

# Bases de Données Relationnelles

## L3 Informatique

Mohamed-Lamine Messai

mohamed-lamine.messai@univ-lyon2.fr



**INSTITUT  
DE LA  
COMMUNICATION**



Université de Lyon, Lyon 2, ERIC UR 3083, Lyon, France

2022-2023

# Introduction

# Où utilise-t-on des bases de données ?

## Un besoin du monde réel

Une entreprise privée de transports scolaires dans la ville de Lyon, désire automatiser la gestion de son réseau. Celui-ci comprend des bus scolaires de différents types ainsi que des chauffeurs.

**Modéliser un besoin du monde réel => automatiser des tâches => proposer un *Système d'Information (SI)* autour d'une base de données**

# Système d'information

## Une définition ?

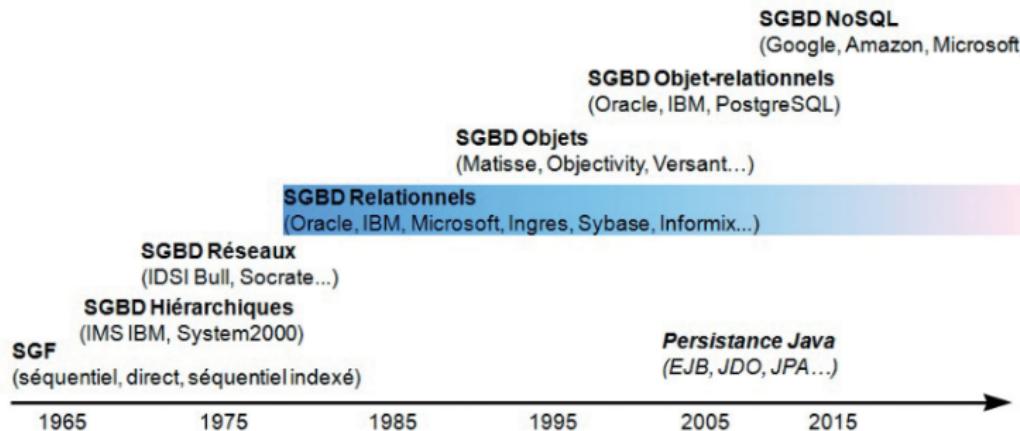
Un **système d'information** est l'ensemble des ressources (matériels, logiciels, données, procédures, ...) structurés pour acquérir, traiter, mémoriser et rendre disponible l'information sous de multiples formes (textes, sons, images, ...) dans et entre plusieurs organisations.

Aujourd'hui, le système d'information permet d'automatiser et de dématérialiser quasiment toutes les opérations incluses dans les activités ou procédures de notre vie quotidienne : personnelle ou professionnelle.

Si l'on doit résumer en quatre étapes, son rôle est de :

**COLLECTER → STOCKER → TRAITER → DIFFUSER**

# Un peu d'historique



Crédit: Livre : Modélisation des bases de données.

# Bases de données : Besoins

## Des besoins dans le domaine de la gestion de l'information

- Décrire l'information
- Manipuler l'information
- Interroger la collection d'informations
- Exactitude et Cohérence
- Garanties tant en terme de fiabilité que de contrôle
- Confidentialité
- Efficacité

# Besoin de pouvoir décrire

Décrire les données de l'application. Par exemple, pour la SNCF cela correspond aux *trains*, *trajets* ou encore *réservations*, *conducteurs*. Cette description doit être effectuée sans faire référence à une solution informatique particulière.

—> **Modélisation conceptuelle**

Elaborer une description équivalente pour le stockage des données dans la SGBD choisie

—> **Modélisation logique**

On utilisera ensuite un langage de description des données

# Besoin de pouvoir manipuler

Créer la base de données initiales avec les données représentant le réseau SNCF

→ **Langage permettant l'insertion de données**

Créer au fur et à mesure les données sur les réservations. Modifier si besoin et éventuellement supprimer toute donnée déjà rentrée

→ **Langage de Manipulation de Données (insertion, modification, suppression) :SQL**

# Besoin de pouvoir interroger

Répondre à toute demande d'information portant sur les données contenues dans la base.

Par exemple :

- Jean Lestrade a-t-il une réservation pour aujourd'hui ?  
Si oui, donner les informations connues sur cette réservation.
- Quels sont les horaires des trains de Lyon à Strasbourg entre 9h et 10h le dimanche ?
- Donner les destinations au départ de Lyon sans arrêt intermédiaire.

→ Langage de requête ou d'interrogation : SQL

## Besoin d'exactitude et de cohérence

Il faut pouvoir exprimer toutes les règles qui contraignent les valeurs pouvant être enregistrées de façon à éviter toute erreur qui peut être détectée.

Par exemple :

- Il ne faut jamais donner la même place dans le même train à deux clients
- Les arrêts d'un train sont numérotés de façon continue (il ne peut y avoir pour un train donné un arrêt n3 s'il n'y a pas un arrêt n2 et un arrêt n1)
- La date de réservation pour un train doit correspondre à un jour de circulation de ce train
- L'heure de départ d'une gare doit être postérieure à l'heure d'arrivée dans cette gare
- L'heure d'arrivée à un arrêt doit être postérieure à l'heure de départ de l'arrêt précédent

→ Langage d'expression de contraintes d'intégrité

# Besoin de garanties

Il ne faut pas que les informations (par exemple, les réservations) soient perdues à cause d'un dysfonctionnement quelconque : erreur de programmation, panne système, panne d'ordinateur, coupure de courant, plantage des serveurs, ...

→ **Garantie de confidentialité**

Il ne faut pas qu'une action faite pour un utilisateur (par exemple l'enregistrement d'une réservation) soit perdue du fait d'une autre faite simultanément pour un autre utilisateur (réservation de la même place).

→ **Garantie de contrôle de concurrence**

# Besoin de confidentialité

Toute l'information doit pouvoir être protégée contre l'accès par des utilisateurs non autorisés que ce soit

- en lecture
- en écriture

Interdire par exemple aux clients de modifier les numéros de trains ou les horaires ou leur réservation.

→ **Garantie de confidentialité**

## Besoin d'efficacité

Le temps de réponse du système doit être conforme aux besoins :

- en interactif : pas plus de deux secondes
- en programmation : assez rapide pour assumer la charge de travail attendue (nombre de transactions par jour).

Usage de mécanismes d'optimisations et, éventuellement, répartition/duplication des données sur plusieurs sites

# Base de Données

Les **Bases de Données** sont des collections de données qui sont structurées et cohérentes.

Les informations sont organisées de manières à être facilement triées, classées et modifiées par le biais d'un logiciel appelé **Système de Gestion de Base de Données** (SGBD).

Le travail effectuer, de conception et de structure de l'information, doit répondre de façon pérenne à un problème, la structure se doit être évolutive et **ne pas répondre uniquement à un problème ponctuel**.

# Plus de précision

Le **Système de Gestion de Bases de Données** est un logiciel permettant de

- **définir une représentation** des informations
- **stocker, interroger et manipuler** de grandes quantités de données (plus que la mémoire vive)
- garantir la **longévité** des données
- garantir l'**accessibilité** de manière **concurrente** (plusieurs utilisateurs simultanés)
- assurer une très grande **fiabilité**

# Composants logiciels d'une base de données

## SGBD

- gère le niveau logique et physique de la base selon l'architecture ANSI-SPARC

## Les outils frontaux L4G

- générateurs : de formes, de rapports, des applications
- intégrés au SGBD ou externes  
Powerbuilder, Borland ...
- Interfaces WEB : HTML, XML, ...
- Interface OLAP & Data Mining  
Intellinet Data Miner (IBM)

Utilitaires : chargement, statistiques, aide à la conception, ...

# Qui utilise des Bases de Données

## Les utilisateurs interactifs

- ils cherchent des informations sans connaître la Base de Données
- utilisent des interfaces (formulaires, web, ...)
- peuvent à la rigueur utiliser des langages tels que QBE

## Les programmeurs d'applications

- construisent les interfaces pour les utilisateurs interactifs
- spécialistes de SQL

## Les DBA ( DataBase Administrators )

- modélisent les bases de données
- créés et maintiennent les bases de données
- ont la priorité sur tous les autres usagers
- peuvent être très bien payé ... car ils sont à la base du BI !

# Conception : Formalisme UML

# Formalismes de modélisation

Différents formalismes (notations graphiques) existent :

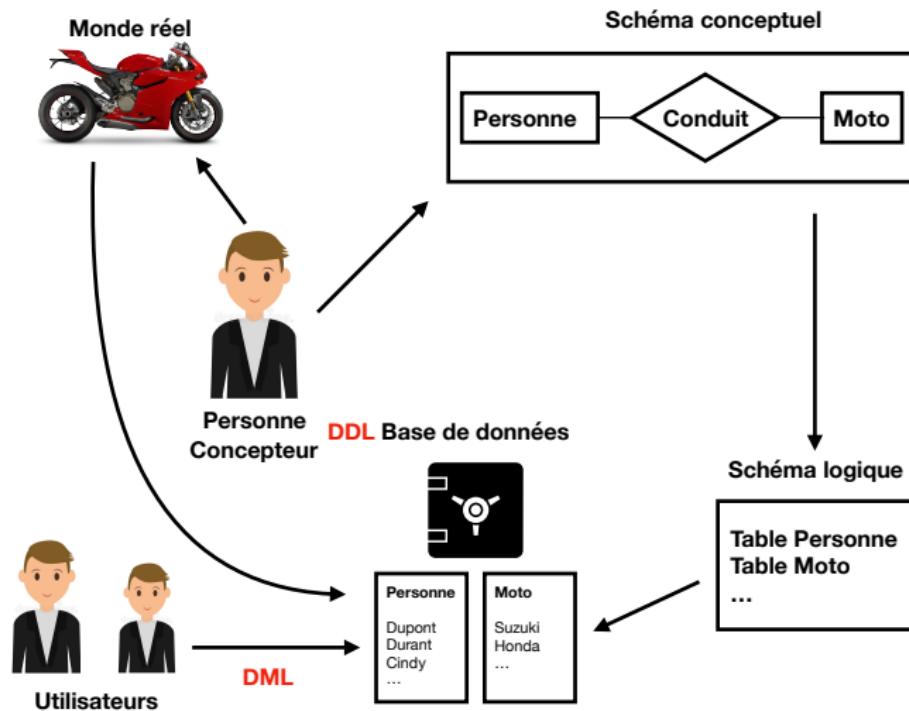
- La méthode Merise (MCD), 1970
- Le modèle Entité-Association, 1976
- Formalisme UML, 1997 : diagramme de classes pour modéliser une base de données

Pourquoi le formalisme UML ?



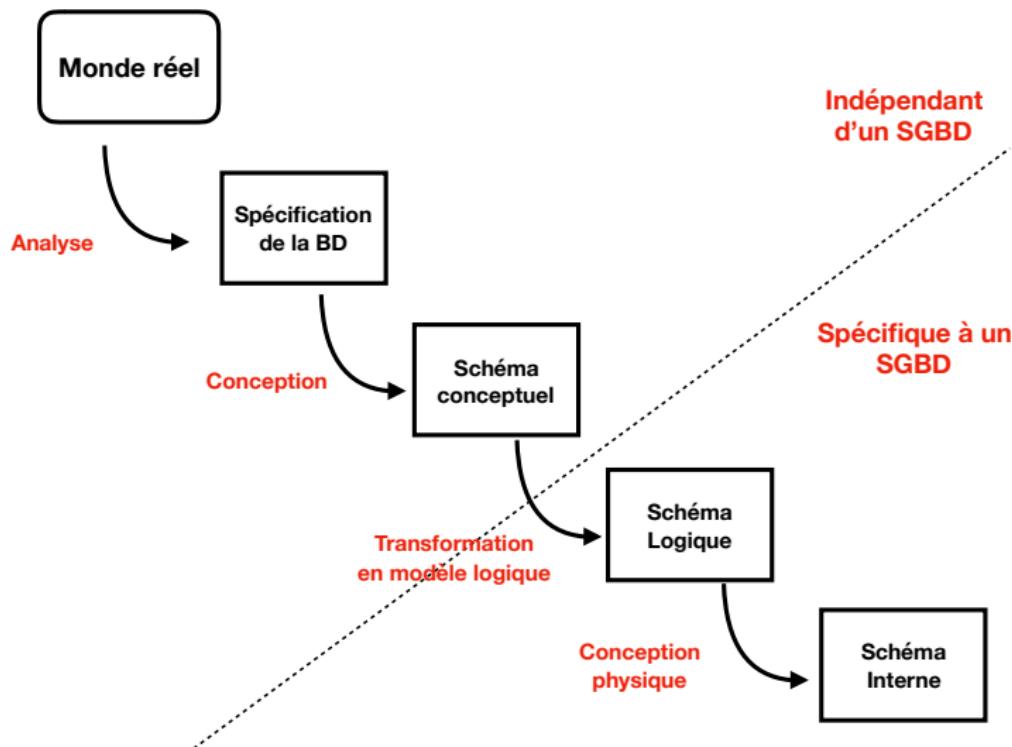
# Niveaux de conception

## De multiples interactions



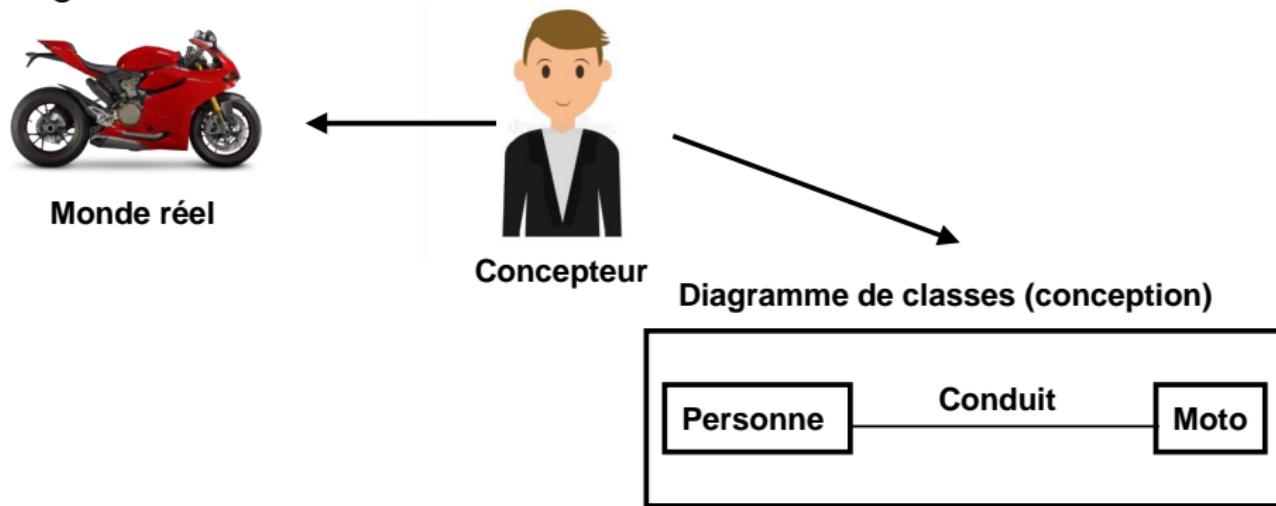
# Étapes de conception d'une BD

Une représentation plus schématique et détaillée



# Focus

Ce qui va nous intéresser en premier lieu c'est comment passer d'une *problématique réelle* au *schéma conceptuel* et comment établir un diagramme de classes.



## Rôle du concepteur

Le rôle du concepteur est fondamental ! C'est à lui que revient la charge de transcrire les observations du monde réel de façon abstraite en prenant en compte tous les liens et leurs caractéristiques. Il doit donc

- réfléchir à la structure de base (tables, attributs, relations) de façon à prendre en compte tous les aspects du problème.
- avoir une représentation cohérente et qui corresponde à la réalité telle qu'elle est perçue par les utilisateurs.

C'est une étape d'autant plus importante qu'elle servira de fondations à ce qui suivra : cohérence - facilité des manipulations - logique.

C'est également une étape qui se veut indépendante de la technologie utilisée (on ne fera pas mention du langage SQL pour le moment !)

# Rôle du concepteur

Pour faire simple, le concepteur doit :

- se mettre à la place de l'utilisateur.
- rester fidèle à la réalité observée.
- développer une représentation simple et compréhensible relative à l'application.

Cela permet aussi de définir *un bon schéma conceptuel* : simplicité - fidélité - longévité - portabilité - compréhensibilité - ...

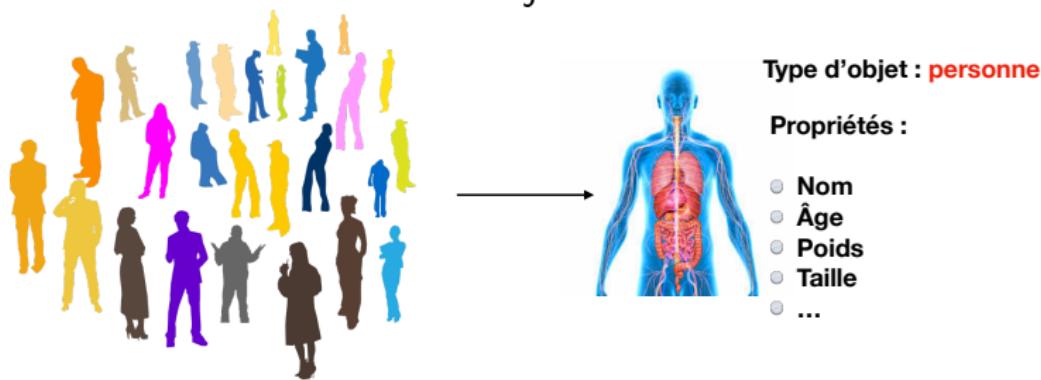
# Élaborer un schéma conceptuel

**Il faut tout d'abord analyser le monde réel :** identifier les différents phénomènes à représenter dans la base de données.

**Les caractériser :** définir leurs caractéristiques : contenu - structure - règles

Réalité perçue → Représentation

Faire abstraction des particularités : passer des *objets* aux *types ou classes d'objets*



# A propos du *langage* UML

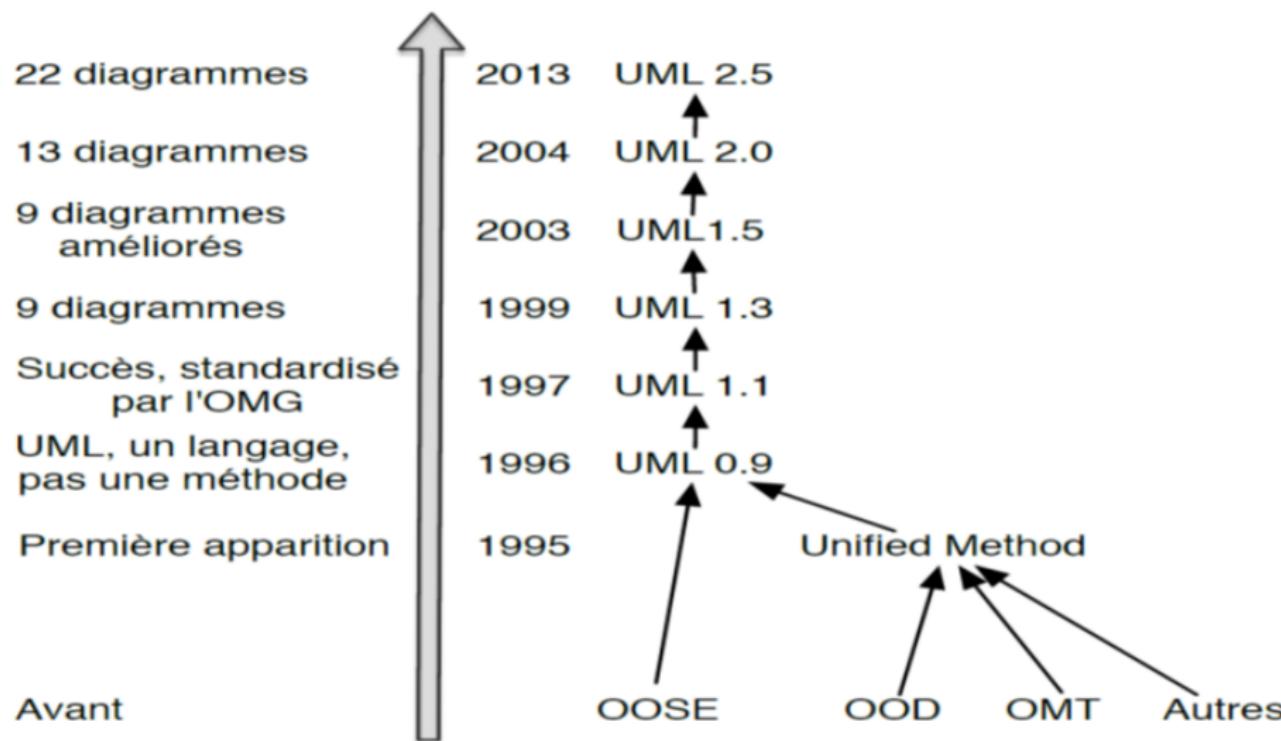
UML est un langage (pas une méthode) de modélisation graphique et textuel qui permet de représenter et de communiquer les divers aspects d'un système.

Langage vs méthode

Langage : notations, grammaire, sémantique.

Méthode : comment utiliser un langage, ensemble d'étapes.

# Apparition et évolution



# Les diagrammes UML

## Diagrammes structurels

- Diagramme de classes
- Diagramme d'objets
- Diagramme de composants
- Diagramme de déploiement

## Diagrammes de comportement

- Diagramme de cas d'utilisation
- Diagrammes d'interaction
  - Diagramme de séquence
  - Diagramme de communication (de collaboration)
- Diagramme d'états-transitions
- Diagramme d'activités

**Pour ce cours : concevoir et transformer un diagramme de classes UML en schéma logique, puis physique.**

# Diagramme de classes

Les classes : Regroupement d'objets de même nature.

Classe
attribut1
attribut2
opération1()
opération2()

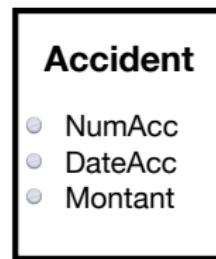
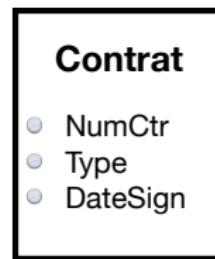
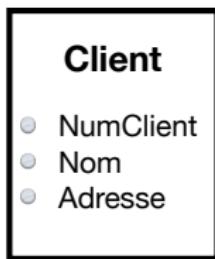
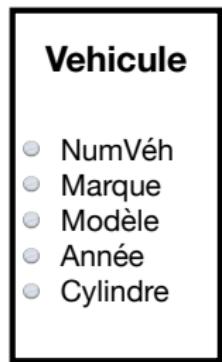
Attribut : Propriété de la classe

- classes ↔ objets
- associations ↔ liens
- attributs ↔ caractéristiques

La base de données contiendra les valeurs des attributs (ou propriétés) des différents objets (instanciation de classes).

# Exemple de classes

Si on s'intéresse aux contrats assurances automobiles :



→ on a 4 entités décrites par respectivement 5, 3, 3 et 3 attributs. **Un attribut a un type : Entier, Réel, Chaîne de caractères, Date**

# Nature des attributs

Un attribut peut être par nature **simple** ou **complexe**

- **Simple** dans le sens où l'attribut est non décomposable : jour de l'année, prénom, année de naissance, lieu, numéro de immatriculation d'un véhicule, numéro de contrat.  
Les valeurs prises sont *simples* et sont aussi bien des nombres que des chaînes de caractères
- **Complex**e dans le sens où il est décomposable : une date qui se décompose en jour - mois - année ou adresse postale complète.  
Il se caractérise donc comme une composition d'attributs simples voire même complexe.

# Nature des attributs

Un attribut peut également être **obligatoire** ou **facultatif**

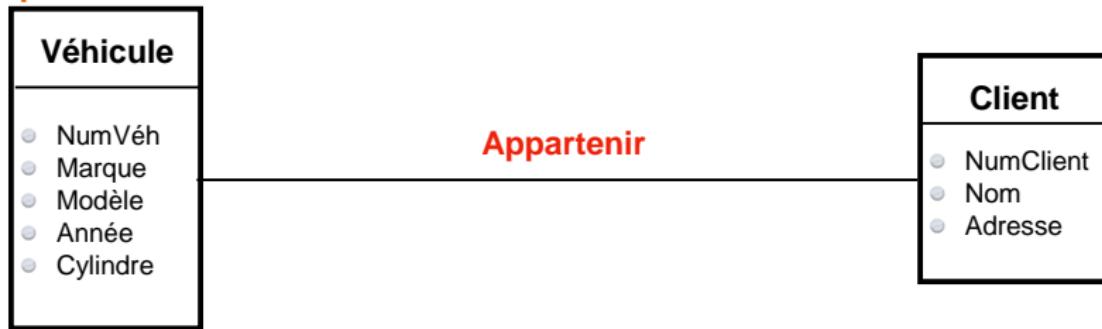
- **Obligatoire** : au moins une valeur est attendue pour cet attribut, *i.e.* le cardinal de l'ensemble des valeurs possibles est au moins égal à 1.  
Ex : Nom, Prénom
- **Facultatif** : il n'est pas nécessaire d'affecter une valeur, le cardinal de l'ensemble des valeurs possibles peut être nul.  
Ex : montant de l'accident, salaire, numéro de téléphone, ...

# Association

## Associations

Relation entre classes. Les associations peuvent être nommées.

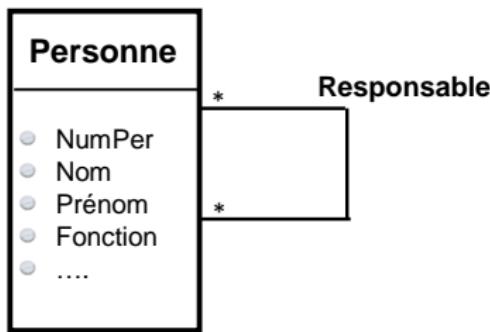
### Exemple



# Association réflexive

## Association d'une classe vers elle-même.

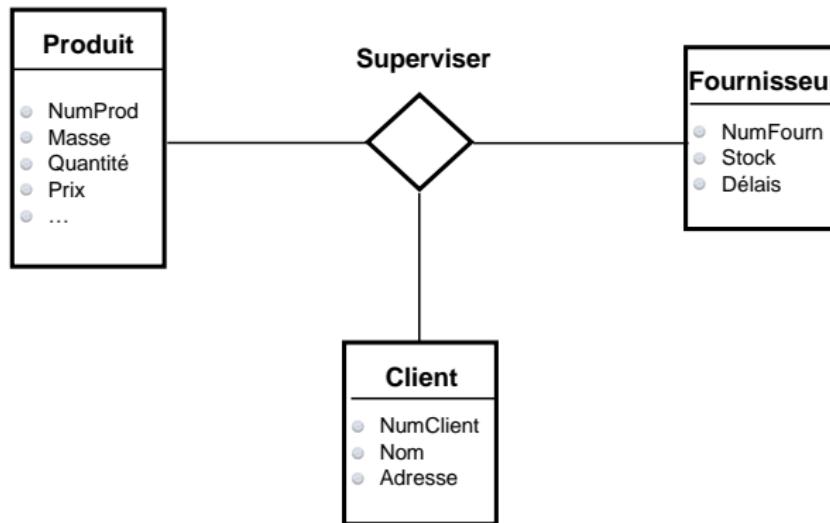
Exemple : cela peut être le cas si l'on raisonne sur la classe PERSONNE avec la relation *Responsable* dans le cadre d'une entreprise.



# Association n-aires

Une association reliant plus de deux classes.

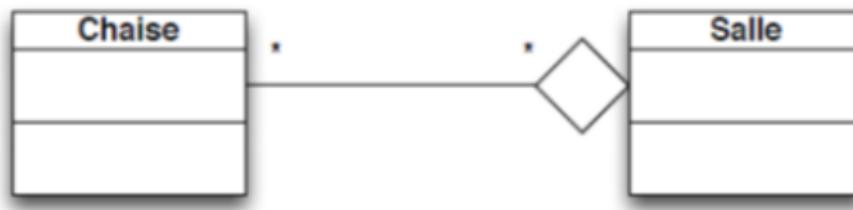
Exemple d'une association ternaire : commerce



Ne pas abuser des associations ternaires.

# Agrégation et composition

Agrégation : montre qu'une classe fait partie d'une autre classe.



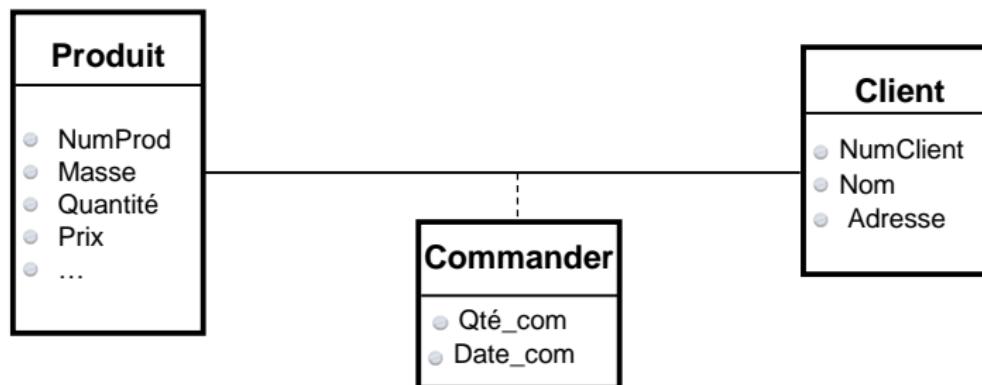
Composition (agrégation composite ou agrégation forte) : cas particulier de l'agrégation.



# Les classes-associations

Une classe permettant de paramétrer une association entre deux classes.

Exemple d'une classe-association



# La multiplicité

## Définition

La multiplicité : montre le nombre d'objets liés par une association.

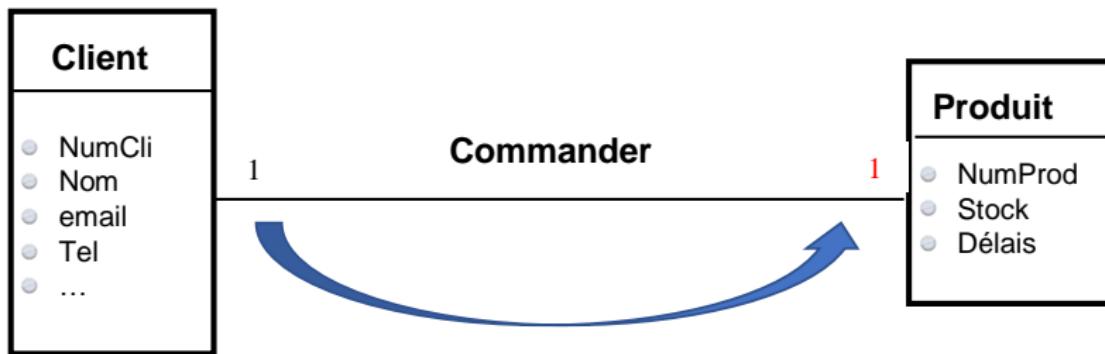
De façon plus précise, on indique les participations minimales et maximales d'un objet à l'association.

- n : exactement n (1..1 ou 1)
- n..m : de n à m
- 1..\* : au moins 1
- 0..\* ou \* : aucun ou plusieurs
- n..\* : de n à plusieurs

# Exemple

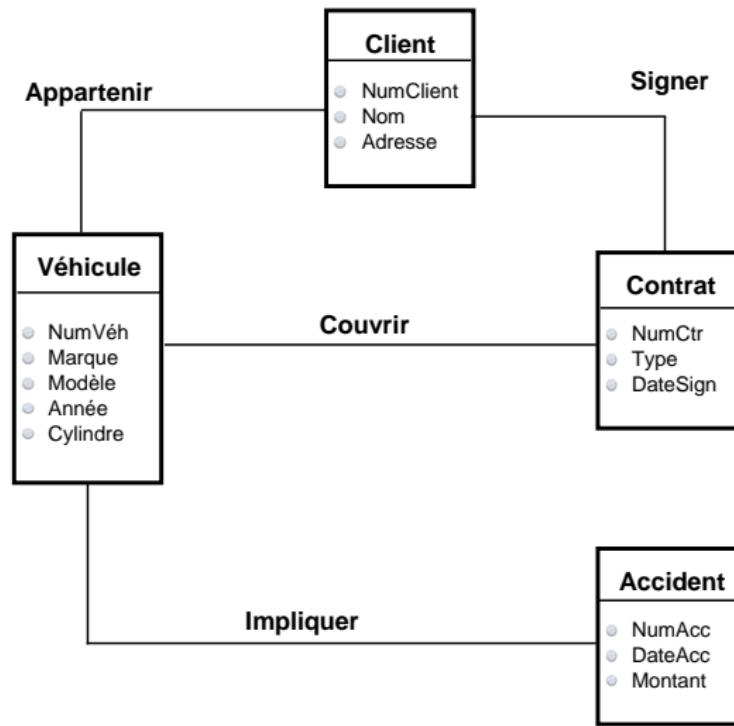
## Association 1..1

Un client donné ne commande qu'un seul produit. Un produit donné n'est commandé que par un seul client.

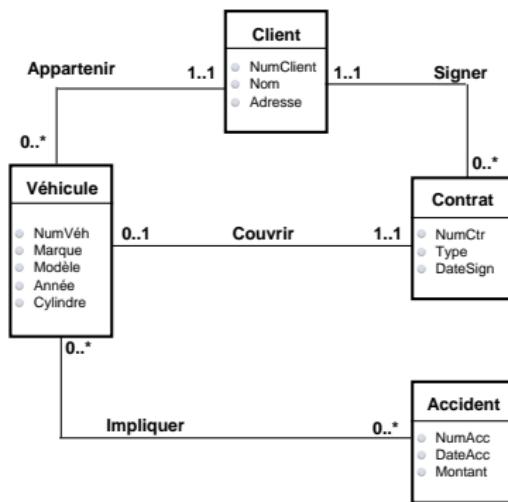


# Exercice

Indiquer les multiplicités sur le diagramme de classes suivant :



# Solution



- un client **peut ne pas posséder de véhicule ou en posséder plusieurs**
- un véhicule appartient à **un seul et unique** client
- un accident peut **ne pas** ou impliquer **plusieurs** véhicules
- un véhicule est couvert par **une seul et unique** assurance
- un contrat est signé par **un seul et unique** client
- un contrat couvre **au maximum** un véhicule
- ...

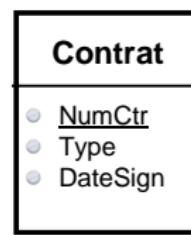
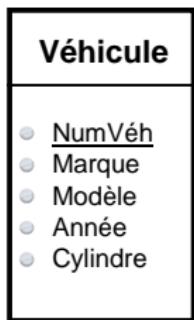
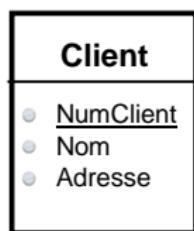
# Identifiants

Un identifiant caractérise de façon unique les objets d'une classe.

Le plus souvent, cet identifiant constitue la **clé primaire** de la table.

**Notations** : par convention graphique l'identifiant est souligné dans la liste des attributs.

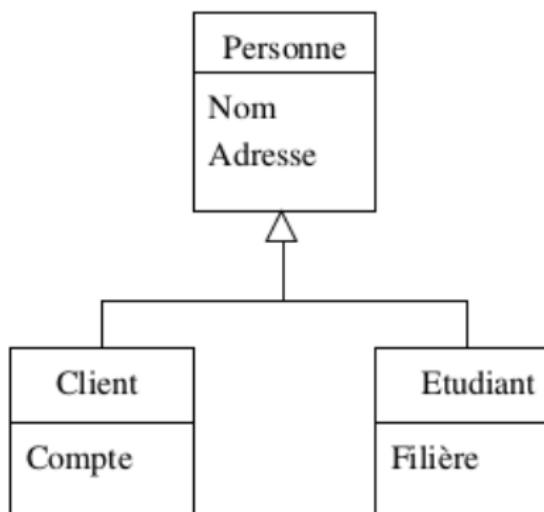
# Identifiants



# L'héritage (Relation de généralisation)

## Définition

Désigne la relation de classification entre un élément général et un élément plus spécifique.



Association est un liant sous-classe et super-classe

# Les étapes pour établir un diagramme de classes

## Démarche à suivre :

- Identifier les objets de même type: les classes
- Identifier les attributs
- Associer les attributs aux différentes classes
- Identifier les associations entre les classes
- Identifier les attributs de chaque association
- Évaluer la multiplicité des associations

**Il est important de bien lire l'énoncé, de bien analyser le contexte de votre étude afin d'identifier clairement les éléments importants qui pourront vous servir à la construction de votre modèle.**

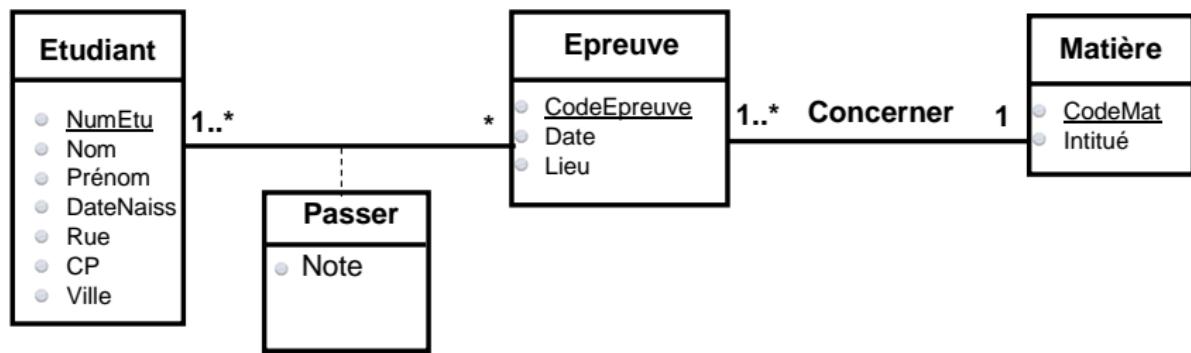
## Exemple (1/2)

**Considérons une base de données relative à la notion d'Examen.**

Proposer un diagramme de classe à partir de l'énoncé suivant

- Les étudiants d'une université sont caractérisés par un numéro unique, leur nom, prénom, date de naissance, rue, code postal et ville.
- Ils passent des épreuves et obtiennent une note pour chacune.
- Les épreuves sont caractérisée par un code ainsi que la date et le lieu auxquels elles se déroulent.
- Chaque épreuve relève d'une matière unique. En revanche, une matière peut donner lieu à plusieurs épreuves.
- Les matières sont caractérisées par un code et un intitulé.

# Solution : Diagramme de classes

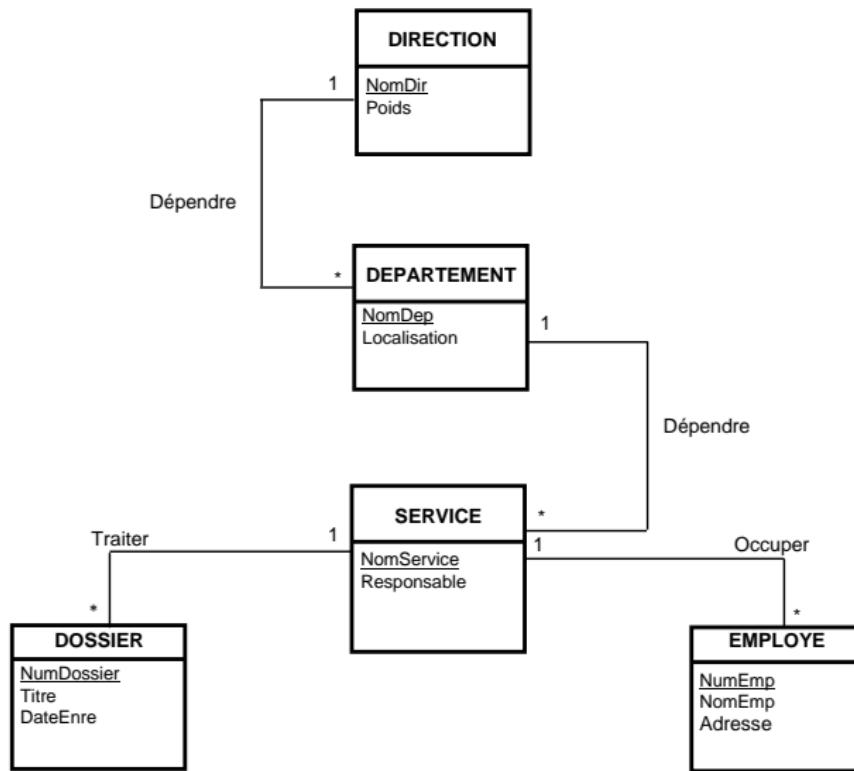


## Exemple (2/2): une structure administrative

**Exemple :** On considère un sous-ensemble d'une structure administrative. D'une direction (caractérisée par un nom identifiant et le nom de son président-directeur général) dépendent plusieurs départements (dotés chacun d'un nom identifiant dans sa direction et de sa localisation). Un département est découpé en services, dotés chacun d'un nom (identifiant de son département) et d'un responsable. Un service a la charge d'un certain nombre de dossiers identifiés par un numéro et dotés d'un titre et d'une date d'enregistrement/ Dans chaque service travaillent des employés identifiés par un numéro et caractérisés par leur nom et par leur adresse.

Proposer un diagramme de classes à cet spécification.

# Solution Diagramme de classe



# Modèle Relationnel

# Modèle Relationnel

- E.F. CODD en 1970 (IBM)
- Un modèle Basé principalement sur deux fondements mathématiques :
  - Théorie des ensembles
  - Logique des prédicats
- des relations (tables) pour enregistrer les données

# Qu'est-ce qu'un SGBD ?

- Système de Gestion de Bases de Données (SGBD) (*DataBase Management Systems (DBMS)*)
- Un ensemble de logiciels permettant aux utilisateurs de définir, créer, maintenir, contrôler et accéder à la BD.
- MySQL, PostgreSQL, ORACLE, SQLServer, DB2, Informix, Sybase, Teradata, ...

# Objectifs des SGBD relationnels

- **Indépendance physique** : un remaniement de l'organisation physique des données n'entraîne pas de modification dans les programmes d'applications
- **Indépendance logique** : un remaniement de l'organisation logique des fichiers n'entraîne pas de modifications dans les programmes d'applications concernés
- **Manipulation facile des données** : un utilisateur néophyte ou profane (*i.e.* non informaticien) doit pouvoir manipuler simplement les données (interrogation et éventuellement mise à jour de la base de données)
- **Administration facile des données** : un SGBD doit fournir des outils pour décrire les données, permettre le suivi de ces structures et autoriser leur évolution (tâche concernant plutôt l'administrateur des bases de données).

# Objectifs des SGBD relationnels

- **Efficacité des accès aux données** : garantie d'un bon débit (nombre de transactions tâches effectuées par seconde) et d'un bon temps de réponse (temps moyen d'attente pour effectuer une tâche)
- **Redondance contrôlée des données** : diminution du volume de stockage, pas de mise à jour multiple, ni d'incohérence
- **Cohérence des données** : l'âge d'une personne doit être un entier strictement positif. le SGBD doit vérifier que les applications respectent certaines règles (contraintes d'intégrités)
- **Partage des données** : utilisation simultanée des données par différentes applications
- **Sécurité des données** : les données doivent être protégées contre les accès non autorisés ou en cas de panne

# Schéma relationnel

Il est important de garder le formalisme suivant en tête :

- Une BD relationnelle est une collection de **relations** (Tables)
- Attributs = Champs/Colonnes d'une table
- Tuples = n-uplets/Enregistrements/Lignes. Remarque: une valeur d'un attribut peut être la valeur nulle (NULL)
- le schéma d'une base de données relationnelle est donc un **ensemble de schémas de relation que l'on notera  $R_i$**
- le schéma d'une relation est constitué **du nom de la relation et d'un ensemble d'attributs** formant un  $n - uplet$  (ici  $n = r_i$ )

$$R_i = (A_1, A_2, \dots, A_{r_i})$$

## Schéma relationnel : exemple

- Le relation EPREUVE a l'ensemble des attributs {Code Epreuve, Date, Lieu}
- Chaque attribut  $A_i$  prend ses valeurs dans un certain domaine  
 $\text{Lieu} \in \{\text{'Amphi 136'}, \text{'K073'}, \text{'Salle 201'}, \text{'Salle 01'}, \dots\}$  ( $\Rightarrow$  contrainte de domaine)
- Extension : ensemble de n-uplets  
 $\langle \text{BDL3}, 12/03/2022, \text{'Amphi 136'} \rangle$   
 $\langle \text{ProgL3}, 13/03/2022, \text{'Amphi 136'} \rangle$   
...

# Contraintes d'intégrité

**Contraintes d'intégrités** : C'est les règles que les données doivent vérifier.

- **Toute relation une clé primaire** : ensemble d'attributs dont les permettent de distinguer les n-uplets les uns des autres. Il ne peut en aucun admettre la valeur Null.  
Ex : CodeEpreuve est une primaire de la relation EPREUVE.
- **Clé étrangère** : il s'agit d'un attribut qui est clé primaire d'une autre relation.
- **Contraintes de domaine**) : les attributs doivent respecter une condition logique. Ex : Note  $\in [0, 20]$ .

# Quelques règles

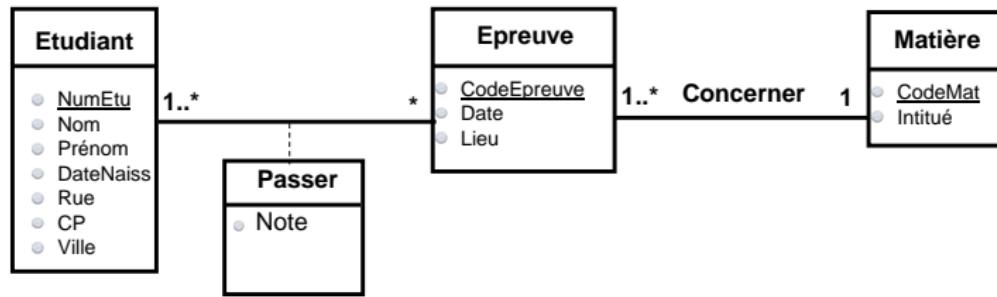
Une relation est définie par :

- son nom
- son  $n - uplets$ , i.e. sa liste d'attributs qui doit respecter des contraintes de domaines (on ne les mentionne pas explicitement dans le schéma)
- son ou ses identifiants (clé primaire, étrangère). Il est d'usage de souligner la clé primaire et de faire précéder d'un # la ou les clé(s) étrangère(s)

Ex : EPREUVE (CodeEpreuve, Date, Lieu, #CodeMat)

# Passage d'un diagramme de classes à un schéma relationnel

Repartons du diagramme de classes concernant les examens :



# Règle 0

Chaque **classe** devient une **relation**.

Les **attributs** de la classe deviennent **attributs** de la relation.

Enfin, **l'identifiant** de la classe devient **clé primaire** de la relation

**Exemple :**

ETUDIANT (NumEtu, Nom, Prénom, DateNaiss, Rue, CP, Ville)

## Règle 1

Chaque **association avec multiplicité maximales 1-1** est prise en compte en incluant la **clé primaire** d'une des relations comme clé étrangère dans l'autre relation.

**Exemple :** Si un étudiant peut posséder une seul et unique carte étudiant, on aura :

CARTE(NumCarte, Crédit, ...)

ETUDIANT(NumEtu, Nom, Prénom, DatNaiss, Rue, CP, Ville,  
#NumCarte)

## Règle 2

Chaque **association avec multiplicité maximales 1-N** est prise en compte en incluant la **clé primaire** de la relation dont la multiplicité maximale est **N** comme **clé étrangère** dans l'autre relation.

**Exemple :** Entre les relations EPREUVE et MATIERE, on aura :

EPREUVE (CodeEpreuve, Date, Lieu, #CodeMat)

MATIERE (CodeMat, Intitulé)

## Règle 3

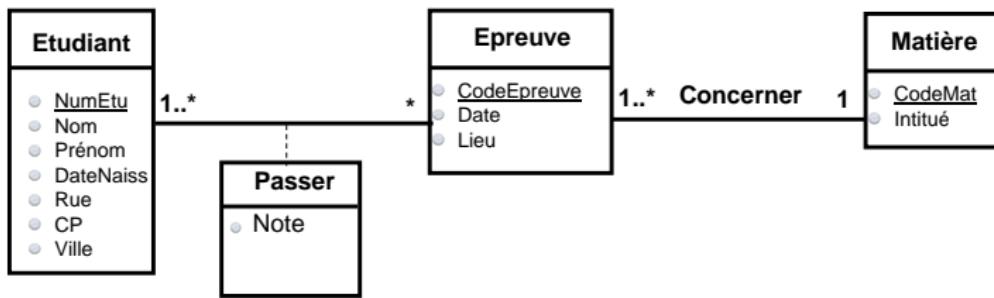
Dans le cas d'associations de multiplicité maximales **plusieurs - plusieurs**, *i.e.* **N-N**, cette dernière est prise en compte dans le schéma relationnel de la façon suivante

- **création d'une nouvelle relation** dont la **clé primaire** est la concaténation des clés primaires des relations participantes
- les attributs de l'association sont insérés dans cette nouvelle relation si nécessaire

**Exemple :** L'association PASSE devient alors

PASSE(#NumEtu,# CodeEpreuve, Note)

# Passage au modèle relationnel : exemple



# Passage au modèle relationnel : exemple

Le modèle relationnel associé serait le suivant :

ETUDIANT (NumEtu, Nom, Prénom, DateNaiss, Rue, CP, Ville)

PASSE(#NumEtu,# CodeEpreuve, Note)

EPREUVE(CodeEpreuve, Date, Lieu, # CodeMat)

MATIERE(CodeMat, Intitulé)

## Règle 5: Traduire l'héritage

- **Opération délicate**

Vous pouvez traduire une association d'héritage par :

- **Distinction:** en créant autant de relations que de classes (surclasse et sous-classes). Chaque relation de l'ensemble possède la même clé primaire. Cette même clé joue le rôle de clé étrangère vers la relation déduite de la surclasse.
- **Push-down :** en migrant tous les attributs de la surclasse dans les sous-classes.
- **Push-up :** en migrant tous les attributs des sous classes dans la surclasse.

# Traduire l'héritage : exemple

- Tableau

# Qu'est-ce qu'une Base de données correcte ?

Il s'agit d'un ensemble de relations tel que :

- Chaque relation décrit un fait élémentaire avec les seuls attributs qui lui sont directement liés
- Il n'y a pas de redondance d'information, génératrices de problèmes lors des mises à jour
- Il n'y a pas de perte d'information  
PERSONNE(Nom, Prénom)  
ADRESSE(No, Rue, Ville)  
Qui habite où ? Impossible de répondre.

# Normalisation

## Qu'est-ce que la normalisation ?

- C'est un processus de transformation d'une relation qui pose des problèmes lors des mises à jour en des relations ne posant pas de problèmes
- On mesure la qualité d'une relation par son degré de normalisation
- Diverses formes normales

Première forme normale

Deuxième forme normale

Troisième forme normale

FNBC

Quatrième forme normale

# Qu'est-ce qu'une Base de données incorrecte ?

Une relation n'est pas correcte si :

- elle implique des répétitions au niveau de sa population
- elle pose des problèmes lors des mises à jour (insertions, modifications, suppressions)
- les conditions pour qu'une relation soit correcte peuvent être définies formellement :

→ Règles de normalisation

# Vérification du modèle relationnel

## Première forme normale

- Une relation est en première forme normale si tous les attributs ne sont pas décomposables (on dit aussi que tous les attributs sont **atomiques**, donc pas **multi-valués**)
- **Exemple :**

La relation DEPARTEMENT(Nom, Adresse, Tel) n'est pas en première forme normale si les attributs Nom et Adresse peuvent être du type [*Jean Paul*] ou [*Rue de Marseille, 69003 Lyon*].

# Vérification du modèle relationnel

## Deuxième forme normale

- Une relation est en deuxième forme normale si elle est en première forme normale et si tout attribut non clé primaire dépend entièrement de la clé primaire (à vérifier dans le cas où l'identifiant est composé)

### Exemples :

La relation CLIENT(NumCli, Nom, Prenom, DateNaiss, Rue, Cp, Ville) est en deuxième forme normale.

La relation COMMANDE(NumProd, NumFour, Quantite, VilleFour) n'est pas en deuxième forme normale car seul *NumFour* → *Ville*

La décomposition suivante donne deux relations en deuxième forme normale :

COMMANDE (NumProd, NumFour, Quantité) FOURNISSEUR(NumFour, Ville Four)

# Vérification du modèle relationnel

## Troisième forme normale

- Une relation est en troisième forme normale si elle est en deuxième forme normale et s'il n'existe aucune dépendance fonctionnelle entre deux attributs non clé primaire.

### Exemples :

La relation MUSIQUE(NoChanson, # NoChanteur, NomChanteur) n'est pas en troisième forme normale

En effet,  $\text{NoChanteur} \rightarrow \text{Nom}$  La décomposition suivante donne deux

relations en troisième forme normale :

R1(NoChanson, # NoChanteur) R2( NoChanteur, Nom)

# Dépendances fonctionnelles

- Soit  $R(X, Y, Z)$  une relation où  $X, Y, Z$  sont des ensembles d'attributs.  $Z$  peut être vide.
- **Définition :**  $Y$  est en dépendance fonctionnelle avec  $X$  ( $X \rightarrow Y$ ) si pour une valeur de  $X$  on détermine une seule valeur de  $Y$  dans la relation  $R$ .
- **Exemple :**  $\text{PRODUIT}(\text{NumProd}, \text{Dési}, \text{Prix})$

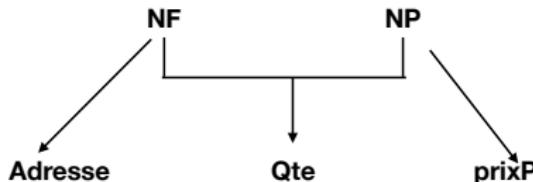
Deux dépendances fonctionnelles possibles :

$\text{NumProd} \rightarrow \text{Dési}$   $\text{NumProd} \rightarrow \text{Prix}$

# Graphe des dépendances fonctionnelles

Il est intéressant, pour chaque relation, de connaître les dépendances fonctionnelles. On peut alors les représenter sous forme de graphes. On considère la relation LIVRAISON(NF, Adresse, NP, PrixP, Qte)

- NF → Adresse : l'adresse du fournisseur ne dépend que du fournisseur
- NP → PrixP : le prix d'un produit ne dépend que du produit
- (NF, NP) → Qte : la quantité livrée dépend du produit et du fournisseur
- [faux : NF → Qté, NP → Qté]



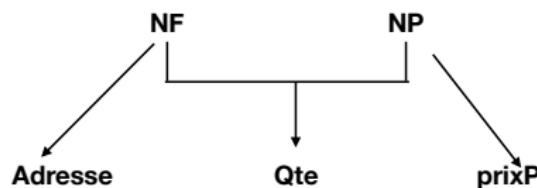
# Dépendance fonctionnelle et identifiants

- Le graphe minimum des dépendances fonctionnelles permet de trouver les identifiants de la table
- L'identifiant d'une table est l'ensemble (minimal) des nœuds du graphe minimum à partir desquels on peut atteindre tous les autres nœuds (via les dépendances fonctionnelles)
- Pour que cela soit faux, il faudrait qu'il y ait deux lignes avec la même valeur de l'identifiant et des valeurs différentes pour les autres attributs, ce qui est en contradiction avec les dépendances fonctionnelles.

Ainsi, dans l'exemple précédent, on remarque la relation n'est pas normalisée, deux identifiants :  $NF$  et  $NP$ . Il va donc falloir la décomposer, i.e. la remplacer par un ensemble de relations.

## Méthode pragmatique

Repartons de notre relation LIVRAISON(NF, Adresse, NP, prixP, Qte) et du graphe des dépendances fonctionnelles associé.



On a deux identifiants : *NF* et *NP* avec les dépendances décrites dans le graphe ci-dessus.

Une bonne décomposition se ferait alors de la façon suivante :

- $NP \rightarrow prixP$  se traduit par PRODUIT(NP, prixP)
- $NF \rightarrow Adresse$  se traduit par FOURNISSEUR(NF, Adresse)
- $NP, NF \rightarrow Qte$  se traduit par LIVRAISON(NF, NP, Qte)

# Méthode formelle de décomposition

## Théorème de Heath

$T(X,Y,Z)$  est décomposable sans perte d'information en

 $T_1(X,Y)$ 

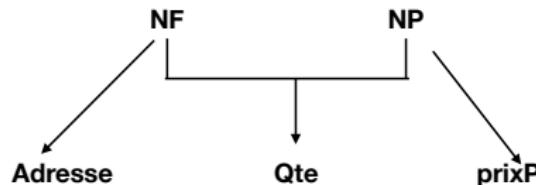
et

 $T_2(X,Z)$ 

si  $X$  est en dépendance fonctionnel avec  $Y$ .

# Application de Heath

Reprendons notre relation LIVRAISON(NF, NP, Adresse, prixP, Qte)



- NF → Adresse, on peut décomposer LIVRAISON en L1 (NF, Adresse)  
L2 (NF, NP, prixP, Qte)
- NP → prixP, on peut décomposer L2 en L3 (NP, prixP) L4 (NP, NF, Qte)
- et on a bien la dépendance fonctionnelle (NP, NF) → Qte.

# Propriétés des DF

## Règles d'inférence d'Armstrong

- Réflexivité:  $\forall X, X \rightarrow X$
- Augmentation: Étant donnée  $X \rightarrow Y$  alors  $\forall A, XA \rightarrow Y$
- Transitivité: Étant données  $X \rightarrow Y$  et  $Y \rightarrow Z$  alors  $X \rightarrow Z$
- Peudo-transitivité: Étant données  $X \rightarrow Y$  et  $WY \rightarrow Z$  alors  $XW \rightarrow Z$
- Fusion: Étant données  $X \rightarrow Y$  et  $X \rightarrow Z$  alors  $X \rightarrow YZ$
- Éclatement: Étant donnée  $X \rightarrow YZ$  alors  $X \rightarrow Y$  et  $X \rightarrow Z$

# Les transactions<sup>1</sup>

L'utilisateur interagit avec la BD via le SGBD en utilisant des requête SQL.  
Le SGBD garantie les propriété ACID :

- Atomicité : L'exécution d'une requête est atomique.
- Cohérence : Une requête exécutée sur des données cohérentes laisse celles-ci dans un état final également cohérent.
- Isolation : Les requêtes sur les données sont exécutées comme si chaque requête disposait de la base de données pour elle seule. le SGBD gère les opérations concurrentes.
- Durabilité : Le SGBD assure que lorsqu'une mise à jour est effectuée, elle est permanente.

---

<sup>1</sup>Livre base de données de Jean-Luc Hainaut

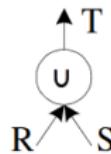
# L'algèbre relationnelle

# Algèbre relationnelle

- Ensemble d'opérateurs qui s'appliquent aux relations
- Opérateurs de l'algèbre relationnelle :
  - Les opérateurs ensemblistes : union, intersection, différence, produit cartésien.
  - Des opérateurs spécifiques aux BD : sélection, projection, renommage, jointure, division.

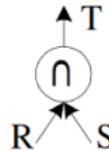
# Union U

- Soient R et S deux relations
- $T = R \cup S$  ou  $T = \text{UNION}(R, S)$  crée une relation comprenant tous les tuples existants dans l'une ou l'autre des relations R et S
- Les 2 relations doivent avoir le même nombre d'attributs, et les mêmes types
- Par convention, la relation résultat possède le schéma du premier opérande (R)
- Élimination des doublon



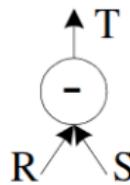
# Intersection $\cap$

- Soient R et S deux relations
- $T=R \cap S$  ou  $T = \text{INTERSECT } (R, S)$  crée une nouvelle relation de même schéma et de population égale à l'ensemble des tuples de R tels qu'il existe un tuple de même valeur dans S
- Les 2 relations doivent avoir le même nombre d'attributs, et les mêmes types
- Par convention, la relation résultat possède le schéma du premier opérande (R)



# Différence – ou \

- Soient R et S deux relations
- $T = R - S$  ou  $T = \text{MINUS}(R, S)$  crée une relation de même schéma et de population égale à l'ensemble des tuples de R moins ceux de S (les tuples qui se trouvent dans R mais pas dans S)
- Les 2 relations doivent avoir le même nombre d'attributs, et les mêmes types
- Par convention, la relation résultat possède le schéma du premier opérande (R)



# Exemples

R1

A1	A2	A3
a1	a2	a3
b1	b2	b3
c1	c2	c3
d1	d2	d3

\* R2

A1	A2	A3
a1	a2	a3
e1	e2	e3
b1	b2	b3

UNION  
R1 ∪ R2

A1	A2	A3
a1	a2	a3
b1	b2	b3
c1	c2	c3
d1	d2	d3
e1	e2	e3

R1

A1	A2	A3
a1	a2	a3
b1	b2	b3
c1	c2	c3
d1	d2	d3

\* R2

A1	A2	A3
a1	a2	a3
e1	e2	e3
b1	b2	b3

INTERSECTION  
R1 ∩ R2

A1	A2	A3
a1	a2	a3
b1	b2	b3

R1

A1	A2	A3
a1	a2	a3
b1	b2	b3
c1	c2	c3
d1	d2	d3

R2

A1	A2	A3
a1	a2	a3
e1	e2	e3
b1	b2	b3

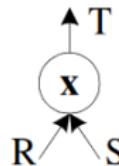
DIFFERENCE

R1 - R2

A1	A2	A3
c1	c2	c3
d1	d2	d3

# Produit cartésien x

- Soient R et S deux relations
- $T = R \times S$  ou  $T = \text{PRODUCT}(R, S)$  crée une nouvelle relation où chaque tuple de R est associé à chaque tuple de S
- Le nombre de tuples est égal à  $|R| \times |S|$  (où  $|R|$  est le nombre de tuples dans la relation R)



# Exemple

IdProf	matière
3115	Anglais
3207	BD

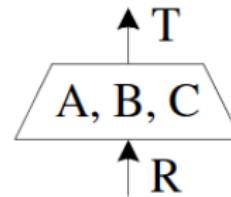
x

IdPromo	type	nbreEleves
22	TP	40
21	Cours	60
20	TD	30

IdProf	matière	IdPromo	type	nbreEleves
3115	Anglais	22	TP	40
3115	Anglais	21	Cours	60
3115	Anglais	20	TD	30
3207	BD	22	TP	40
3207	BD	21	Cours	60
3207	BD	20	TD	30

# Projection $\Pi$

- Soit  $R$  une relation
- L'opérateur projection, noté  $\Pi$ , consiste à créer une relation à partir d'une autre en ne gardant que les attributs spécifiés dans la projection.
- $T = \Pi< A, B, C > (R)$  ou  $T = \text{PROJECT} (R / A, B, C)$



# Exemple

$$\Pi_{matière} ($$

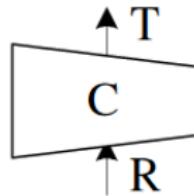
IdProf	matière
3121	Maths
3116	BD
3115	Anglais

$$)$$
 =

matière
Maths
BD
Anglais

# Sélection (Restriction) $\sigma$

- Soit R une relation
- L'opérateur de sélection (ou restriction), noté  $\sigma$ , consiste à créer une relation à partir d'une autre en ne gardant que les tuples pour lesquels un attribut vérifie certaines conditions.
- $T = \sigma<C>(R)$  ou  $T = \text{RESTRICT } (R / C)$



# Exemple

$$\sigma_{\text{matière}=\text{'Anglais'}}($$

IdProf	matière
3121	Maths
3116	BD
3115	Anglais

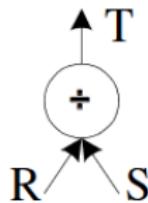
)

=

IdProf	matière
3115	Anglais

# Division $\div$

- Soient R et S deux relations
- R (A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>) et S (A<sub>p+1</sub>, ..., A<sub>n</sub>)
- T = R  $\div$  S ou T = DIVISION (R, S)
- T(A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>p</sub>) contient tous les tuples tels que leur concaténation à chacun des tuples de S donne toujours un tuple de R



# Exemple

R : obtenu

Etudiant	Cours
Alain	BD
Alain	Maths
Alain	Java
Annie	Anglais
Alban	Java
Alban	BD

V : Pré-requis

Cours
Java
BD

÷

=

$R \div V$

Etudiant
Alain
Alban

$R \div V$  : les étudiants ayant tous les pré-requis.

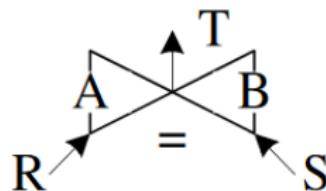
# Renommage $\rho$

- Soit R une relation
- L'opérateur de renommage, noté  $\rho$ , permet de renommer les attributs d'une relation. Renommage d'un ou plusieurs attributs d'une relation R, avec A1 qui devient A'1, ..., Ak qui devient A'k

$$\rho_{A'1/A1, \dots, A'k/Ak}(R)$$

# Jointure naturelle $\bowtie$

- $T = R \bowtie S$  ou  $T = \text{JOIN } (R, S)$
- Produit cartésien  $R \times S$  et restriction  $A = B$  sur les attributs  $A \in R$  et  $B \in S$



# Exemple

IdPers	Nom	Prenom	IdAdr
1021	Dupont	Bob	8
1022	Durant	Alice	9
1023	Martin	Fred	7



IdPers=IdEleve

IdEleve	Année	Dpt
1021	3	GBM
1024	4	INFO

=

IdPers	Nom	Prenom	IdAdr	IdEleve	Année	Dpt
1021	Dupont	Bob	8	1021	3	GBM

# $\theta$ -jointure

- $T = R \theta_C S$  ou  $T = \text{JOIN}_C (R, S)$
- Produit cartésien  $R \times S$  et restriction selon la condition  $C$
- Utile lorsqu'il n'y a pas d'attributs en commun entre  $R$  et  $S$  ou pour des comparaisons autres que l'égalité

# Requête algébrique

- Une requête en algèbre relationnel est une combinaison d'opérateurs
- Soient deux relations  $R(a,b)$  et  $S(a,c)$ , quelques requêtes :

La liste des a de la relation R :  $\sqcap_a(R)$

Les b de la relation R pour les tuples dont le a vaut 'abcde' :

$$\sqcap_b(\sigma_{a='abcde'}(R))$$

Les b et c des tuples de R et S qui ont la même valeur pour a :

$$\sqcap_{b,c}(R \theta_a S)$$

# Langage SQL

# Avant propos

- SQL **Structured Query Language** ce qui signifie *Langage de requête structurée* en français
- SQL permet la définition, la manipulation et le contrôle d'une base de données relationnelle. Il se base sur l'algèbre relationnelle
- Norme internationale depuis 1987 (ISO 9075).
- SQL = LDD + LMD + LCD

# Avant propos

Le langage SQL se décompose en trois parties :

- **Le langage DDL (Data Description Language)** : est la partie qui est consacrée à *la définition ou la modification de structure*. C'est donc la partie qui va nous permettre de définir des tables, de la modifier, de définir la nature des colonnes de notre table et éventuellement les contraintes qui y sont associées.
- **Le langage DML (Data Manipulation Language)** : est la partie qui consacrée à la *manipulation des données* (extraction et modification).
- **Le langage DCL (Data Control Language)** : Commandes SQL pour contrôler des données, attribution de droits d'accès, ...

# Choix de PostgreSQL

- SGBD relationnel, mais aussi relationnel-objet
- Licence libre
- Multi-plateformes: Linux, Windows, Mac OS
- Largement utilisé : par Cisco, skype, ...



# Le langage DDL

Cette partie est à la base d'un SGBD et va permettre de mettre en place **toute la structure de la base de données** :

- définir une table
- définir la nature des éléments de cette table, *i.e.* le type des attributs
- modifier une table existante en ajoutant, supprimant des colonnes
- supprimer une table
- prendre en compte les différentes contraintes d'intégrités
- ajouter des structures physiques
- ...

## Création d'une table

La création d'une table se fait à l'aide de CREATE TABLE. La commande générique est la suivante

```
CREATE TABLE nom-table(  
    id0 type-données PRIMARY KEY,  
    colonne1 type-données DEFAULT val,  
    colonne2 type-données  
    colonne3 type-données,  
    ...  
)
```

On peut aussi créer une table à partir d'une table existante

```
CREATE TABLE Table2 AS (  
SELECT Attr1, Attr2, ...  
FROM Table1  
)
```

# Nature des attributs ou type des données

- **NUMERIC(n)** : nombre entier à n chiffres
- **NUMERIC(n, m)** : nombre réel à n chiffres au total (virgule comprise) et m chiffres après la virgule
- **VARCHAR(n)** : chaîne de caractères de taille n
- **DATE** : date au format 'AAAA-MM-JJ'
- Liste complète:

<https://docs.postgresql.fr/8.1/datatype.html>

**Choisir le meilleur type possible.**

**Pour éviter: Le gaspillage d'espace mémoire. Les problèmes des fonctionnalités liées aux types de données.**

# Création de type : CREATE TYPE

## Exemple

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

# Contraintes sur les colonnes et tables

## Sur les **colonnes ou attributs**

- NOT NULL
- UNIQUE
- PRIMARY KEY
- REFERENCES nom-table
- CHECK (condition)

## Sur une **table**

- UNIQUE(nom-col)
- PRIMARY KEY (nom-col)
- FOREIGN KEY (nom-col) REFERENCES nom-table nom-col
- CHECK (condition)

## Quelques exemples de création de tables

- **Exemple 1 :** créer une table *client* avec les attributs *NumCli*, *Nom*, *DateNaiss*, *telephone*, *solvable* en attribuant les types adéquats à chacun d'eux et la clef primaire au bon attribut.

## Quelques exemples de création de tables

- **Exemple 1 :** créer une table *client* avec les attributs *NumCli*, *Nom*, *DateNaiss*, *telephone*, *solvable* en attribuant les types adéquats à chacun d'eux et la clef primaire au bon attribut.

```
CREATE TABLE CLIENT (
    NumCli NUMERIC(8),
    Nom VARCHAR(30),
    DateNaiss DATE,
    Telephone CHAR(10),
    Solvable BOOLEAN,
    PRIMARY KEY (NumCli)
)
```

## Quelques exemples de création de tables

### ● Exemple 2 :

On considère une première table *service* avec les attributs *NumService* et *Nom Service*.

On considère également la table *employé* avec les attributs *mat*, *nomEmploye*, *fonction*, *dateEmbauche*, *salaire*, *commission*, *numService*

Ecrire les requêtes permettant de définir chacune de ces tables en prenant soit d'indiquer le type de chaque attribut, les clefs primaires de chaque table ainsi que les références étrangères s'il y a lieu.

## Quelques exemples de création de tables

### ● Exemple 2 :

On considère une première table *service* avec les attributs *NumService* et *Nom Service*.

On considère également la table *employé* avec les attributs *mat*, *nomEmploye*, *fonction*, *dateEmbauche*, *salaire*, *commission*, *numService*

Ecrire les requêtes permettant de définir chacune de ces tables en prenant soit d'indiquer le type de chaque attribut, les clefs primaires de chaque table ainsi que les références étrangères s'il y a lieu.

```
CREATE TABLE SERVICE (
    NumService INTEGER,
    NomService VARCHAR(30),
    PRIMARY KEY(NumService)
)
```

## Quelques exemples de création de tables

- Exemple 2 (suite) :

```
CREATE TABLE EMPLOYE (
    mat INTEGER,
    nomEmploye VARCHAR(15),
    fonction VARCHAR(15),
    dateEmbauche DATE,
    salaire NUMERIC(7,2),
    comission NUMERIC(7,2),
    NomService INTEGER,
    PRIMARY KEY(mat),
    FOREIGN KEY(NomService) REFERENCES Service(NumService)
)
```

# Modification structure existante

- Ajout d'attributs

```
ALTER TABLE nom_table ADD attribut type (taille), ...
```

- Exemple 1

```
ALTER TABLE Client ADD telephone CHAR(8);
```

- Exemple 2

```
ALTER TABLE Produit ADD CodeCategorie INTEGER;  
ALTER TABLE Produit ADD FOREIGN KEY (CODECategorie)  
REFERENCES Categorie (CodeCatégorie);
```

# Modification structure existante

- **Modification d'attributs**

```
ALTER TABLE nom_table MODIFY attribut type (taille),...
```

- **Exemple 1**

```
ALTER TABLE client MODIFY telephone CHAR(10);
```

- **Exemple 2**

```
ALTER TABLE Employe MODIFY Salaire MONEY;
```

# Modification structure existante

- **Suppression d'attributs**

```
ALTER TABLE nom_table DROP attribut
```

- **Exemple 1**

```
ALTER TABLE Client DROP DateNaissance;
```

- **Exemple 2**

```
ALTER TABLE Employe DROP NumService;
```

Attention ! Cette suppression est possible uniquement si la relation entre la table Employé et la table Service est cassée. Dans le cas contraire, la SGBD vous empêchera de faire cette manipulation.

# Modification structure existante

- Suppression d'une table

```
DROP TABLE nom_table
```

- Exemple 1

```
DROP TABLE Services
```

Attention ! Cette suppression est possible uniquement si la table *Services* n'est pas reliée à une autre table de la base de données.

## Actions déclenchées

Nous avons vu que des actions menées sur certaines tables peuvent avoir des répercussions sur des autres tables.

Par exemple, que se passerait-il quand on détruit/met à jour une clé primaire ou un attribut de type unique qui est référencé par un attribut FOREIGN KEY d'une autre table ?

**Exemple :**

```
CREATE TABLE Departement (
    numero_departement INTEGER PRIMARY KEY ,
    numero_manager INTEGER REFERENCES
        EMPLOYEE(numero_employe)
)
```

Quid en cas de mise à jour ?

# Actions déclenchées

## Deux circonstances

- ON DELETE
- ON UPDATE

## Trois possibilités d'actions

- SET NULL
- SET DEFAULT : valeur par défaut si existe, sinon NULL
- CASCADE : on répercute les m.a.j.

## Exemple

```
CREATE TABLE Departement (
    numero_departement INTEGER PRIMARY KEY,
    numero_manager INTEGER REFERENCES Employe(numero_emp)
        ON DELETE SET NULL
        ON UPDATE CASCADE
)
```

# Retour sur les contraintes -> CHECK

- Permet de mettre des contraintes sur les colonnes d'une table.

## Exemple :

```
CREATE TABLE divisions(
    num_div INTEGER CHECK (num_div between 10 AND 99),
    nom_div VARCHAR(9) CHECK (nom_div = UPPER(nom_div)),
    bureau VARCHAR(10) CHECK
    (bureau IN ('Lyon', 'Paris', 'Lille'))
)
```

# Retour sur les contraintes -> CHECK

- Un autre exemple de contraintes portant sur plusieurs colonnes

## Exemple :

```
CREATE TABLE Employe(
    numero_employe INTEGER PRIMARY KEY,
    nom_employe VARCHAR(10),
    nom_job VARCHAR(9),
    numero_manager INTEGER,
    salaire DECIMAL(7,2),
    commission DECIMAL(7,2),
    numero_departement SMALLINT(2),
    CONSTRAINT salaire_com CHECK (salaire+commission <=5000)
)
```

# Retour sur les manipulations de tables

```
DROP TABLE nom_table CASCADE CONSTRAINTS
```

## CASCADE CONSTRAINTS

- Supprime toutes les contraintes référençant une clé primaire (PRIMARY KEY) ou une clé unique (UNIQUE) de cette table
- Si on cherche à détruire une table dont certaines attributs sont référencés sans spécifier CASCADE CONSTRAINT, on a un message d'erreur.

# Retour sur les manipulations de tables

## Une structure plus complète

```
ALTER TABLE nom_table{
    ADD contrainte_table
    | DROP {
        PRIMARY KEY
        | UNIQUE (nom_colonne)
        | CONSTRAINT nom_contrainte  }
        CASCADE CONSTRAINTS
    | RENAME CONSTRAINT ancien TO nouveau
    | MODIFY CONSTRAINT nom_contrainte
        statut_contrainte
}
```

# Ajouter une contrainte

```
... ADD constraint_table
```

## Quelques exemples :

- on veut ajouter une contrainte à la table Employe

```
ALTER TABLE Employe  
    ADD CONSTRAINT seuil_commission  
        CHECK ((commission/salaire)<=1)
```

- on veut définir une clé primaire dans la table Pays (non définie plus tôt)

```
ALTER TABLE Pays  
    ADD CONSTRAINT cle_pays PRIMARY KEY (nom)
```

# Supprimer une contrainte

DROP

```
{PRIMARY KEY  
| UNIQUE nom_colonne  
| CONSTRAINT nom_contrainte}  
CASCADE CONSTRAINTS
```

## Exemple :

```
ALTER TABLE Departements  
DROP PRIMARY KEY CASCADE CONSTRAINTS
```

# Le langage DML

Le langage DDL permet donc de bien définir notre bases de données, la structure de nos différentes tables, attributs ainsi que les contraintes relatives à chaque tables et attributs.

Cette partie plutôt technique reste cependant très importante pour tout spécialiste en SGBD qui se respecte mais aussi pour modéliser un problème

# Le langage DML

Le langage DDL permet donc de bien définir notre bases de données, la structure de nos différentes tables, attributs ainsi que les contraintes relatives à chaque tables et attributs.

Cette partie plutôt technique reste cependant très importante pour tout spécialiste en SGBD qui se respecte mais aussi pour modéliser un problème

La partie qui va nous intéresser est susceptible d'intéresser un public plus important, même non initié.

# Manipulation des données

Trois commandes permettant d'effectuer des manipulations de base sur les données

- **INSERT INTO** : qui permet d'ajouter des valeurs dans une ligne d'une table
- **UPDATE** : qui permet de changer les valeurs dans une ligne d'une table
- **DELETE FROM** : qui permet de supprimer les valeurs d'une ligne d'une table

# Insérer des données

## Ajout d'un enregistrement

```
INSERT INTO nom_table (A1, A2, ...) VALUES (V1, V2, ...)
```

ou, si l'on souhaite remplir tous les attributs

```
INSERT INTO nom_table VALUES (V1, V2, ...)
```

## Exemples

- ```
INSERT INTO Produit  
VALUES (400, "chemise", 78)
```
- ```
INSERT INTO Vendeur (Nom, Prenom, Comm, Fixe, num_secteur)  
VALUES ("Chalbier", "Germaine", 0.03, NULL, 1)
```

# Modifier des valeurs existantes

## Mettre à jour un enregistrement

UPDATE nom\_table SET Attribut = valeur ou sous requête

ou, si l'on souhaite uniquement modifier les lignes respectant une certaine condition

UPDATE nom\_table SET Attribut = valeur ou sous requête  
WHERE conditions

## Exemples

- UPDATE Employe  
SET Salaire = 1000;
- UPDATE Etudiant  
SET Note = 15 WHERE Nom = "Girfaut"

# Supprimer des valeurs enregistrées

## Supprimer un enregistrement

DELETE FROM nom\_table WHERE conditions

### Exemples

- DELETE \* FROM Etudiants

ou

DELETE ALL FROM Etudiants

ou

TRUNCATE TABLE Etudiants

La différence entre **TRUNCATE** et **DELETE** se fait lors de la présence d'un auto-incrémentation dans la table. **TRUNCATE** réinitialise l'auto-incrémentation alors que **DELETE** non.

Suite ...

C'est tout ce qu'il y a besoin de savoir sur la manipulation des données ...

On va maintenant se focaliser sur :

# Les requêtes SQL

## Structure générale d'une requête

**SELECT** [ { **DISTINCT** | **UNIQUE** } | **ALL** ] liste\_colonnes 5  
**FROM** nomTable1 [, nomTable2 ...] 1  
[**WHERE** condition] 2  
[**GROUP BY** liste\_colonnes] 3  
[**HAVING** condition] 4  
[ { **UNION** | **MINUS** | **INTERSECT** } (autre\_requête) ] 6  
[**ORDER BY** liste\_colonnes] 7  
;

# Structure générale d'une requête

Considérons premièrement les trois clauses :

```
SELECT liste_attributs  
FROM liste_tables  
WHERE condition
```

- **SELECT** définit le format du résultat cherché
- **FROM** définit à partir de quelles tables le résultat est calculé
- **WHERE** définit les prédictats de sélection du résultat

# Sélection et projection des données

## Sélectionner l'ensemble des valeurs d'une table de données

```
SELECT * FROM Employe
```

On peut également créer une requête qui ne retourne qu'un sous-ensemble des attributs, i.e. faire une projection des données

```
SELECT A1, A2 FROM nom_table
```

La requête ne retourne que les attributs *A1* et *A2* de la table *nomtable*

# Sélection des lignes

## Exemples

SELECT \* FROM PAYS

NOM	CAPITALE	POPULATION	SURFACE
Irlande	Dublin	5	70
Autriche	Vienne	9	83
Angleterre	Londres	53	244
Japon	Tokyo	450	398
USA	Washington	314	441

SELECT Nom, Capitale FROM Pays

NOM	CAPITALE
Irlande	Dublin
Autriche	Vienne
Angleterre	Londres
Japon	Tokyo
USA	Washington

# Sélection des lignes

Sélectionner des lignes respectant une ou plusieurs condition(s) donnée(s)

```
SELECT * FROM Pays WHERE Surface > 100
```

NOM	CAPITALE	POPULATION	SURFACE
Angleterre	Londres	53	244
Japon	Tokyo	450	398
USA	Washington	314	441

On peut bien évidemment mettre plusieurs conditions afin de créer un filtre plus précis sur les données. On peut également combiner la *projection* avec la *sélection*.

# Opérateur de comparaison

## On peut comparer des nombres

- " $=$ " pour un test d'égalité
- " $<>$ " pour dire "différent de"
- " $>$ " pour dire "plus grand que"
- " $<$ " pour dire "plus petit que"
- "**BETWEEN v1 AND v2**" ou " $>v1 \text{ AND } <v2$ " pour dire "compris entre v1 et v2"
- "**NOT BETWEEN v1 AND v2**" ...

Mais on peut aussi faire des tests pour comparer des chaînes de caractère ou des bouts de chaines de caractères (**LIKE**) ou regarder si elle se trouve dans une liste (**IN**).

# Opérateur de comparaison

## Comparaison totale des chaînes de caractère

- ... WHERE Pays LIKE ...
- ... WHERE Pays IN Liste\_Pays

## Comparaison partielle des chaînes de caractère

- % : un ou plusieurs caractères  
... WHERE Pays LIKE '%lan%',
- un underscore pour remplacer exactement un caractère  
... WHERE Pays LIKE 'I\_lande'

Notez l'existence de **NOT LIKE** par opposition à **LIKE**

# Opérateur de comparaison

## Comparaison à un ensemble

On cherche par exemple tous les employés ayant un salaire plus grand que les employés administratifs

```
SELECT * FROM Employe
```

```
WHERE Salaire > ALL (SELECT Salaire From Employe  
                      WHERE Statut = "Administratif")
```

NOM	PRENOM	SALAIRE	STATUT
Bonnot	Jean	2100	Ingénieur
Smith	John	1800	Administratif
Afeu	Pierre	1500	Administratif
Lafleur	Marie	1700	Technicien
Rose	Sylvie	2500	DRH

NOM	PRENOM	SALAIRE	STATUT
Bonnot	Jean	2100	Ingénieur
Rose	Sylvie	2500	DRH

Donnez une requête équivalente qui fournirait la même table de résultats.

## Sélection des lignes

Il arrive dans certaines tables que des doublons soient présents (erreur humaine ou technique). Dans certaines études, il est parfois intéressant de savoir combien de clients différents fréquentent le magasin sur une période donnée. Cela peut se faire à l'aide de la commande **DISTINCT**

### Exemple :

On veut supprimer tous les doublons de personne (Nom, Prenom) se trouvant dans la table.

```
SELECT DISTINCT (Nom, Prenom) FROM Client
```

## Intersection entre deux tables

On peut être amené à étudier deux tables portant sur deux enseignes différentes avec des attributs identiques et on souhaite savoir si ces deux enseignes ont des clients en commun.

Pour cela on peut comparer leur fichier en réalisant une **intersection** entre les deux tables.

```
SELECT * FROM table_1  
INTERSECT  
SELECT * FROM table_2
```

Remarque : Les tuples en double sont éliminés du résultat (pas de copie dans la nouvelle table)

## Union et Différence

De la même façon, on peut réaliser une **union** entre deux tables, ce qui peut se révéler pratique lorsque notre table a été initialement fragmenté en plusieurs tables différentes.

```
SELECT * FROM table_1
UNION
SELECT * FROM table_2
```

On peut aussi **retrancher** des éléments existants à une table selon une autre table

```
SELECT * FROM table_1
MINUS
SELECT * FROM table_2
```

Remarque : Les tuples en double sont éliminés du résultat (pas de copie dans la nouvelle table)

# Jointure entre deux tables différentes

Dans le cadre d'un traitement, il est souvent nécessaire de croiser les éléments se trouvant sur deux tables ou plus, c'est ce que l'on appelle une **jointure**.

Considérons les deux tables suivantes :

NOM	CAPITALE	POPULATION	SURFACE
Irlande	Dublin	5	70
Autriche	Vienne	9	83
Angleterre	Londres	53	244
Japon	Tokyo	450	398
USA	Washington	314	441

Année	Lieu	Pays
1896	Athènes	Grèce
1900	Paris	France
1904	St Louis	USA
1908	Londres	UK

# Jointure entre deux tables différentes

La jointure se fait en précisant les deux tables utilisées pour la requête mais aussi **l'attribut commun** sur lequel on réalise la jointure.

## Exemple

```
SELECT Année, Lieu, Pays, Capitale  
FROM JO, Pays  
WHERE JO.Pays = Pays.Nom
```

Année	Lieu	Pays	Capitale
1908	Londres	UK	Londres
1904	St Louis	USA	Washington

## Jointure sur une même table

On se pose souvent ce genre de question lorsque que l'on souhaite comparer un même attribut de plusieurs lignes différentes.

**Exemple :** Comment comparer les populations des pays ?

```
SELECT P1.nom, P1.population, P2.nom, P2.population  
FROM Pays as P1, Pays AS P2  
WHERE P1.population > P2.population
```

Toutes les paires de pays telles que le premier pays a une population plus grande que le deuxième pays

Notez l'utilisation d'alias pour faire appel plusieurs fois à la table Pays.

## Jointure sur une même table

P1.NOM	P1.POPULATION	P2.NOM	P2.POPULATION
Autriche	9	Irlande	5
Autriche	9	Suisse	8
UK	53	Irlande	5
UK	53	Autriche	9
UK	53	Suisse	8
...	...	...	...

Quelle valeur d'attribut n'apparaîtra jamais dans P1.nom ? Idem dans P2.nom ?

## Tri des résultats

Il est possible de trier les résultats d'une requête selon un ou plusieurs attribut(s)

```
SELECT * FROM Pays ORDER BY POPULATION DESC
```

NOM	CAPITALE	POPULATION	SURFACE
Japon	Tokyo	450	398
USA	Washington	314	441
Angleterre	Londres	53	244
Autriche	Vienne	9	83
Irlande	Dublin	5	70

Autre option **ASC**.

On peut ordonner selon plusieurs attributs quand cela est nécessaire et surtout en cas de doublon avec le premier attribut mentionné.

## Variables calculées

On souhaite aussi calculer certaines grandeurs en fonction des attributs existants, *i.e.* calculer la valeur d'un nouvel attribut.

Par exemple, imaginons que l'on souhaite déterminer la densité de population des différents pays et l'afficher avec les autres colonnes. Tout cela avec seulement deux chiffres après la virgule

```
SELECT *, ROUND(Population/Surface,2) AS Densite  
FROM Pays
```

NOM	CAPITALE	POPULATION	SURFACE	Densité
Irlande	Dublin	5	70	0.07
Autriche	Vienne	9	83	0.11
Angleterre	Londres	53	244	0.22
Japon	Tokyo	450	398	1.13
USA	Washington	314	441	0.71

# Les fonctions d'agrégations

## Quelques fonctions d'agrégations

- **AVG** pour calculer la moyenne des valeurs d'un attribut
- **VARIANCE** : pour calculer la variance des valeurs d'un attribut
- **STDDEV** : pour calculer l'écart-type des valeurs d'un attribut
- **SUM** pour calculer la somme des valeurs d'un attribut
- **MIN** : valeur minimal d'un attribut
- **MAX** : valeur maximum d'un attribut
- **COUNT** : nombre de valeurs d'un attribut

Les quatre premiers sont spécifiques aux attributs numériques et les trois derniers sont globaux

# Fonctions d'agrégations

- **Exemple 1** : Quantité total en stock

```
SELECT SUM (Qte)
FROM Produits
```

- **Exemple 2** : Nombre clients

```
SELECT Count(Num_Client)
FROM Client
```

- **Exemple 3** : Le nombre de clients qui ont passé une commande

```
SELECT COUNT(DISTINCT Num_Client)
FROM Commandes
```

- **Exemple 4** : Salaire moyen dans une entreprise

```
SELECT AVG(salaire)
FROM Entreprises
```

# Les fonctions d'agrégations

On utilise souvent ces fonction d'agrégations pour calculer une moyenne ou un max, non pas sur l'ensemble des données, mais sur des groupes de données.

On effectue cela à l'aide de la clause **GROUP BY**.

- **Exemple 1** : Quantité en stock par catégorie

```
SELECT Catégorie, SUM(Qte)
FROM Produits
GROUP BY CATEGORIE
```

- **Exemple 2** : Le salaire moyen selon le poste occupé

```
SELECT Poste, AVG(salaire)
FROM Entreprises
GROUP BY Ville
```

## Les fonctions d'agrégations

On considère maintenant l'exemple suivant avec une liste de pays semblables aux exemples utilisés précédemment :

NOM	CAPITALE	POPULATION	SURFACE	CONTINENT
Irlande	Dublin	5	70	Europe
Autriche	Vienne	9	83	Europe
Angleterre	Londres	53	244	Europe
Mexique	Mexico	250	289	Amérique
USA	Washington	314	441	Amérique

Et regardons un exemple d'utilisation d'une fonction d'agrégation sur les continents.

Ecrire une requête qui détermine la plus faible, la plus grande et la valeur moyenne de population sur chaque continent ainsi que la somme des surfaces sur chaque continent et le nombre de pays sur chaque continent

# Les fonctions d'agrégations

```
SELECT Continent, MIN(Population), MAX(Population), AVG(Population),
       SUM(Surface), COUNT(*)  
FROM Pays  
GROUP BY Continent
```

CONTINENT	MIN(Population)	MAX(Population)	AVG(Population)	SUM(Surface)	COUNT(*)
Europe	5	53	22..3	397	3
Amérique	250	314	282	730	2

# Les fonctions d'agrégations

## Une autre fonction de groupement HAVING

Il s'agit d'une condition *a posteriori* sur un résultat de groupement contrairement à la clause **WHERE**.

La clause **HAVING** ne s'utilise qu'avec **GROUP BY**.

**Exemple :** on souhaite retourner uniquement les continents dont la surface excède 500.

```
SELECT Continent, SUM(Surface)
FROM Pays
GROUP BY Continent
HAVING SUM(Surface) > 500
```

CONTINENT	SUM(Surface)
Amérique	730

**Remarque :** nous aurions également pu utiliser un alias pour effectuer cette requête.

# Les requêtes imbriquées/emboîtées

On considère la base de données suivante :

- Produit(np, nomp, couleur, poids, prix) : base de produits
- Usine(nu, nomu, ville, pays) : base des usines
- Fournisseur (nf, nomf, type, ville, pays) : base des fournisseurs
- Livraison(np, nu, nf, quantité) : base des livraisons avec les références associées

## Les requêtes imbriquées/emboîtées

On souhaite déterminer le nom et la couleur des produits livrés par le fournisseur 1. On a deux solutions possibles.

# Les requêtes imbriquées/emboîtées

On souhaite déterminer le nom et la couleur des produits livrés par le fournisseur 1. On a deux solutions possibles.

**Solution 1 :** une jointure déclarative

```
SELECT nomp, couleur FROM Produit, Livraison  
WHERE (Livraison.np = Produit.np) AND nf = 1
```

# Les requêtes imbriquées/emboîtées

On souhaite déterminer le nom et la couleur des produits livrés par le fournisseur 1. On a deux solutions possibles.

**Solution 1 :** une jointure déclarative

```
SELECT nomp, couleur FROM Produit, Livraison  
WHERE (Livraison.np = Produit.np) AND nf = 1
```

**Solution 2 :** une jointure procédurale (emboîtement)

```
SELECT nomp, couleur FROM Produit  
WHERE np IN  
(SELECT np FROM Livraison WHERE nf = 1)
```

# Les requêtes imbriquées/emboîtées

## Retour sur la requête une jointure procédurale (emboîtement)

```
SELECT nomp, couleur FROM Produit  
WHERE np IN  
(SELECT np FROM Livraison WHERE nf =1)
```

- IN compare chaque valeur de *np* avec l'ensemble (ou multi-ensemble) de valeurs retournés par la sous-requête
- IN peut aussi comparer des tuples de valeurs

```
SELECT nu FROM Usine  
WHERE (ville, pays) IN  
(SELECT ville, pays, FROM Fournisseur)
```

# Les requêtes imbriquées/emboîtées

**Un autre exemple** une jointure procédurale (emboîtement)

```
SELECT nomf
FROM Produit, Usine, Fournisseur, Livraison
WHERE Produit.couleur = 'rouge'
      AND (Usine.ville = 'Londres' OR Usine.ville = 'Paris')
      AND Livraison.np = Produit.np
      AND Livraison.nf = Fournisseur.nf
      AND Produit.np = Livraison.np
```

Que fait la requête ci-dessus ?

# Les requêtes imbriquées/emboîtées

**Un autre exemple** une jointure procédurale (emboîtement)

```
SELECT nomf
FROM Produit, Usine, Fournisseur, Livraison
WHERE Produit.couleur = 'rouge'
      AND (Usine.ville = 'Londres' OR Usine.ville = 'Paris')
      AND Livraison.np = Produit.np
      AND Livraison.nf = Fournisseur.nf
      AND Produit.np = Livraison.np
```

Que fait la requête ci-dessus ?

Elle nous renseigne sur le nom des fournisseurs qui approvisionnent une usine de Londres ou de Paris en un produit rouge

# Les requêtes imbriquées/emboîtées

## Exercice

Ecrire une requête qui nous renseigne sur le nom des fournisseurs qui approvisionnent une usine de Londres ou de Paris en un produit rouge. La requête doit se faire sous forme de requêtes emboîtées !

```
SELECT nomf
FROM Fournisseur
WHERE nf IN
  (SELECT nf FROM Livraison
  WHERE np IN
    (SELECT np FROM Produit
    WHERE couleur = 'rouge')
  AND nu IN
    (SELECT nu FROM Usine
    WHERE ville = 'Londres' OR ville = 'Paris'))
```

# Les requêtes imbriquées/emboîtées

## Quantificateur ALL

On souhaite maintenant avoir le numéro des fournisseurs qui ne fournissent **que des produits rouges**

# Les requêtes imbriquées/emboîtées

## Quantificateur ALL

On souhaite maintenant avoir le numéro des fournisseurs qui ne fournissent **que des produits rouges**

```
SELECT nf FROM Fournisseur  
WHERE 'rouge' = ALL  
  (SELECT couleur FROM Produit  
   WHERE np IN  
     (SELECT np FROM Livraison  
      WHERE Livraison.nf = Fournisseur.nf))
```

- La requête imbriquée est ré-évaluée pour chaque tuple de la requête (ici pour chaque *nf*)
- tous les éléments de l'ensemble doivent vérifier la condition

# Les requêtes imbriquées/emboîtées

## Quantificateur EXISTS

On souhaite maintenant avoir le nom des fournisseurs qui fournissent au moins un produit rouge

# Les requêtes imbriquées/emboîtées

## Quantificateur EXISTS

On souhaite maintenant avoir le nom des fournisseurs qui fournissent au moins un produit rouge

```
SELECT nomf FROM Fournisseur  
WHERE EXISTS  
(SELECT *  
  FROM Livraison, Produit  
 WHERE Livraison.nf = Fournisseur.nf  
   AND Livraison.np = Produit.np  
   AND Produit.couleur = 'rouge')
```

- **EXISTS** test si l'ensemble n'est pas vide.

# Les jointures - JOIN

Nous avojs déjà vu comment faire des jointures entre des tables à l'aide de la clause **WHERE** à l'aide des clés primaires. C'est ce que l'on appelle une jointure **interne**, *i.e.* qui ne sélectionne que les données qui ont une correspondance entre les deux tables.

Il existe une autre façon de formuler une jointure **interne** à l'aide de la fonction clause **JOIN**.

## Les jointures - JOIN

Nous avojs déjà vu comment faire des jointures entre des tables à l'aide de la clause **WHERE** à l'aide des clés primaires. C'est ce que l'on appelle une jointure **interne**, *i.e.* qui ne sélectionne que les données qui ont une correspondance entre les deux tables.

Il existe une autre façon de formuler une jointure **interne** à l'aide de la fonction clause **JOIN**.

A l'image des jointures **internes**, il existe également des jointures **externes** qui, elles, sélectionnent toutes les données, même si certaines n'ont pas de correspondance dans l'autre table.

# Les jointures - JOIN

## Quelques clauses pour effectuer une jointure

- **INNER JOIN** : retourne les enregistrements lorsque la condition est vérifiée dans les deux tables
- **CROSS JOIN** : effectue le produit cartésien entre deux tables
- **LEFT JOIN** : retourne tous les éléments de la table de gauche même si la condition n'est pas vérifiée dans l'autre table
- **RIGHT JOIN** : même chose mais à droite
- **FULL JOIN** : retourne les résultats quand la condition est vraie dans au moins une des deux tables.
- **SELF JOIN** : permet d'effectuer la jointure d'une table avec elle-même
- **NATURAL JOIN** : jointure naturelle entre deux tables s'il y a au moins une colonne qui porte le même nom

On se focalisera uniquement sur les clauses **INNER JOIN**, **LEFT JOIN** and **RIGHT JOIN**.

# Les jointures - JOIN

On considère les deux tables suivantes : une table *Jeux* et une table *Joueurs* contenant différentes informations relatives à des jeux vidéos et aux propriétaires des jeux vidéos.

ID	Nom	ID_proprietaire	Console	Prix
1	Super Mario Bros	1	NES	4
2	Sonic	2	Megadrive	2
3	Zelda : Ocarina of time	1	Nintendo 64	15
4	Mario Kart 64	1	Nintendo 64	25
5	Super Smash Bros Melee	3	GameCube	55
6	Bomberman	6	NES	10

ID	Prenom	Nom	Téléphone
1	Florent	Dugommier	01 44 77 21 33
2	Patrick	Lejeune	03 22 17 41 22
3	Michel	Doussand	04 11 78 02 00
4	Romain	Vipelli	01 21 98 51 01

## Les jointures Internes

Partant des deux tables précédentes, je souhaite associer retrouver le nom du propriétaire de chaque jeu.

Il s'agit ici d'une jointure **interne** que l'on peut effectuer avec la clause **WHERE** :

## Les jointures Internes

Partant des deux tables précédentes, je souhaite associer retrouver le nom du propriétaire de chaque jeu.

Il s'agit ici d'une jointure **interne** que l'on peut effectuer avec la clause **WHERE** :

```
SELECT j.nom AS nom_jeu, p.prenom as Nom_proprietaire  
FROM Jeux as j, Joueurs as p  
WHERE j.ID_proprietaire = p.ID
```

## Les jointures Internes

Partant des deux tables précédentes, je souhaite associer retrouver le nom du propriétaire de chaque jeu.

Il s'agit ici d'une jointure **interne** que l'on peut effectuer avec la clause **WHERE** :

```
SELECT j.nom AS nom_jeu, p.prenom as Nom_proprietaire  
FROM Jeux as j, Joueurs as p  
WHERE j.ID_proprietaire = p.ID
```

Mais on peut aussi le faire à l'aide de la clause **INNER JOIN**

## Les jointures Internes

Partant des deux tables précédentes, je souhaite associer retrouver le nom du propriétaire de chaque jeu.

Il s'agit ici d'une jointure **interne** que l'on peut effectuer avec la clause **WHERE** :

```
SELECT j.nom AS nom_jeu, p.prenom as Nom_proprietaire  
FROM Jeux as j, Joueurs as p  
WHERE j.ID_proprietaire = p.ID
```

Mais on peut aussi le faire à l'aide de la clause **INNER JOIN**

```
SELECT j.nom AS nom_jeu, p.prenom as Nom_proprietaire  
FROM Joueurs as p  
INNER JOIN jeux as j  
ON j.ID_proprietaire = p.ID
```

# Les jointures internes

Nom_jeu	Nom_proprietaire
Super Mario Bros	Florent
Sonic	Patrick
Zelda : Ocarina of time	Florent
Mario Kart 64	Florent
Super Smash Bros Melee	Michel

## Les jointures externes

Ces jointures permettent de récupérer toutes les données, même celles qui n'ont pas de correspondance.

**Exemple :** une jointure externe à gauche, on remplace **INNER** par **LEFT**, ce qui nous donne

```
SELECT j.nom AS nom_jeu, p.prenom as Nom_proprietaire  
FROM Joueurs as p  
LEFT JOIN jeux as j  
ON j.ID_proprietaire = p.ID
```

## Les jointures externes

Ces jointures permettent de récupérer toutes les données, même celles qui n'ont pas de correspondance.

**Exemple :** une jointure externe à gauche, on remplace **INNER** par **LEFT**, ce qui nous donne

```
SELECT j.nom AS nom_jeu, p.prenom as Nom_proprietaire  
FROM Joueurs as p  
LEFT JOIN jeux as j  
ON j.ID_proprietaire = p.ID
```

Nom_jeu	Nom_Proprietaire
Super Mario Bros	Florent
Zelda : Ocarina of Time	Florent
Mario Kart 64	Florent
Sonic	Patrick
Super Smash Bros	Michel
NULL	Romain

# Les jointures externes

## Remarques :

Dans ce premier exemple la table *Joueurs* sert de table de référence et toute les lignes de joueurs apparaissent bien dans la table de résultat (même pour les joueurs qui ne possèdent pas de jeux vidéos !!!)

Dans ce cas, un joueur qui ne possède pas jeu vidéo se verra associer la valeur **NULL**

On associe ensuite à chaque joueur les jeux vidéos qu'il possède.

En fine, un joueur apparaîtra autant de fois dans la table qu'il possède de jeux vidéos.

# Les jointures externes

## Exemple :

Une jointure externe à droite, on remplace **INNER** par **RIGHT**, ce qui nous donne

```
SELECT j.nom AS nom_jeu, p.prenom as Nom_proprietaire  
FROM Joueurs as p  
RIGHT JOIN jeux as j  
ON j.ID_proprietaire = p.ID
```

# Les jointures externes

## Exemple :

Une jointure externe à droite, on remplace **INNER** par **RIGHT**, ce qui nous donne

```
SELECT j.nom AS nom_jeu, p.prenom as Nom_proprietaire  
FROM Joueurs as p  
RIGHT JOIN jeux as j  
ON j.ID_proprietaire = p.ID
```

Nom	ID_proprietaire
Super Mario Bros	Florent
Sonic	Patrick
Zelda : Ocarina of time	Florent
Mario Kart 64	Florent
Super Smash Bros Melee	Michel
Bomberman	NULL

# Création de Vues

Une vue est une table *virtuelle* calculée à partir d'autres tables grâce à une requête.

```
CREATE VIEW nom_vue AS requête
```

Exemple :

```
CREATE VIEW lesfact AS SELECT NuFact, Cocli FROM Facture;
```

La modification d'une table de la requête de la vue affecte la vue.

Pas de ORDER BY dans une vue.

Si la table Facture est remaniée, la vue lesfact doit être refaite, mais les requêtes qui utilisent cette vue n'ont pas à être remaniées.

# Raisons d'utilisation

- Masquer la complexité d'une requête
- Confidentialité
- Requêtes fréquemment utilisées
- Indépendance logique

## Création d'index

Un index est une structure de données physique permettant d'accélérer les accès aux données.

Objectif : Optimiser les requêtes: sélection, jointure

```
CREATE INDEX idx_NomClient ON client (nom);
```

Suppression :

```
DROP INDEX nom_index;
```

Index implicite: la clé primaire d'une table est automatiquement indexée.

# LCD: Contrôle d'accès aux données

- Les BD comptent souvent plusieurs utilisateurs, notamment lorsqu'elles sont partagées en réseau et tous, n'ont pas nécessairement des besoins identiques.
- Tous les utilisateurs ne sont donc pas habilités à consulter et modifier toutes les données d'une base. SQL offre des fonctions de règlementation des droits d'accès aux données sous la forme de priviléges.
- Seul l'administrateur qui a créé un élément (une table ou une vue) a la possibilité d'accorder ou de retirer des droits sur cet élément.  
Ex : accorder le droit de visionner le contenu d'une table, de modifier une table partiellement ou totalement, supprimer une table, ...

# Contrôle d'accès aux données : Principe

Un privilège est l'autorisation qui est accordée à un utilisateur d'effectuer une opération sur un objet. Un privilège concerne donc, et **doit spécifier**, une opération, un objet (ressource) de la base de données ou de son environnement, l'utilisateur qui accorde le privilège (c'est celui qui exécute la requête de création ou retrait du privilège) et celui qui le reçoit.

## Privilèges sur une table

Les principales opérations admises sur le contenu des tables sont les suivantes :

- **SELECT** : extraction des données
- **INSERT** : ajout de lignes
- **DELETE** : suppression de lignes ou tables
- **UPDATE** : modification des valeurs de certaines colonnes

La commande ci-dessous permet aux utilisateurs nommés de consulter le contenu d'une table *Produit* et de modifier les valeurs de *Qstock* et *Prix* (ex : *des droits accordés aux personnels en gestion des stocks*)

```
GRANT select, update(Qstock, Prix)
ON Produit
TO P_Mercier, S_Financiers
```

## Privilèges sur une table

Il est également possible de transmettre au destinataire d'un privilège, le privilège de transmettre celui-ci à d'autres utilisateurs. La commande ci-dessous permet aux utilisateurs aux utilisateurs nommés d'effectuer n'importe quelle manipulation sur la table *Produit* mais aussi de transmettre ce privilège s'il juge cela nécessaire

```
GRANT ALL  
ON Produit  
TO P_Mercier, S_Financiers  
WITH grant option
```

Un privilège peut aussi être accorder à l'ensemble des utilisateurs

```
GRANT select  
ON Produit  
TO public
```

## Privilèges sur une table

Il existe d'autres privilèges liés notamment à l'administration de la base de données, en particulier à la définition et à la modification de structure de données : **CREATE TABLE**, **ALTER TABLE**, **DROP INDEX** ou encore **DELETE**.

Tout comme il est possible d'accorder des droits/privilèges à des utilisateurs, il est également possible de leur **retirer des droits** : ce qui est plutôt pratique lorsque certains utilisateurs, extérieurs à une entreprise, ont fini de travailler sur un projet et qu'ils n'ont donc plus besoin d'avoir accès à une base de données.

# Privilèges sur une table : REVOKE

La commande **REVOKE** permet de retirer un privilège préalablement accorder à un utilisateur

```
REVOKE update(Prix)  
ON Produit  
TO P_Mercier
```

```
REVOKE run  
ON Compta  
FROM P_Mercier
```

## Remarques :

- lorsqu'un même privilège fut accordé à un même utilisateur par plusieurs personnes indépendantes, alors cet utilisateur disposera de ce privilège tant que chacune des personnes ne le lui aura pas retiré.
- il est aussi possible de retirer la faculté de transmettre un privilège par une commande telle que :

```
REVOKE grant option for update (Compte)  
ON Client  
FROM P_Mercier
```

## Exemples :

Ecrire les commandes permettant d'effectuer les demandes suivantes :

- 1) Le droit de d'ajouter et de consulter des enregistrements de la table Clients est accorder à tous les utilisateurs
- 2) Pierre obtient tous les privilèges sur la table Client et a le droit d'accorder des privilèges à d'autres utilisateurs
- 3) Tous les utilisateurs ont le droit de consulter toutes les tables
- 4) Marine a le droit d'ajouter des lignes à la table Produit
- 5) Marine a le droit de modifier le contenu de la table Catalogue et peut déléguer ce droit
- 6) Thomas et Pierre peuvent mettre à jour les prix proposés par les fournisseurs

## Exemples :

1) GRANT SELECT, INSERT

ON Clients

TO PUBLIC

2) GRANT ALL

ON Client

TO Pierre

WITH Grant Option

3) GRANT SELECT

ON ALL

TO PUBLIC

# Exemples :

4) GRANT INSERT

ON Produit

TO Marine

5) GRANT UPDATE

ON Catalogue

TO Marine

WITH GRANT OPTION

6) GRANT UPDATE(Prix)

ON Fournisseurs

TO Thomas, Pierre

## Exemples :

Il nous reste encore à regarder comment créer des utilisateurs.  
Cela se fait à l'aide de la commande **CREATE USER**.

### Création d'utilisateurs

```
CREATE USER Nom_User WITH PASSWORD = '*****', MUST_CHANGE
```

### Exemple :

```
CREATE USER Guillaume WITH PASSWORD = '1234', MUST_CHANGE
```

### Supprimer des utilisateurs

```
DROP USER Nom_User
```