données  
`CREATE USER manuel WITH PASSWORD  
'jw8s0F4' CREATEDB;`
- Un utilisateur pouvant créer des utilisateurs  
`CREATE USER manuel WITH PASSWORD  
'jw8s0F4' CREATEUSER;`

# Utilisation de rôles

- **À la base** : il faut créer des groupes puis les associer aux utilisateurs.
- **Maintenant** : il faut créer des rôles puis les faire hériter aux utilisateurs (*un utilisateur est un rôle particulier*).

# Exemples de rôles

- **GROUPES**

**CREATE GROUP** vendeur **WITH** Alice, Bob;

**GRANT SELECT ON** client **TO GROUP** vendeur;

**CREATE** Claire **IN GROUP** vendeur;

- **ROLES**

**CREATE ROLE** vendeur

**GRANT SELECT ON** client **TO GROUP** vendeur;

**CREATE ROLE** Alice **WITH INHERIT IN ROLE** vendeur;

# Révocation

## Suppression de droits

- Mis en place par un utilisateur
- Le **pouvoir de révocation est limité par les autorisations** de l'utilisateur en question.

## Exemples :

- Le super-utilisateur peut faire toute les révocations qu'il souhaite.
- Paul qui a offert le droit d'INSERT sur une table X à Alice ne peut que lui retirer ce droit (et pas un autre).

# Supprimer des droits

```
REVOKE [ GRANT OPTION FOR ]  
{{SELECT | INSERT | UPDATE | DELETE | RULE |  
REFERENCES | TRIGGER} | ALL [PRIVILEGES]}  
ON [ TABLE ] nom_table [, ...]  
FROM {user | GROUP name | PUBLIC} [, ...]  
[CASCADE | RESTRICT]
```

# Révoquer la création de table

**REVOKE [GRANT OPTION FOR]**

**{{CREATE | TEMPORARY | TEMP } [,...]**

**| ALL [PRIVILEGES]}**

**ON DATABASE ma\_base [,...]**

**FROM {user | GROUP name | PUBLIC} [,...]]**

**[CASCADE | RESTRICT]**

# Supprimer le droit d'administration

```
REVOKE [ADMIN OPTION FOR]
rôle [...] FROM utilisateur [...]
[CASCADE | RESTRICT]
```

# Exemples

- Retire au groupe PUBLIC le droit d'insertion dans la table FILMS :

`REVOKE INSERT ON films FROM PUBLIC;`

- Retire tous les droits de l'utilisateur Bob sur la vue GENRES :

`REVOKE ALL PRIVILEGES ON genres FROM Bob;`

**! Révoque juste les droits qui ont été donnés !**

- Retire le rôle ADMINS à Alice :

`REVOKE Admins FROM Alice;`

# Conclusion

L'administration d'une base de données peut se faire par l'utilisation conjointe de **vues** et de **droits**.

## Exemple

interdire l'accès aux tables mais autoriser l'accès des vues dérivées de ces tables.

- Les vues présentent les données indépendamment des tables.
- Les droits organisent les utilisateurs en fonction de groupes et de rôles.



# TRANSACTIONS, RÉSISTANCE AUX PANNES ET CONCURRENCE D'ACCÈS

Transactions

# Notion de transaction

Une transaction est une suite d'opérations interrogeant la BD, pour laquelle l'ensemble des opérations doit être, soit validé, soit annulé.

Toute transaction est réalisé ou rien ne l'est :

- **Validation** : toute la transaction est prise en compte,
- **Annulation** : la transaction n'a aucun effet.

# Validation d'une transaction

**Explicites : COMMIT;**

**Implicites (Oracle) :**

- Commande de déconnexion en mode interactif tout ordre de mise à jour du schéma (create, drop, alter...)
- Commande « grant »
- Toute mise à jour des données en mode de confirmation automatique (autocommit on)

**Effet** : confirme toutes les mises à jour depuis le début de la transactions (i.e. depuis la dernière confirmation ou annulation)

# Annulation d'une transaction

**Explicites : ROLLBACK**

**Implicites** : déconnexion anormale (autre que « exit »)

**Effet** : annule toutes les mises à jour depuis le début de la transactions (i.e. depuis la dernière confirmation ou annulation)



## TRANSACTIONS, RÉSISTANCE AUX PANNES ET CONCURRENCE D'ACCÈS

Résistance aux pannes

# Résistance aux pannes

**Le SGBD doit permettre de :**

- Minimiser le travail perdu
- Assurer un retour à des données cohérentes

**A quoi sont dues les pannes ?**

- Erreur humaine
- Erreur de programmation
- Défaillance matérielle

# Types de pannes

- **La panne sur une action** : lorsqu'une commande SGBD est mal exécutée
- **La panne de transaction** : erreur de programmation, accès concurrents, deadlock...
- **La panne système** : nécessite le redémarrage du système (erreur logicielle, coupure de courant...)
- **La panne mémoire secondaire** : suite à une défaillance matérielle ou logicielle impliquant de mauvaises écritures

# Reprise sur panne

**Le SGBD doit fournir un protocole aux applications permettant de :**

- Faire une transaction
- Défaire une transaction
- Refaire une transaction

**Trois moyens à conjuguer :**

- La journalisation
- Les sauvegardes
- La réPLICATION

**En cas de panne :**

- On reprend l'état sauvegardé de la base
- On ré-exécute toutes les actions du journal
- La réPLICATION permet de limiter les interruptions de service



## TRANSACTIONS, RÉSISTANCE AUX PANNES ET CONCURRENCE D'ACCÈS

Concurrence d'accès

# Gestion de la concurrence d'accès

- Accès concurrent
  - Il y a un accès concurrent lorsque plusieurs utilisateurs (transactions) accèdent en même temps à la même donnée dans une base de données.
- Gestion des accès concurrents (contrôle de concurrence)
  - S'assurer que l'exécution simultanée des transactions produit le même résultat que leur exécution séquentielle (l'une puis l'autre)

# Accès concurrents

- Problèmes posés par les accès concurrents
  - Perte de mise à jour
  - Lecture impropre
  - Lecture non reproductible
  - Objets fantômes

# Perte de mise à jour

T1 et T2 modifient simultanément A

<b><i>T<sub>1</sub></i></b>	<b><i>T<sub>2</sub></i></b>	<b><i>BD</i></b>
		<i>A</i> = 10
read <i>A</i>		
	read <i>A</i>	
<i>A</i> = <i>A</i> + 10		
write <i>A</i>		<i>A</i> = 20
	<i>A</i> = <i>A</i> + 50	
	write <i>A</i>	<i>A</i> = 60

Les modifications effectuées par T1 sont perdues

# Lecture impropre

<b>T<sub>1</sub></b>	<b>T<sub>2</sub></b>	<b>BD</b>
		$A + B = 200$
		$A = 120$ $B = 80$
read A		
$A = A - 50$		
write A		$A = 70$
	read A	
	read B	
	display A + B (150 est affiché)	
read B		
$B = B + 50$		
write B		$B = 130$

T2 lit une valeur de A non validée, affiche une valeur incohérence

# Lecture impropre (suite)

<b>T<sub>1</sub></b>	<b>T<sub>2</sub></b>	<b>BD</b>
		<i>A</i> = 50
	<i>A</i> = 70	
	write <i>A</i>	<i>A</i> = 70
read <i>A</i> (70 est lu)		
	rollback (La valeur initiale de <i>A</i> est restaurée)	<i>A</i> = 50

T1 lit une valeur de A non confirmée

# Lecture non reproductible

<b><math>T_1</math></b>	<b><math>T_2</math></b>	<b><math>BD</math></b>
		$A = 10$
	read A (10 est lu)	
$A = 20$		
write A		$A = 20$
	read A (20 est lu)	

$T_2$  lit deux valeurs de A différentes

# Objet fantôme

<b><math>T_1</math></b>	<b><math>T_2</math></b>	<b><math>BD</math></b>
		$E = \{1, 2, 3\}$
display card(E) 3 est affiché		
	insert 4 into E	$E = \{1, 2, 3, 4\}$
display card(E) 4 est affiché		

# Contrôle des accès

- Verrouillage :
  - Le verrouillage est la technique la plus classique pour résoudre les problèmes dus à la concurrence:
    - Avant de lire ou écrire une donnée, une transaction peut demander un verrou sur cette donnée pour interdire aux autres transactions d'y accéder.
    - Si ce verrou ne peut être obtenu, parce qu'une autre transaction en possède un, la transaction demandeuse est mise en attente.
  - Afin de limiter les temps d'attente, on peut jouer sur :
    - La granularité du verrouillage : pour restreindre la taille de la donnée verrouillée (n-uplet, une table)
    - Le mode de verrouillage: pour restreindre les opérations interdites sur la donnée verrouillé.

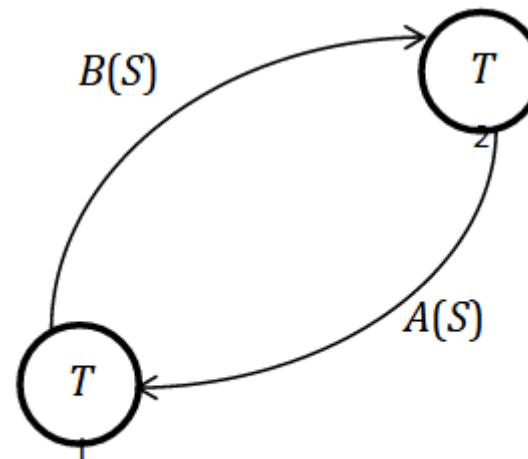
# Exemple verrouillage

T1	T2	Résultat A=10
Read A avec verrou		
A=A+10	Read A avec verrou	
	attente	A=20
Write A Commit;		A=20
	Read A avec verrou	20
	A=A+50	50
	Write A commit	A=70

# Problème de verrouillage : interblocage

- L'impasse générée par deux transactions (ou plus) qui attendent, l'une, que des verrous se libèrent, alors qu'ils sont détenue par l'autre :

$T_1$	$T_2$
lock X A	
	lock X B
lock S B	
attente	lock S A
attente	attente



graphe d'attente

# Résolution de l'interblocage

- Deux approches :
  - Prévention :
    - Toutes les ressources nécessaires à la transaction sont verrouillées au départ
    - Problème : cas des transactions qui ne démarrent jamais !
    - Méthode peu utilisée aujourd'hui
  - Détection :
    - On inspecte à intervalles réguliers le graphe d'attente pour détecter si un interblocage s'est produit. Dans ce cas, on défait l'une des transactions bloquées et on la relance un peu plus tard.
    - On annule une transaction dont le temps d'attente dépasse un certain seuil, et on la relance un peu plus tard.



# PROCÉDURES STOCKÉES ET DÉCLENCHEURS

Procédures stockées

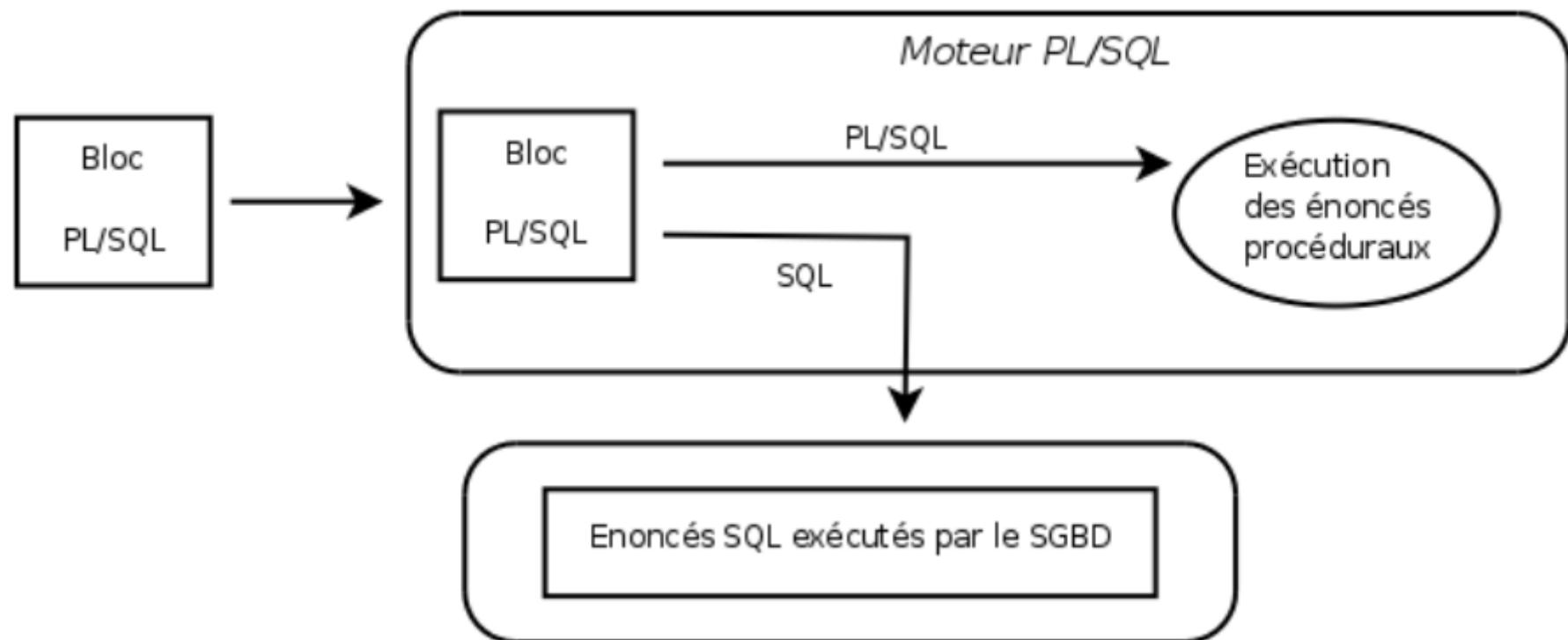
# Limites du SQL

- **Langage non procédural**
- Il n'a pas de :
  - Variables
  - Itérations
  - Branchements conditionnels
- Impossible de lier plusieurs requêtes SQL :  
**regrouper un bloc de commandes et le soumettre au noyau**

# PL/SQL

- Langage fournissant une interface procédurale au SGBD Oracle
- Intégration du langage SQL en lui apportant une dimension procédurale
- Réalisation de traitements algorithmiques (ce que ne permet pas SQL)
- Mise à disposition de la plupart des mécanismes classiques de programmation des langages hôtes tels que C, C++, JAVA ...

# Environnement PL/SQL



# Avantage de PL/SQL

- Structures itératives : WHILE ... LOOP, FOR ... LOOP, LOOP ...
- Structures conditionnelles : IF ... THEN ... ELSE | ELSEIF ... ENDIF, CASE ...
- Déclaration des curseurs et des tableaux
- Déclaration de variables
- Affectation de valeurs aux variables
- Branchements : GOTO, EXIT
- Exceptions : EXCEPTION

# Utilisation de PL/SQL

Le PL/SQL peut être utilisé sous trois formes :

- Un bloc de code, exécuté comme une unique commande SQL, via un interpréteur standard (SQLplus ou iSQL\*Plus)
- Un fichier de commande PL/SQL
- Un programme stocké (procédure, fonction, trigger)

# Langage PL/SQL : Blocs

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

# Anatomie d'un bloc

DECLARE

*Déclarations de constantes et de variables*

BEGIN

*Commandes exécutables*

END;

# Sous-bloc DECLARE (1)

- Une variable, c'est :
  - 30 caractères au plus
  - commence par une lettre
  - peut contenir \_, \$ et #
  - insensible à la casse
  - portée habituelle des langages à blocs
  - doit être déclarée avant utilisation

## Sous-bloc DECLARE (2)

- Déclaration d'un type d'un attribut
  - < variable > table.attribut%type
- Déclaration d'un type n-uplet
  - < variable > table%rowtype
  - < variable > record
- Déclaration d'une date
  - < variable > date
- Déclaration d'un entier
  - < variable > integer
- Déclaration d'une constante
  - < variable > CONSTANT := constante

## Sous-bloc DECLARE (3)

```
employe emp%ROWTYPE;  
nom emp.nom.%TYPE;
```

```
select * INTO employe  
from emp  
where matricule = 900;
```

```
nom := employe.nome;  
employe.dept := 20;
```

...

```
insert into emp values employe;
```

## Sous-bloc DECLARE (4)

Utilisation du type record:

```
TYPE nomRecord IS RECORD (
    champ1 type1,
    champ2 type2
);
```

- Dans tous les cas :
  - Déclarations multiples **interdites**
  - Si une variable porte le même nom qu'une colonne d'une table, c'est la **colonne qui l'emporte**

# Sous-bloc BEGIN ... END;

- Partie BEGIN ... END :
  - Affectation
  - Branchement conditionnel
  - Itération

# Branchement conditionnel

```
IF condition THEN  
ELSE IF condition THEN  
ELSE IF condition THEN  
ELSE instruction  
END IF;  
END IF;  
END IF;
```

# Choix (type simple)

CASE expression

WHEN expr1 THEN instruction1 ;

WHEN expr2 THEN instruction2 ;

...

ELSE instructions ;

END CASE;

# Itérations (1)

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

```
WHILE condition LOOP  
instructions ;  
END LOOP;
```

## Itérations (2)

```
FOR element IN [REVERSE] domaine  
LOOP  
instructions ;  
END LOOP;
```

Domaines :

- intervalle comme 1..100
- éléments d'une table (SELECT)

# Procédures et fonctions

- Offrir aux programmeurs la possibilité de créer des blocs de traitements.
- Introduire quelques bases de la programmation dans les moteurs SQL
- Les procédures et les fonctions sont stockées dans la base de données comme les autres objets (tables, requêtes, ...)

# Création d'une fonction/procédure

```
CREATE FUNCTION gen_cle_client () RETURNS OPAQUE AS
'
DECLARE
nocli integer;
BEGIN
SELECT nocli INTO max(no_client) FROM client;
IF nocli ISNULL THEN
nocli:=0;
END IF;
NEW.no_client:=nocli+1;
RETURN NEW;
END;
'
LANGUAGE 'plpgsql';
```

# CREATION avec paramètres

```
CREATE FUNCTION double (integer)
RETURNS integer
AS
'BEGIN
RETURN 2*$1;
END; '
LANGUAGE 'plpgsql';
• Les paramètres sont utilisés via les macros
$1, $2, ..., $x pour les paramètres 1, 2, ..., xème
```

# Remplacement

- REPLACE permet de changer le code d'une fonction existante
- En général lors de la création on peut aussi mettre la primitive REPLACE.

**CREATE OR REPLACE FUNCTION** double (integer)

RETURNS integer

AS

'BEGIN

RETURN 2\*\$1;

END; '

LANGUAGE 'plpgsql';

# Exemple

```
CREATE OR REPLACE FUNCTION gen_cle_client () RETURNS void AS
'
DECLARE
i RECORD;
total real;
BEGIN
FOR i IN  SELECT nocli, count(qtité*PU)
INTO total
FROM commande GROUP BY nocli
LOOP
IF total > 10000 THEN
INSERT i into table_TB_CLIENT;
END IF;
END LOOP;
END;
'
LANGUAGE 'plpgsql';
```



## PROCÉDURES STOCKÉES ET DÉCLENCHEURS

Déclencheurs

# Qu'est-ce qu'un trigger ?

- Un programme **stocké** dans une BD
  - associé à une **table** donnée
  - associé à un **événement** se produisant sur cette table
- Le trigger est exécuté lorsque l'événement auquel il est attaché se produit sur la table.

# Intérêt

- Maintenance des tables facilitées
- Mise à jour automatique cohérente
- Sécurité renforcée
- Gestion d'un historique
- Gestion événementielle transparente
- Analyse de données pour la décision
- Implémentation des MCT

# Les événements déclenchants

Le programme associé au trigger se déclenche lorsque l'un des événements se produit :

- Insertion dans la table : INSERT
- Mise à jour : UPDATE
- Suppression : DELETE

# Caractéristique

- Un trigger est associé à une seule table
- S'exécute à l'arrivée de l'événement
- Déclenche un bloc PL/SQL (fonction)
- Détruit avec la destruction d'une table
- Peut être désactivé.

# Types de Déclencheurs

- **Triggers table (STATEMENT)**

Sont exécutés une seule fois lorsque des modifications surviennent sur une ou plusieurs lignes de la table.

*Il n'est pas possible d'avoir accès à la valeur ancienne et la valeur nouvelle (OLD et NEW)*

- **Triggers ligne (ROW)**

Sont exécutés séparément pour chaque ligne modifiée dans la table.

*Il est possible d'avoir accès à la valeur ancienne et la valeur nouvelle grâce aux mots clés OLD et NEW*

# SYNTAXE Trigger Table

```
CREATE [OR REPLACE] TRIGGER nom_trigger
{BEFORE | AFTER} événement
ON nom_table
DECLARE
-- Déclarations variables, curseurs, records, ...
BEGIN
-- Traitement
EXCEPTION
-- Gestionnaires d'exceptions
END [nom_Trigger]
```

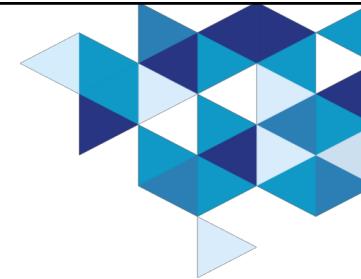
# SYNTAXE Trigger ligne

```
CREATE [OR REPLACE] TRIGGER nom_trigger
{BEFORE | AFTER} événement
ON nom_table
FOR EACH ROW [WHEN condition]
[REFERENCING {[old [AS] nom_old] | New [AS] nom_new}]}
DECLARE
-- Déclarations variables, curseurs, records, ...
BEGIN
-- Traitement
EXCEPTION
-- Gestionnaires d'exceptions
END [nom_Trigger]
```

# Exemple

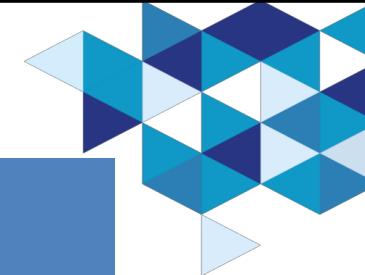
```
CREATE FUNCTION gen_cle_client () RETURNS OPAQUE AS
'DECLARE
nocli integer;
BEGIN
SELECT nocli INTO max(no_client) FROM client;
IF nocli ISNULL THEN
nocli:=0;
END IF;
NEW.no_client:=nocli+1;
RETURN NEW;
END; '
LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER trig_bef_ins_client
BEFORE INSERT
ON client
FOR EACH ROW
EXECUTE PROCEDURE gen_cle_client();
```



## PARTIE IV : BASES DE DONNÉES ET LANGAGE DE PROGRAMMATION

## PLAN



- Introduction
  - Besoins
  - Connectivité BD de Java: JDBC
  - Les drivers JDBC
- API de JDBC : Accès aux données
  - Driver manager
  - Interface de Connexion
  - Invocation de requêtes et de procédures
  - Récupération des résultats
  - Exemples

# Besoins

- Applications interagissant avec des SGBD
  - page web contenant une applet qui accède à des données distantes
  - accès depuis des postes hétérogènes (Windows, UNIX ...) aux BDs via Intranet ou Internet.
- Portabilité => standardisation des accès
  - "*write it once and run it anywhere*"
- Java : Excellent langage pour les applications BD
  - robuste, sûr, facile à comprendre et à utiliser et téléchargeable automatiquement sur le réseau
  - **à condition de** définir un accès unifié aux différents SGBD (et SQL)

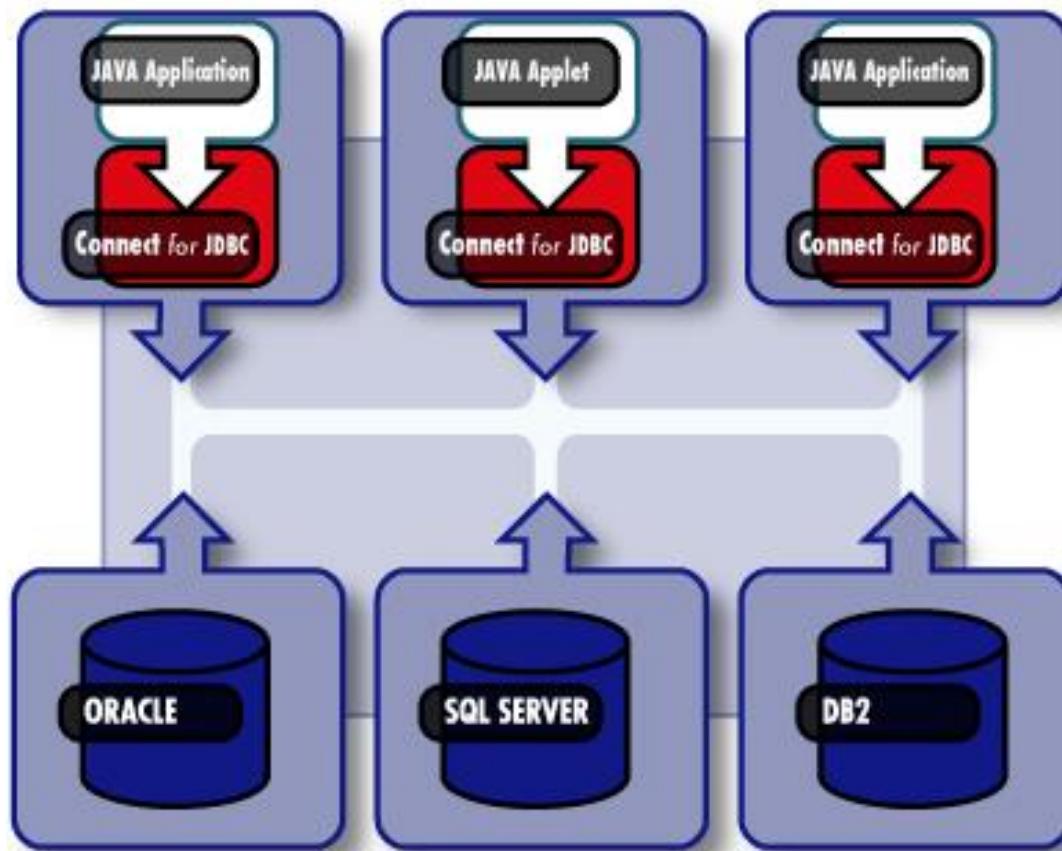
## Besoins (2/2)

- Pourquoi pas ODBC de MicroSoft
  - ODBC (Standard d'accès en langage C) inadapté car :
    - ✓ utilise les pointeurs et le typage faible (`void *`) du langage C
    - ✓ appel C depuis Java => manque en : sécurité, robustesse, portabilité d'applications, etc...
    - ✓ trop complexe à apprendre pour une application simple
      - mélange les aspects simples et avancés
    - ✓ contraire au style de java :
      - utilise des fichiers locaux pour la configuration
    - ✓ Driver utilisant C =>
      - installation obligatoire sur chaque client

# Objectifs de JDBC

- Fournir un accès homogène aux SGBDR
- Abstraction des SGBDR cibles
- Requêtes SQL
- Simple à mettre en oeuvre

# Interface Application/BD



# JDBC : Définitions

- JDBC = *Java DataBase Connectivity*
  - ✓ Ref. JDBC : <https://docs.oracle.com/javase/tutorial/jdbc/>
- Ensemble de classes et de méthodes communes
  - ✓ API ou Interface de Programmation d'Application
- Accès aux BDs relationnelles
  - ✓ Supporte la norme SQL2 Entry Level (noyau minimal) puis SQL3
  - ✓ Basé sur X/Open SQL Call Level Interface (~ODBC de Microsoft)
  - ✓ Passerelle JDBC-ODBC possible
  - ✓ Définit un label "*JDBC Compliant*" de conformité du SGBD
  - ✓ Peut transmettre les requêtes propres à un SGBD (risque de non portabilité)

# JDBC : Fonctionnement

Composé de deux ensembles d'interfaces abstraites :

- l'**API du pilote (*driver*) JDBC**

- ✓ Destinée aux éditeurs de SGBD pour développer le driver
- ✓ Consiste à implanter les éléments définis dans l'interface abstraite

- l'**API JDBC**

- ✓ Destinée aux développeurs de programmes Java utilisant une BD
- ✓ Définit l'accès à une BD, l'envoi de requêtes et la réception des résultats

## Rôle du driver

- Le *driver* permet à JDBC d'interagir avec le SGBDR
- Chaque base de données utilise un pilote (*driver*) qui lui est propre et qui permet de convertir les requêtes dans le langage natif du SGBDR
- Les drivers dépendent du SGBD auquel ils permettent d'accéder
- Tous les SGBDs importants du marché ont un (et même plusieurs) driver, fourni par l'éditeur du SGBD ou par des éditeurs de logiciels indépendants.

## Les drivers JDBC (1/3)

- Il existe des *drivers* pour Oracle, Sybase, Informix, DB2, ...
- 4 types de drivers :
  - 1. *Bridge ODBC* (fourni avec JDBC)
  - 2. *Native-API partly-Java driver*
  - 3. *JDBC-Net all-Java driver*
  - 4. *Native-protocol all-Java driver*
- Les types 1 et 2 nécessitent des architectures 3-tiers pour les applets

# Les drivers JDBC (2/3)

- Les 4 types de drivers :

## 1. Pont JDBC-ODBC plus driver ODBC (ex de JavaSoft)

- le code ODBC doit être chargé sur chaque client

## 2. API native (partiellement en Java)

- convertit les appels JDBC en appels à l'API client propre au SGBD, dont le code doit être chargé sur chaque client (Ex: oracle.oci8)

## 3. JDBC-Net (driver Java pur)

- convertit les appels JDBC en protocole indépendant du SGBD (basé sur un middleware)
- solution adaptée à l'utilisation en Intranet.

## 4. Protocole natif (driver Java pur)

- convertit les appels JDBC en protocole du réseau utilisé directement par le SGBD. Permet un appel direct du client au serveur BD => pratique pour les accès Intranet.
- Comme ce sont des protocoles propriétaires, les éditeurs de SGBD développent cette solution pour la vendre (ex: oracle.thin)

# Les drivers JDBC (3/3)

Application Java

JDBC Driver Manager

JDBC Net

Pont JDBC-ODBC

API native

natif pur Java



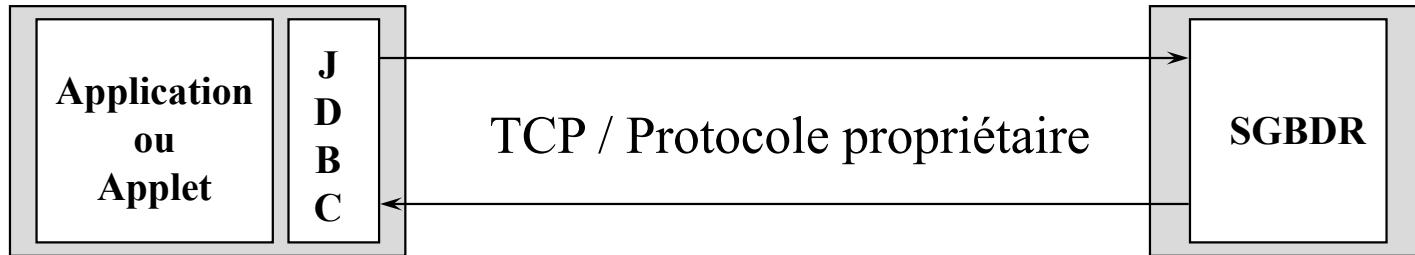
Protocole  
JDBC

Protocole d'accès propriétaire  
au SGBDR Distant

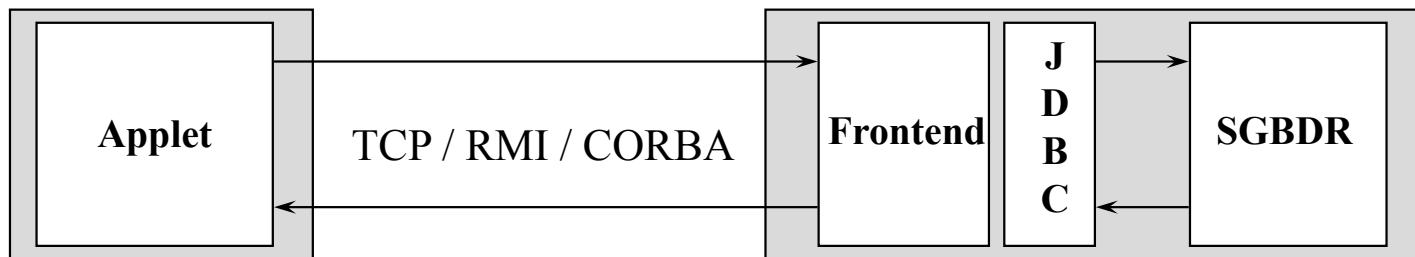
# JDBC et le client/serveur

- **Architecture à 2 niveaux (modèle 2-tiers)**
  - ✓ L'application Java communique directement avec la BD
- **Architecture à 3 niveaux (modèle 3-tiers)**
  - JDBC permet de standardiser l'accès depuis un middleware
    - ✓ Les commandes sont envoyées à un "middle tier" ou middleware qui envoie le SQL au SGBD
    - ✓ Le résultat est retourné par le SGBD au middleware qui les renvoie à l'utilisateur.
- **Le type d'architecture dépend du Driver JDBC**

# Architectures 2-tier et 3-tier



**Architecture 2-tier**



**Architecture 3-tier**

## 2 - L'API JDBC

# Accès aux données

1. Charger le *driver*
2. Connexion à la base de données
3. Création d'un *statement*
4. Exécution de la requête
5. Lecture des résultats

# Définition du driver

- Le DriverManager gère la liste des drivers JDBC chargés dans la machine virtuelle
  - Fait la correspondance entre les URL et les drivers disponibles
  - Si plusieurs, il consulte la liste des drivers configurés
- Chargement du driver de 2 manières :

✓ Soit par la méthode Class.forName (recommandée) :

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");  
//charger le driver  
String url = "jdbc:odbc:fred";
```

```
DriverManager.getConnection(url,"userID","passwd");
```

✓ Soit en le rajoutant à une liste de drivers

```
DriverManager.registerDriver(new  
oracle.jdbc.driver.OracleDriver());  
➤ si plusieurs drivers, il consulte cette liste
```

# L'interface de Connexion

- C'est le canal de communication entre Java et la BD
  - Crée à partir d'un URL et gérée par le DriverManager
  - Des arguments (nom utilisateur, mot de passe) peuvent être fournis
  - Gère les transactions SQL
    - ✓ par défaut en mode automatique (chaque requête est validée)
    - ✓ en mode non automatique, connection.commit() (resp. rollback()) valide (resp. annule la transaction)

# Traitement des exceptions

- Il vaut mieux le faire sérieusement pour remonter les erreurs :

```
catch (SQLException ex) {  
    // Si une exception SQL survient, il affiche les messages d'erreurs du SGBD  
    System.out.println ("\n*** ERREUR SQL ***\n");  
    while (ex != null) {  
        System.out.println ("SQL Etat: " + ex.getSQLState ());  
        System.out.println ("Message: " + ex.getMessage ());  
        System.out.println ("Code de l'erreur: " +  
            ex.getErrorCode ());  
        ex = ex.getNextException ();  
    }  
} // catch SQL
```

# Chargement du driver

- Utiliser la méthode `forName` de la classe `Class` :

- `Class.forName("sun.jdbc.odbc.JdbcOrcdbDriver") ;`
  - `Class.forName("postgres95.pgDriver") ;`
  - `Class.forName("oracle.jdbc.driver.OracleDriver") ;`

# Connexion à la base de données

- Accès via un URL qui spécifie :
  - l'utilisation de JDBC
  - le driver ou le type du SGBDR
  - l'identification de la base
- Exemple :
  - jdbc:odbc:ma\_base
  - jdbc:pg95:mabase?username=toto:password=titi
  - jdbc:oracle:thin:@maMachine:1521:maBase
- Ouverture de la connexion :

```
Connection conn = DriverManager.getConnection(url, user, password);
```

# Création d'un Statement (1/3)

- 3 types de *statements* :
  - **statement** : requêtes simples (**Attention aux injections SQL !!**)
  - **prepared statement** : requêtes précompilées
  - **callable statement** : procédures stockées
- Création d'un *statement* :

```
Statement stmt =  
    conn.createStatement();
```

## Création d'un Statement (2/3)

- L'interface Statement gère les requêtes statiques (simples et sans paramètres)
  - SELECT (résultat dans la classe ResultSet);
    - ✓ lancée par la méthode executeQuery ()
  - INSERT, DELETE, UPDATE (retourne un entier) et requêtes du LDD
    - ✓ Lancée par la méthode executeUpdate ()
- L'interface ResultSet
  - Gère l'accès aux résultats d'un SELECT
  - Curseur sur l'ensemble des tuples résultats parcouru par la méthode next() et lu par get...()
  - Les colonnes sont référencées par leur numéro ou par leur nom

## Création d'un Statement (3/3)

- L'interface `PreparedStatement` gère les requêtes dynamiques (contenant des paramètres)
  - peut être un `SELECT` ou une mise à jour
  - requête compilée qui peut avoir ou non des paramètres
  - méthodes `set()` et `clearParameters()` pour affecter les paramètres
  - lancée par la méthode `execute()`
- L'interface `CallableStatement` gère l'invocation de procédures stockées dans le SGBD
  - méthodes `set()`, `get()`
  - `registerOutParameter()` pour affecter les paramètres et les déclarer
  - lancée par la méthode `execute()`

## Exécution d'une requête (1/2)

- 3 types d'exécutions :
  - executeQuery : pour les requêtes qui retournent un ResultSet
  - executeUpdate : pour les requêtes INSERT, UPDATE, DELETE, CREATE TABLE et DROP TABLE
  - execute : pour quelques cas rares (procédures stockées)

## Exécution d'une requête (2/2)

- Exemple d'exécution de requête :

```
String myQuery = "SELECT prenom, nom, email " +  
    "FROM employe " +  
    "WHERE (nom='Dupont') AND (email IS NOT NULL) " +  
    "ORDER BY nom";
```

```
ResultSet rs = stmt.executeQuery(myQuery);
```

## Lecture des résultats (1/5)

- `executeQuery()` renvoie un `ResultSet`
- Le `ResultSet` se parcourt itérativement ligne par ligne
- Les colonnes sont référencées par leur numéro ou par leur nom
- L'accès aux valeurs des colonnes se fait par les méthodes `getXXX()` où `XXX` représente le type de l'objet
- Pour de très grandes lignes, on peut utiliser des *streams*.

## Lecture des résultats (2/5)

- Avec JDBC 1.0, Impossible de revenir au tuple précédent ou de parcourir l'ensemble dans un ordre aléatoire
- Avec JDBC 2.0, on peut parcourir le ResultSet :
  - d'avant en arrière : `next()` vs. `previous()`
  - en déplacement absolu (aller à la n-ième ligne) :  
`absolute(int row)`, `first()`, `last()`, ...
  - en déplacement relatif (aller à la n-ième ligne à partir de la position courante du curseur, ...) :  
`relative(int row)`, `afterLast()`, `beforeFirst()`, ...

## Lecture des résultats (3/5)

Cas des valeurs nulles :

**Pour repérer les valeurs NULL de la base : utiliser la méthode wasNull () de ResultSet (elle renvoie true si l'on vient de lire un NULL, false sinon).**

**Les méthodes getXXX () de ResultSet convertissent une valeur NULL SQL en une valeur acceptable par le type d'objet demandé :**

- ✓ **les méthodes retournant un objet (getString () , getDate () , . . . ) retournent un "null " Java**
- ✓ **les méthodes numériques (getByte () , getInt () , etc) retournent "0"**
- ✓ **getBoolean () retourne "false"**

# Lecture des résultats (4/5)

Type JDBC	Type Java
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Correspondance des types :

JDBC se base sur les types définis dans SQL2 (ou SQL3 à partir de JDBC 2.0)

*La méthode **ResultSet.getString** convertit tout type en type string. Utile pour l'affichage.*

*La méthode **ResultSet.getObject** convertit tout type en type corr. Java.*

*La méthode **PreparedStatement.setObject** accepte tout type Java permet d'avoir des applications génériques où les paramètres en entrées sont donnés au moment de l'exécution.*

## Lecture des résultats (5/5)

```
java.sql.Statement stmt = conn.createStatement();  
  
ResultSet rs = stmt.executeQuery("SELECT a,b,c FROM Table1");  
  
while (rs.next())  
{  
    // impression des valeurs pour la ligne actuelle  
    int i = rs.getInt("a");  
    String s = rs.getString("b");  
    byte b[] = rs.getBytes("c");  
    System.out.println("ROW = "+i+" "+s+" "+b[0]);  
}
```

# Exemple 1

```
public class TestJDBC {  
  
    public static void main(String[] args) throws Exception {  
        Class.forName("postgres95.pgDriver");  
        Connection conn =  
        DriverManager.getConnection("jdbc:pg95:mabase", "dupond", "");  
        Statement stmt = conn.createStatement();  
        ResultSet rs = stmt.executeQuery("SELECT * from employe");  
        while (rs.next()) {  
            String nom = rs.getString("nom");  
            String prenom = rs.getString("prenom");  
            String email = rs.getString("email");  
        }  
    }  
}
```

## Exemple 2

- Requête paramétrée de mise à jour dans la relation AMI

```
// admettons qu'une connexion "conn" soit déjà établie
Java.sql.PreparedStatement requete =
conn.prepareStatement("UPDATE ami SET remarque = ?
WHERE age = ?") ;

// On fixe le 1e paramètre
requete.setString (1, "dizaine...") ;
for (ans = 10; ans <=120; ans += 10)
{
    requete.setInt (2, ans) ;
    // le 2e paramètre varie selon les données
    int nbTuple = requete.executeUpdate () ;
    // nb de tuples affectés
}
```

## Exemple 3 (1/2)

```
/*
 * Cet exemple montre comment obtenir la liste des usagers
 * disposant d'un compte dans la base. (de la doc. de Oracle)
 */

import java.sql.*;
class Employee
{
    public static void main (String args [])
        throws SQLException
    {
        // Chargement du driver JDBC-Oracle
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

        // Connexion à la base de données <host>:<port>:<sid>.
        Connection conn = DriverManager.getConnection (
            "jdbc:oracle:thin:@titan:1521:a99", "momo", "toto");
    }
}
```

## Exemple 3 (2/2)

```
// Créer un descripteur de requêtes
Statement stmt = conn.createStatement ();

// Sélectionner les noms de la table
ResultSet rset = stmt.executeQuery (
    "select username from dba_users");

// Itérer afin d'obtenir la liste de tous les noms !
while (rset.next ())
    System.out.println (rset.getString(1));

// Fermeture de la connexion à la base
rset.close ();
stmt.close ();
conn.close ();

}
```

# Accès à une BD Oracle via JDBC

- Chargement du Driver :

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- Connexion à la base :

```
Connection conn = DriverManager.getConnection  
("jdbc:oracle:thin:@machine:port:base", "Log", "Psw");
```

- Création de requête :

```
Statement stmt = conn.createStatement();
```

- Exécution de requêtes SELECT :

```
ResultSet rset = stmt.executeQuery("SELECT Num, Nom FROM  
EMP");
```

- Exécution des requêtes de mise à jour et Create/Drop :

```
stmt.executeUpdate("INSERT INTO emp VALUES (val1, val2,  
...);") ;
```

```
stmt.executeUpdate("UPDATE emp SET num = 20 WHERE num=10");
```

# Accès aux métadonnées

Informations décrivant les données à 2 niveaux :

– Informations sur la connexion :

conn.getMetaData -> de la classe DatabaseMetaData

permet des retours tels que : getURL de la BD, etc.

– Informations sur les données d'un "resultSet" :

rs.getMetaData -> resultSetMetaData

La méthode getMetaData() permet d'obtenir les métadonnées d'un ResultSet

**Elle renvoie des ResultSetMetaData (nombre de colonnes, leurs appellations, leurs types, etc)**

On peut connaître :

- ✓ Le nombre de colonne : getColumnCount ()
- ✓ Le nom d'une colonne : getColumnLabel (int col)
- ✓ Le type d'une colonne : getColumnType (int col)
- ✓ ...

# Exemple de méta-données

- Récupération de la description d'une table  
(donnée par la chaîne nom\_table) :

```
rset := stmt.executeQuery("Select * from "+ nom_table);
rsetMeta = stmt.getMetaData();
int nbCol = rsetMeta.getColumnCount(); //Lit le nb de colonnes

//Affiche entêtes et types de colonnes
for (int i=1; i<=nbCol; i++) {
    System.out.print (rsetMeta.getColumnName(i) + "    ");
    System.out.println;
    System.out.print (rsetMeta.getColumnType(i) + "    ");
}
```

# Exemple Récapitulatif (1/2)

```
// Les 7 étapes illustrant l'utilisation de JDBC
// pour accéder à une BD Oracle

// 1 – Importer le package java.sql
import java.sql.*;

// 2 – Charger et enregistrer le driver
Class.forName("oracle.jdbc.driver.OracleDriver");

// 3 – Etablir la connexion au serveur de BD
// Connexion à une BD locale sur le serveur mars, sur le port 1521 et
// dont le SID (System Identifier) de la BD est "orcl"
// avec comme utilisateur "scott" ayant le mot de passe "tiger"

Connection conn = DriverManager.getConnection(
    "jdbc:oracle:thin:@mars:1521:orcl", "scott", "tiger");
```

## Exemple Récapitulatif (2/2)

// 4 – Créer un ordre SQL

```
Statement stmt = conn.createStatement();
```

// 5 – Exécuter l'ordre SQL

```
ResultSet rset = stmt.executeQuery("select ename  
from emp");
```

// 6 – Rapporter les résultats

```
while (rset.next())  
    System.out.println (rset.getString (1));  
}
```

// 7 – Fermer l'ordre et la connexion

```
rset.close();  
stmt.close();  
conn.close();
```

# Types de programmes Java

- Différentes méthodes :
  - **Application, STAND ALONE** (avec un main) :  
    --> javac NomFichier.java  
    --> java NomFichier
  - **Applet** :  
    --> javac NomFichier.java  
    --> NomFichier.html comprenant :
    - 1 - Si les classes de l'applet et du driver sont dans le même répertoire que la page HTML

```
<applet code="JdbcApplet" archive="JdbcApplet.zip" width=500 height=200>
</applet>
```
    - 2 - Autrement, les paramètres (tags) CODEBASE et ARCHIVE précisent leur emplacement sur le serveur ou le chemin complet du fichier zip

```
<applet codebase="." archive="classes111.zip"
code="JdbcApplet" width=500 height=200>
</applet>
```
- **Remarque** : Cela est possible si le paramètre du browser est :  
`netscape.security.PrivilegeManager.enablePrivilege ("UniversalConnect");`

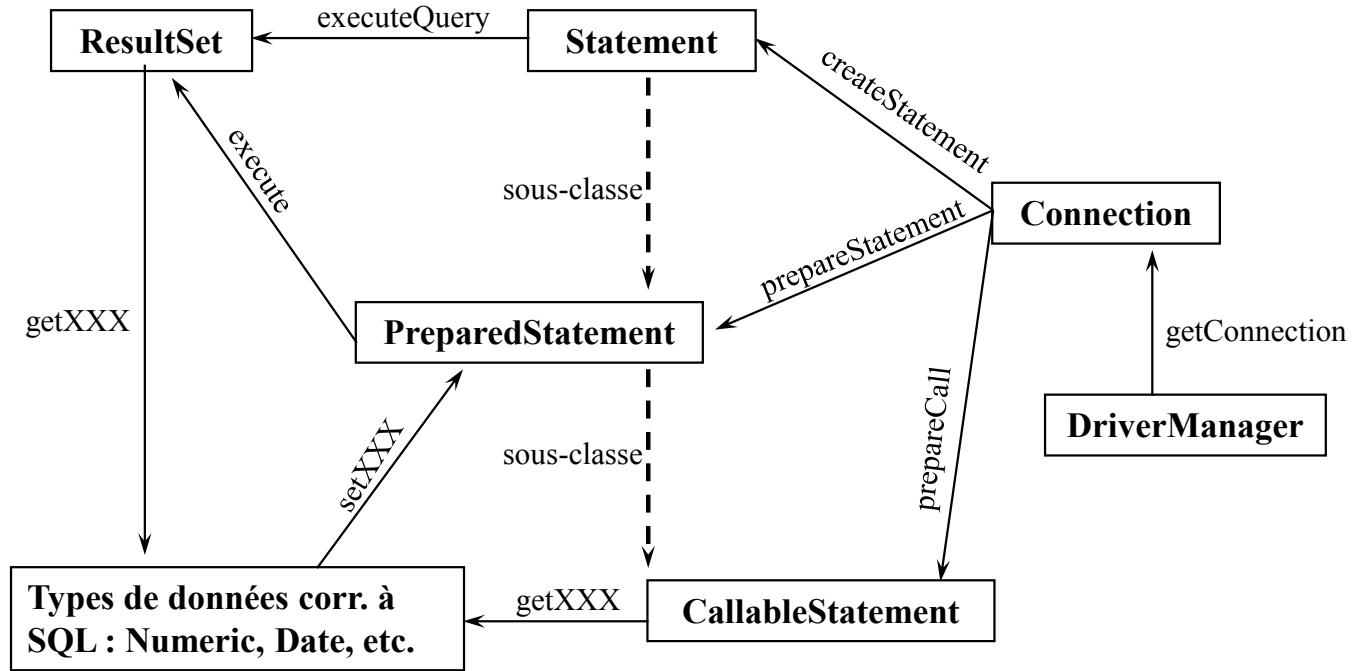
# Exemple de code d'une applet

```
import java.sql.*;  
public class JdbcApplet extends java.applet.Applet  
{  
    Connection conn;  
    public void init()  
    {  
        // Chargement du driver Oracle  
        Class.forName ("oracle.jdbc.driver.OracleDriver");  
  
        // Connexion à la BD distante rdbms3  
        conn = DriverManager.getConnection  
            ("jdbc:oracle:thin:scott/tiger@www-  
             aurora.us.oracle.com:1521:rdbms3");  
        ...  
    }  
}
```

# Principales interfaces de l'API JDBC

Interface	Description
java.sql.Driver	Gère les accès à un SGBD à travers un protocole
java.sql.DriverManager	Gère le chargement des drivers et la création des connexions TCP
java.sql.Connection	Gère les connexions existantes, crée les requêtes, gère les transactions
java.sql.Statement	Gère les requêtes à exécuter, reçoit les résultats
java.sql.ResultSet	Gère l'accès aux tuples d'un résultat
java.sql.PreparedStatement	Représente une requête paramétrée
java.sql.CallableStatement	Gère l'invocation de procédures stockées

# Schéma des liens entre interfaces



# Exemple d'insertion dans une table Clients

Clients : Table					
	Référence Clients	Nom	Prénom	Age	Adresse
	12	Bozo	Le clown	105	Cirque Pinder
	17	Casimir	?	26	Ile aux enfants
▶	(Numéro Auto)			0	

```
public void insert( Personne personne ) {
    try {
        String req =
        "INSERT INTO Clients (Nom,Prénom,Age,Adresse) VALUES ( ?, ?, ?, ? )";
        PreparedStatement prepStmt = connexionBASE.prepareStatement( req );
        prepStmt.setString( 1, personne.getNom() );
        prepStmt.setString( 2, personne.getPrenom() );
        prepStmt.setInt( 3, personne.getAge() );
        prepStmt.setString( 4, personne.getAdresse() );
        prepStmt.executeUpdate();
        prepStmt.close();
    } catch (SQLException pbSQL) {
        pbSQL.printStackTrace();
    }
}
```

# Exemple d'Analyse d'une table Clients

ResultSetMetaData
int getColumnCount() String getColumnName()

```
String req = "SELECT * FROM Clients" ;  
  
ResultSet resultat = requete.executeQuery( req );  
ResultSetMetaData metaData = resultat.getMetaData() ;  
int nbColonnes = metaData.getColumnCount() ;  
  
for( int i=1; i<=nbColonnes; i++ ){  
    String nomColonne = metaData.getColumnName( i ) ;  
    // ...  
}  
  
while( resultat.next() ){  
    for( int i=1; i<=nbColonnes; i++ ){  
        Object objet = resultat.getObject( i ) ;  
        if( objet != null ){  
            String valeur = objet.toString() + " " ) ;  
        }  
    }  
}
```

# JDBC et les transactions

- Par défaut, la connexion est en mode "*auto-commit*" : un commit est effectué automatiquement après chaque ordre SQL.
- On peut :
  - Enlever l'auto-commit par :  
`conn.setAutoCommit(false)`
  - Valider la transaction par : `conn.commit()` .
  - Annuler la transaction par : `conn.rollback()` .

# Exemple de gestion de transaction

```
public void transferer( Client client1, Client client2 ){
    Connection connexion = null ;
    try{
        Connection connexion = DriverManager.getConnection( baseODBC, "", "" );
connexion.setAutoCommit( false ) ;
        Float mont = new Float( montant ) ;
        String requete = "UPDATE Banque SET Solde=(Solde-1000) WHERE Détenteur=?";
        PreparedStatement statement = connexion.prepareStatement( requete ) ;
        statement.setString( 1, client1.getNom() ) ;
        statement.executeUpdate() ;
client2.verifier(); // lance une exception
        requete = "UPDATE Banque SET Solde=(Solde+1000) WHERE Détenteur=?";
        statement = connexion.prepareStatement( requete ) ;
        statement.setString( 1, client2.getNom() ) ;
        statement.executeUpdate() ;
connexion.commit();
    } catch( Exception e ){
        try{
            connexion.rollback();
        } catch( SQLException sql ){}
    } ...
}
```

Banque : Table			
	Référence Compte	Détenteur	Solde
	1	Marcel	10,00 F
	2	Picsou	10 000 000,00 F
▶	(NuméroAuto)		0,00 F

# Gestion de pool de connexions par synchronisation

```
public class Banque{  
    Connection connexion ;  
  
    public Banque() throws ClassNotFoundException {  
        // ...  
        connexion = DriverManager.getConnection( "jdbc:odbc:banque", "", "" );  
    }  
  
    synchronized public boolean transferer( Client client1, Client client2 ){  
        // ...  
        connexion.setAutoCommit( false ) ;  
        // ...  
    }  
}
```

- **Inconvénient** : un objet Banque est verrouillé à chaque appel de la méthode transferer => mauvaises performances !

# Solution plus performante : utiliser un pool de connexions

```
public class Banque{  
  
    PoolDeConnexions pool ;  
  
    public Banque() throws Exception{  
        pool = new PoolDeConnexions() ;  
    }  
    public boolean transferer( Client client1, Client client2, float montant ){  
        Connection connexion = null ;  
        try{  
            connexion = pool.getConnection() ;  
            // ...  
        } finally {  
            if( connexion != null ) pool.libereConnexion( connexion ) ;  
        }  
    }  
}
```

# JDBC et la sécurité

- Traitée à 3 niveaux :
  - **Sécurité dans l'applet**
    - ✓ Stockage local interdit
    - ✓ Applet, serveur web, driver et BD proviennent de la même machine
  - **Sécurité dans l'application**
    - ✓ Accès aux ressources locales et au réseau possible
    - ✓ Accès simultané et sûr à plusieurs SGBD
  - **Sécurité des drivers**
    - ✓ Si connexion TCP/IP non partagée, le *SecurityManager* normal suffit
    - ✓ Si connexion partagée entre plusieurs *connexions JDBC* (accès simultané par plusieurs utilisateurs), à charge du driver de contrôler les autorisations d'accès.

# Evolution

- Version JDBC 1.0
  - Vise une API de bas niveau
    - ✓ minimale, simple, ouverte
    - ✓ base pour développer des interfaces de plus haut niveau
- Version JDBC 2.0
  - Extension de java.sql :
    - ✓ nouvelles méthodes dans les classes initiales
    - ✓ types SQL3 : BLOB, CLOB, type utilisateur et REF
    - ✓ mise à jour en batch, ...
  - + Nouvelle API javax.sql :
    - ✓ Rowset : pour travailler en mode déconnecté
    - ✓ JNDS: gère l'Id. de la connexion et du driver (code indép. de l'URL et du driver)
    - ✓ Transactions distribuées (*standard 2PC de Java TransO service*)
- ORM (Object/Relational Mapping) Hibernate, JPA, ...

# PHP et les Bases de données Oracle

- Les fonctions permettant la connexion à une base Oracle depuis PHP ont été écrites avec la librairie OCI8 (Oracle8 Call-Interface) d'Oracle.
- Connexion au serveur  
La connexion à un serveur Oracle se fait avec la fonction *OCILogOn* :  
`$connexion = OCILogOn("utilisateur", "mot_de_passe", $bdd);`
- Exécuter une requête PL/SQL
  - **préparation de la requête** : La préparation s'effectue avec la fonction *OCIParse* :  
`$stmt = OCIParse($connexion, "SELECT * FROM table");` *OCIParse* utilise deux arguments : un identifiant de connexion (résultat de la fonction *OCILogOn*) et la requête.  
On peut combiner deux *OCIParse* sur le même identifiant de connexion (*\$connexion*) et récupérer les 2 résultats dans 2 variables *\$stmt1* et *\$stmt2* pour exécuter 2 requêtes en même temps sur la même base.
  - **Exécution de la requête proprement dite** : L'exécution s'effectue avec la fonction *OCIExecute* :  
`OCIExecute($stmt);`
- Lire le résultat d'une requête  
Il faut utiliser conjointement 2 fonctions : *OCIFetch* et *OCIResult*. *OCIFetch* permet de récupérer une ligne du résultat, alors que *OCIResult* permet de lire une valeur d'une ligne récupérée par *OCIFetch*.
- Terminer une requête  
A la fin d'utilisation d'un résultat de requête, il est préférable de libérer les ressources allouées avec la fonction *OCIFreeStatement* (*\$stmt*)
- Pour plus de détails : <http://www.php.net> ou <http://www.themanualpage.org/php.html>