Bases de données

Cours I - Rappels

Guénaël Cabanes
cabanes@lipn.univ-paris13.fr

Partie I

-

Introduction

De l'information aux données

Toute connaissance peut être numérisé :

- Valeurs numériques
- Textes
- Images/vidéos
- Son/Musique
- Signaux

Une donnée est une information numérisée

Les données

- Toute information doit être stockée sous forme de données pour pouvoir être manipulé
- Un ensemble de données est un modèle numérique d'un phénomène du monde réel

Les données décrivent/modélisent des objets réels

Problème de volume

Une base de données peut contenir des milliards de lignes de données reparties dans des dizaines de millier de tables.

Comment garantir l'accessibilité de ces données, leur protection contre les incidents, des temps d'accès satisfaisants ?

Problème de maintenance

Les données sont en constante évolution:

- Nouvelles données
- Changement de la description des données
- Données obsoletes

Comment préserver les informations et les programmes utilisateurs ?

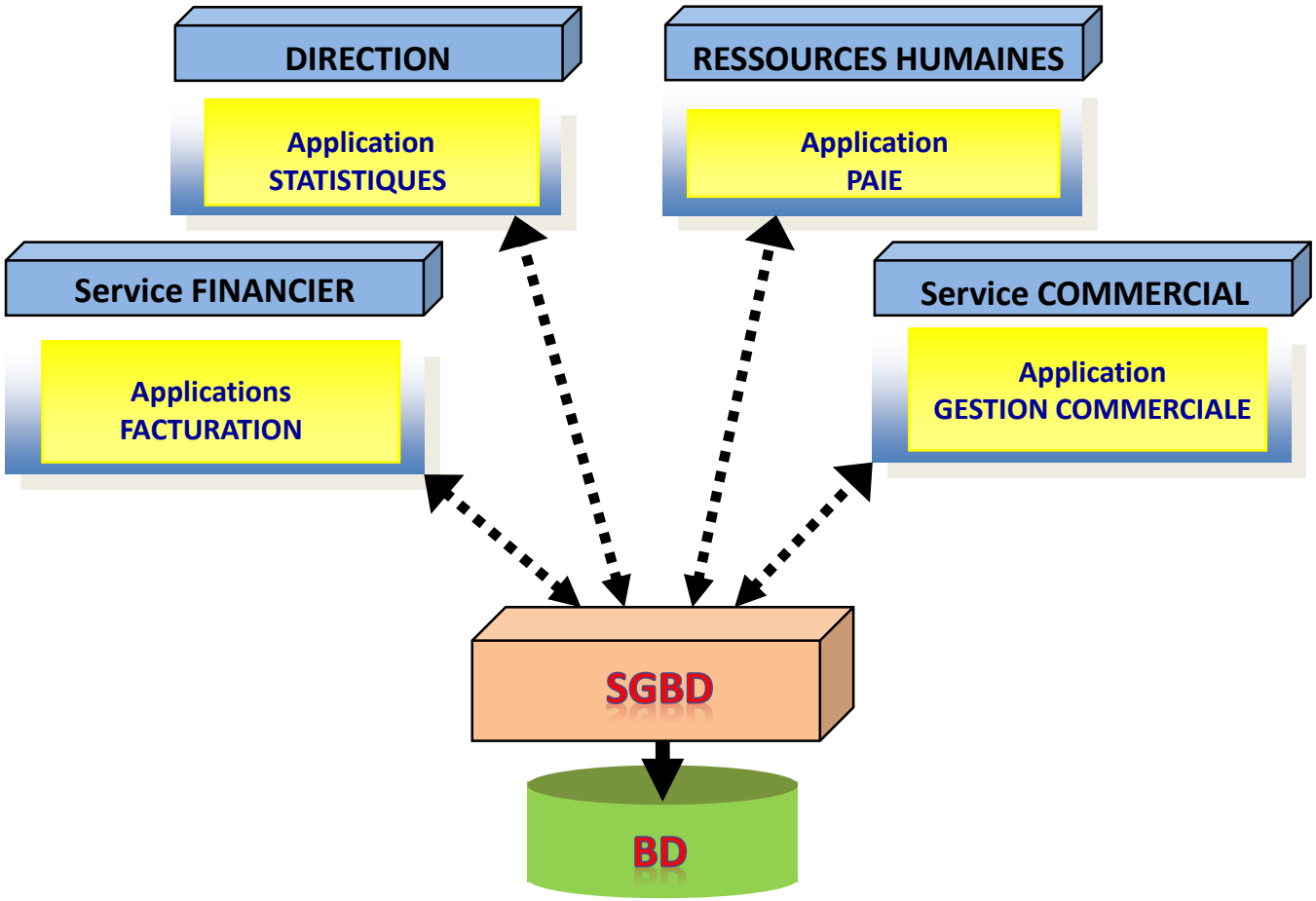
Problème de structure

Les données sont souvent distribuées:

- Sur plusieurs sites
- Entre plusieurs utilisateurs

Comment garantir la cohérence, la protection et l'accessibilité des données?

Structure d'une BD



Données sur les
EMPLOYES, CLIENTS, FOURNISSEURS, FACTURES

Avantages

- Intègre toutes les données pertinentes à un système dans une même structure que l'on nomme une Base de Données
- Assure la gestion de ces données par un logiciel dit *Système de Gestion de Bases de Données* (SGBD)
- Offre une vision partielle (vue) de l'ensemble des données à chaque utilisateur.
- Assure le partage des données entre les différents utilisateurs.

Rôles d'un SGBD

- **Intégrer toutes les données** pertinentes à un système dans une même structure (la Base de Données)
- **Offrir une vision (vue) partielle de l'ensemble des données** à chaque utilisateur. Cette vision correspond aux besoins de chacun
- **Assurer le partage des données** entre les différents utilisateurs

Trois niveaux d'abstraction

Les SGBD reposent sur trois niveaux d'abstraction qui :

- Assurent l'indépendance logique et physique des données
- Autorisent la manipulation de données
- Garantissent l'intégrité des données
- Optimisent l'accès aux données.

La description de ces trois niveaux et la correspondance entre eux est faite à travers le dictionnaire de données.

Le niveau externe

Ce niveau détermine le schéma externe qui contient les vues des utilisateurs sur la base de données c'est-à-dire le sous-ensemble de données accessibles ainsi que certains assemblages d'information et éventuellement des informations calculées.

Il peut donc exister plusieurs schémas externes représentant différentes vues sur la base de données avec des possibilités de recouvrement.

Le niveau conceptuel

Il correspond à la vision générale des données, indépendante des applications individuelles et de la façon dont les données sont stockées.

Cette représentation est en adéquation avec le modèle de données utilisé.

Le niveau conceptuel est défini au travers du schéma conceptuel.

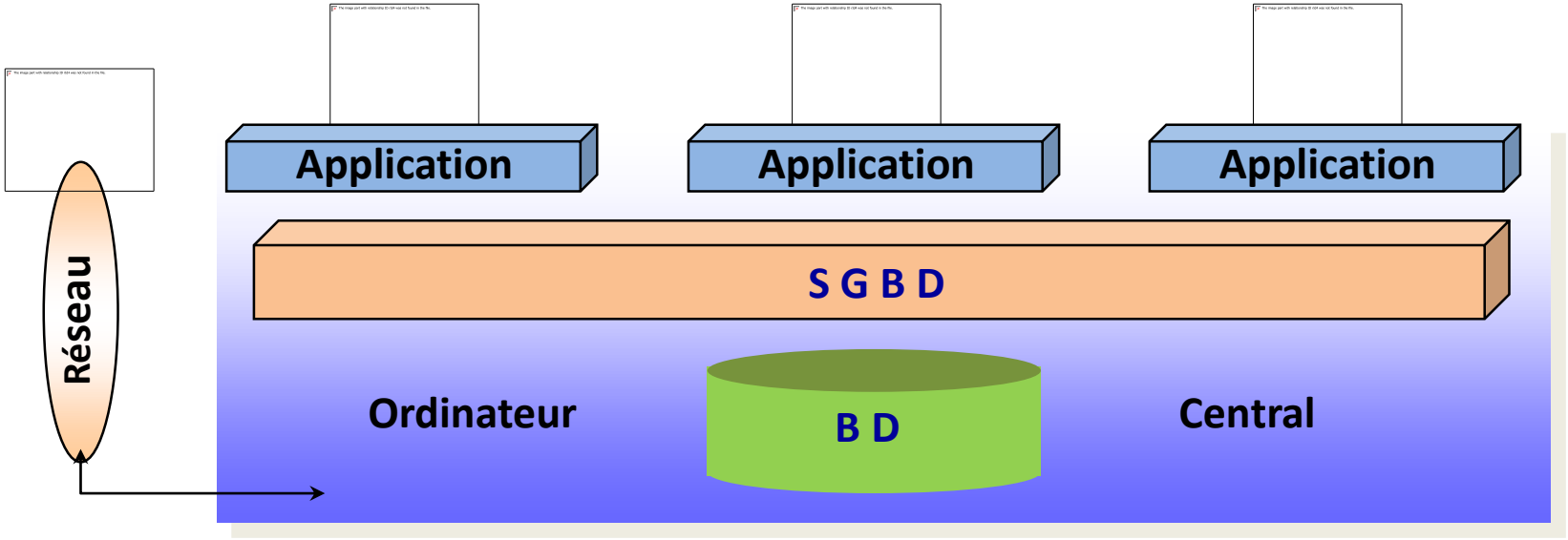
Le niveau physique

Il regroupe les services de gestion de la mémoire secondaire.

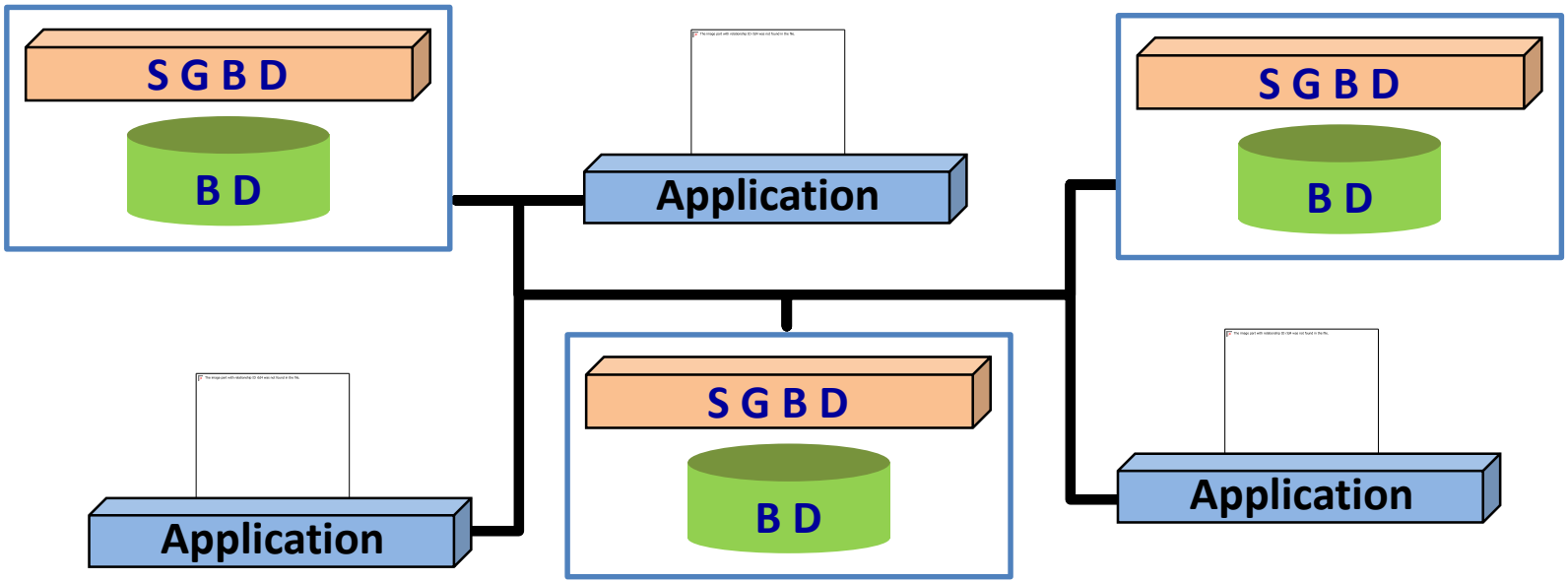
Il s'appuie sur un système de gestion de fichiers pour définir la politique de stockage ainsi que le placement des données.

Cette politique est définie en fonction des volumes de données traitées et en fonction de l'environnement matériel disponible.

Architecture centralisée



Architecture distribuée



Partie II

-

Conception d’une BD

Les modèles

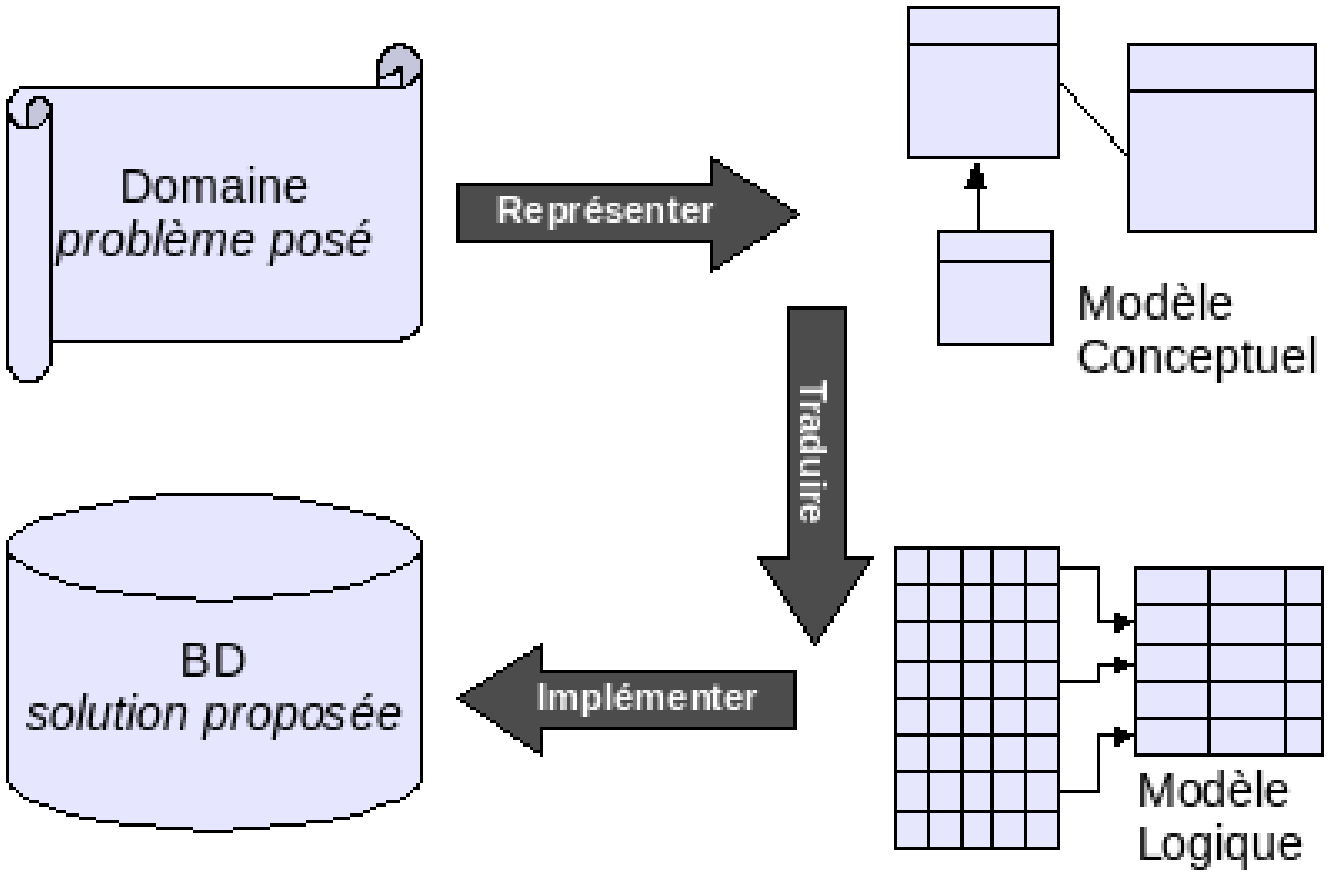
Il existe plusieurs modèles de bases de données différents :

- les modèles hiérarchique et réseau (1960-80)
- le modèle relationnel (1980 et +)
- le modèle objet (2000 et +)

Le modèle relationnel a prouvé sa forte popularité :

- simplicité de la représentation de données
- puissance de ses opérateurs de manipulation de données.

Méthode de conception

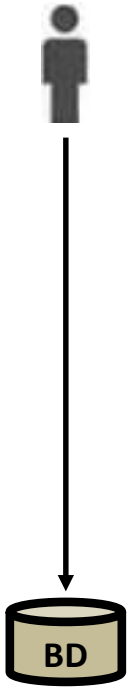


Méthode de conception

1. **Conception & Modélisation de BD (formalisme)**
   ➔ Modèle Entité-Association E/A

2. **Implantation de BD**
   ➔ Modèle Relationnel

3. **Programmation dans le SGBD**
   ➔ Langages (SQL, PL/SQL, Java…)

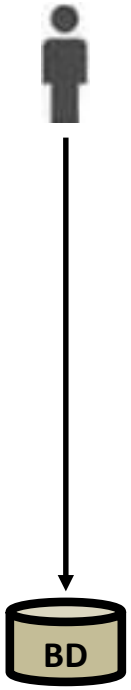


Méthode de conception

1. **Étape conceptuelle**
   ➔ Modèle Entité-Association E/A

2. **Étape logique**
   ➔ Modèle Relationnel

3. **Étape physique**
   ➔ Langages (SQL, PL/SQL, Java…)



Le modèle Entité-Association

Le modèle **Entité-Association (EA)** a été introduit en 1976.

Depuis, il a fait l'objet de plusieurs extensions (1990 +).

La version de base de ce modèle constitue un excellent outil de communication entre les concepteurs de systèmes d'information et les utilisateurs.

Le modèle Entité-Association

Le modèle **Entité-Association (E/A)** s'appelle aussi le modèle **Entité/Relation**.

En anglais il s'appelle : « **Entity Relationship Model (ER)** ».

Depuis, il a fait l'objet de plusieurs extensions :

le modèle entité/association étendu

= **Extended Entity Relationship Model (EER).**

Les attributs

Un **attribut** (ou une **propriété**) est une information élémentaire définie sur un domaine (Entier, Réel, Caractères, Date, Temps) décrivant une entité ou une association.

Exemples :

NomClient	:	Chaine de caractères(30)/Texte(30)
NuméroProduit	:	Entier
PrixUnitaireProduit	:	Réel (10dont2déc)
DateContactClient	:	Date (jj/mmm/aaaa)

Les entités

Une **entité** (ou un **individu**) est:

- un objet réel a modéliser
- décrit par un ensemble d'attributs
- indépendamment des autres entités

Exemples :

Une personne définie par son numéro de sécurité sociale, son nom, son prénom et sa date de naissance :

158097520072013	**CLEMENCE**	**Adam**	**17/SEP/1958**
269117520072013	**CLEMENCE**	**Eve**	**22/NOV/1969**

Une commande définie par son numéro et sa date :

FB01	**16/OCT/1996**
IB02	**19/JUIN/2001**

Les types d'entité

Un **type d'entité** (ou une **classe d'entités**) est un ensemble d'entités de même nature (ayant notamment les mêmes attributs). Chaque entité est identifiée de façon unique.

Par convention, un type d'entité est représenté par un rectangle divisé en deux parties :

- la partie supérieure contient le nom du type d'entité, et
- la partie du dessous contient l'ensemble de(s) attribut(s) qui décrivent le type.

Exemples :

PERSONNE
PersNcq
PersNom
PersPrénom …
PersNaiss

COMMANDE
ComNum
ComDate

VEHICULE
VNimmat
VMarque
VModèle …
VDatCircul

Nom du Type d'Entité
Attribut-1
Attribut-2
Attribut-3 …
Attribut-n

Les identifiants d'entité

Un **identifiant d'entité** est une liste minimale d'un ou plusieurs attributs permettant de distinguer de manière unique chaque entité

Par convention, l'identifiant d'entité est souligné.

Exemples : PersNcq ; ComNum ; VNimmat

PERSONNE
PersNcq
PersNom
PersPrénom …
PersNaiss

COMMANDE
ComNum
ComDate

VEHICULE
VNimmat
VMarque
VModèle …
VDatCircul

Nom du Type d'Entité
Attribut(s) → Identifiant
Attribut-a
Attribut-b …
Attribut-z

Les associations

Une **association** (ou **une relation)** est un lien sémantique défini entre deux ou plusieurs entités, identifié par rapport aux entités participantes, éventuellement décrit par un ensemble d'attributs.

Exemples : L'association **Acheter/Posséder** est identifiée par le numéro de sécurité sociale de la personne et le numéro d'immatriculation de sa voiture. Les **occurrences** ci-dessous donnent la liste des personnes qui ont acheté des véhicules et les différentes dates d'acquisitions.

158097520072013	41 HCC 75	10/10/1988
158097520072013	324 JZD 75	20/05/1991
158097520072013	768 KPP 75	17/11/1993
269117520072013	777 RA 94	08/07/2003
← Identifiant (2 attributs) →		← Attribut →

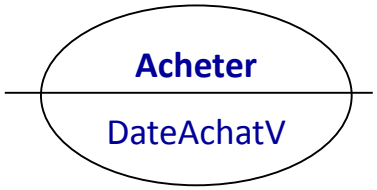
Les types d'association

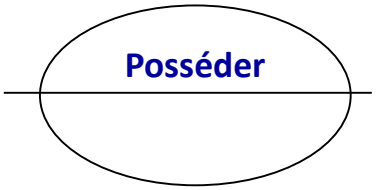
Un **type d'association** (ou une **structure d'association**) est une abstraction représentant une famille d'associations, décrite par un ensemble d'attributs, identifiant de façon unique chacune des associations par les identifiants des entités participantes.

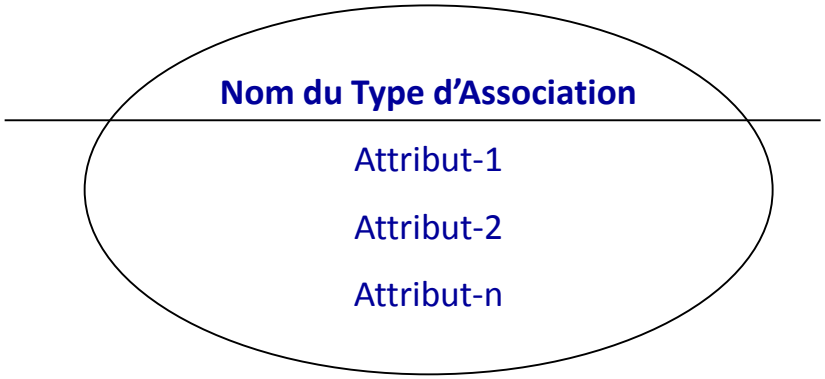
Par convention, un type d'association est représenté par un ovale divisé en deux parties :

- la partie supérieure contient le nom du type d'association, et
- la partie inférieure contient éventuellement la liste des attributs qui décrivent le type (autres que les identifiants des entités participantes).

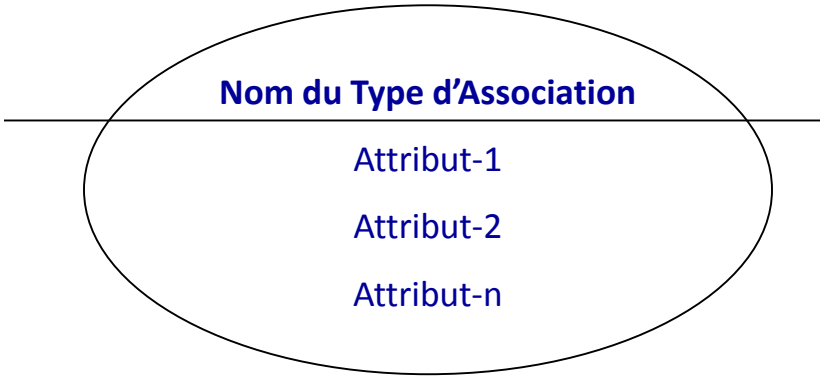
Exemples :



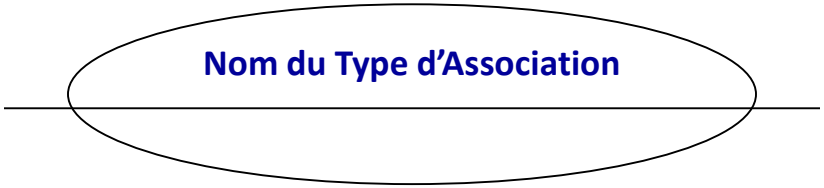




Les types d'association



Association porteuse de données



Association non porteuse de données ;

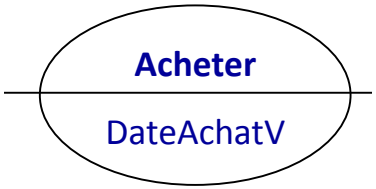
Elle est sans attribut

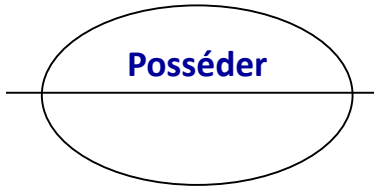
Dimension d'un type d'association

La **dimension d'un type d'association** est le nombre d'entités qui interviennent dans chaque occurrence de l'association.

La dimension correspond au nombre de "pattes" (les arcs entre les types d'entité et les types d'association) qui figurent sur la représentation graphique du type de l'association.

Exemples :

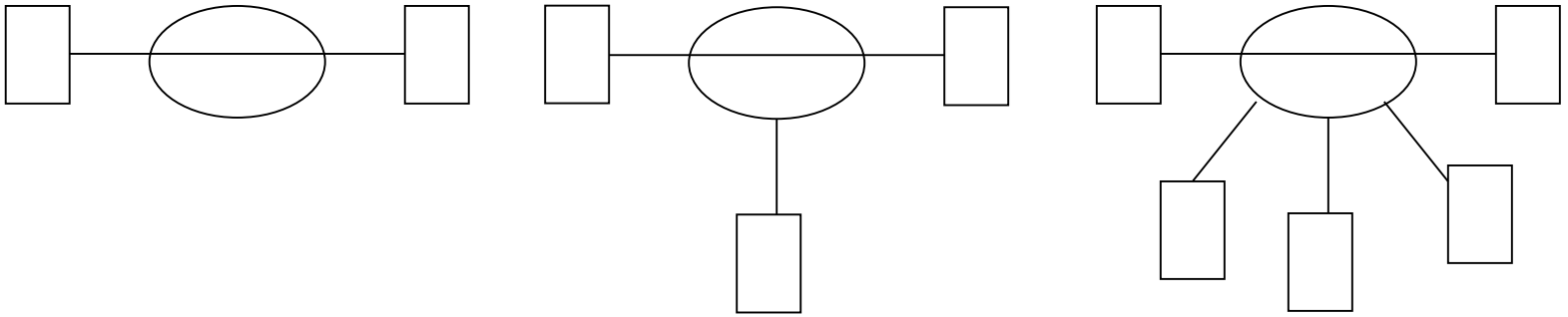




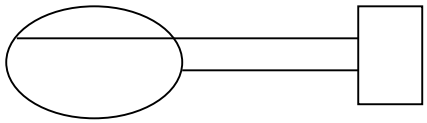
La dimension de l'association **Acheter/Posséder** est égale à 2, car elle relie PERSONNE à VEHICULE

Dimension d’un type d’association

Un type d'association de dimension 2 est dit **binaire.** Il est dit **ternaire** si la dimension est de 3 et plus généralement **n-aire** pour la dimension n (n>3).

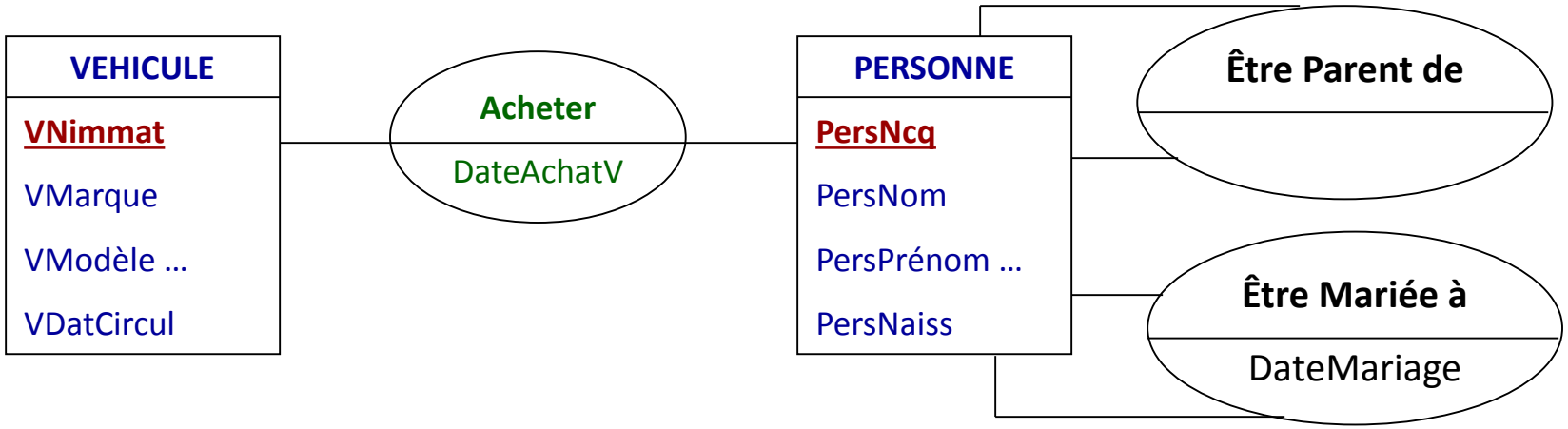


Un type d'association est dit **réflexif (récursif)** si elle relie plusieurs occurrences d’un même type d'entité.



Dimension d'un type d'association

Exemples :

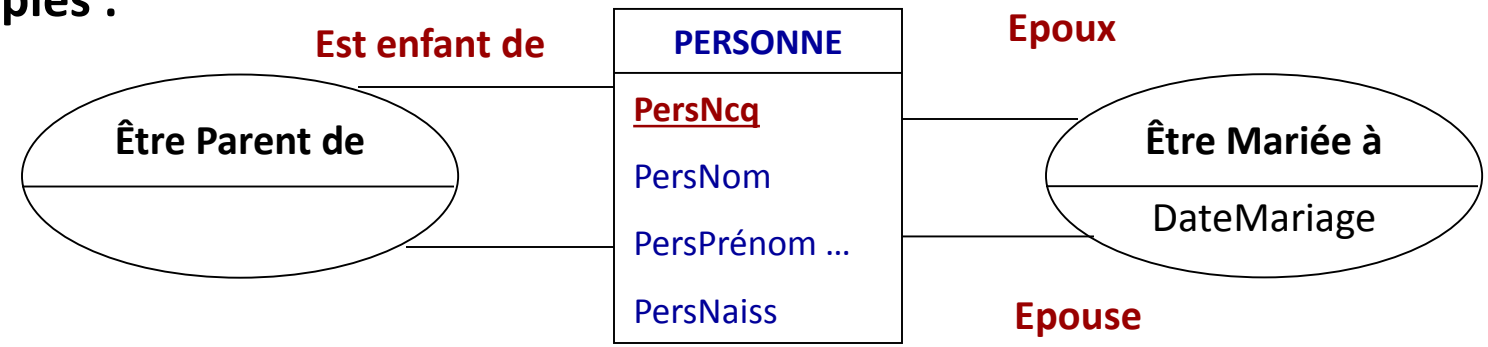


Rôles

La notion de **rôle** sert à distinguer deux occurrences d'un même type d'entité participant à une occurrence du type d'association réflexif.

Il est obligatoire sur l'un des deux arcs d'un lien réflexif. Les rôles doivent être différents dans les types réflexifs.

Exemples :



Cardinalité d'une association

Les **cardinalités** indiquent, pour chaque couple (type d'entité, type d'association), les nombres minimaux et maximaux d'occurrences de l'association pouvant exister pour une occurrence de l'entité.

C'est un couple de valeurs MIN, MAX spécifiant :

- si la relation est partielle (min=0) ou totale (min=1)
- si elle représente une fonction monovaluée (MAX=1) ou multivaluée (MAX>1).

Cardinalité d'une association

Les quatre cardinalités les plus connues sont :

- 0,1 (au plus un)
- 1,1 (exactement un)
- 0,n (zéro ou plus)
- 1,n (au moins un)

Mais il existe d'autres cardinalités :

- 0,b (au plus b)
- a,n (au moins a)
- a,a (exactement a)
- a,b (au moins a et au plus b, a<b)
- $\{a_1, a_2, \ldots, a_p\}$ (soit a_1 fois, soit a_2 fois, … soit a_p fois)

Cardinalité d'une association

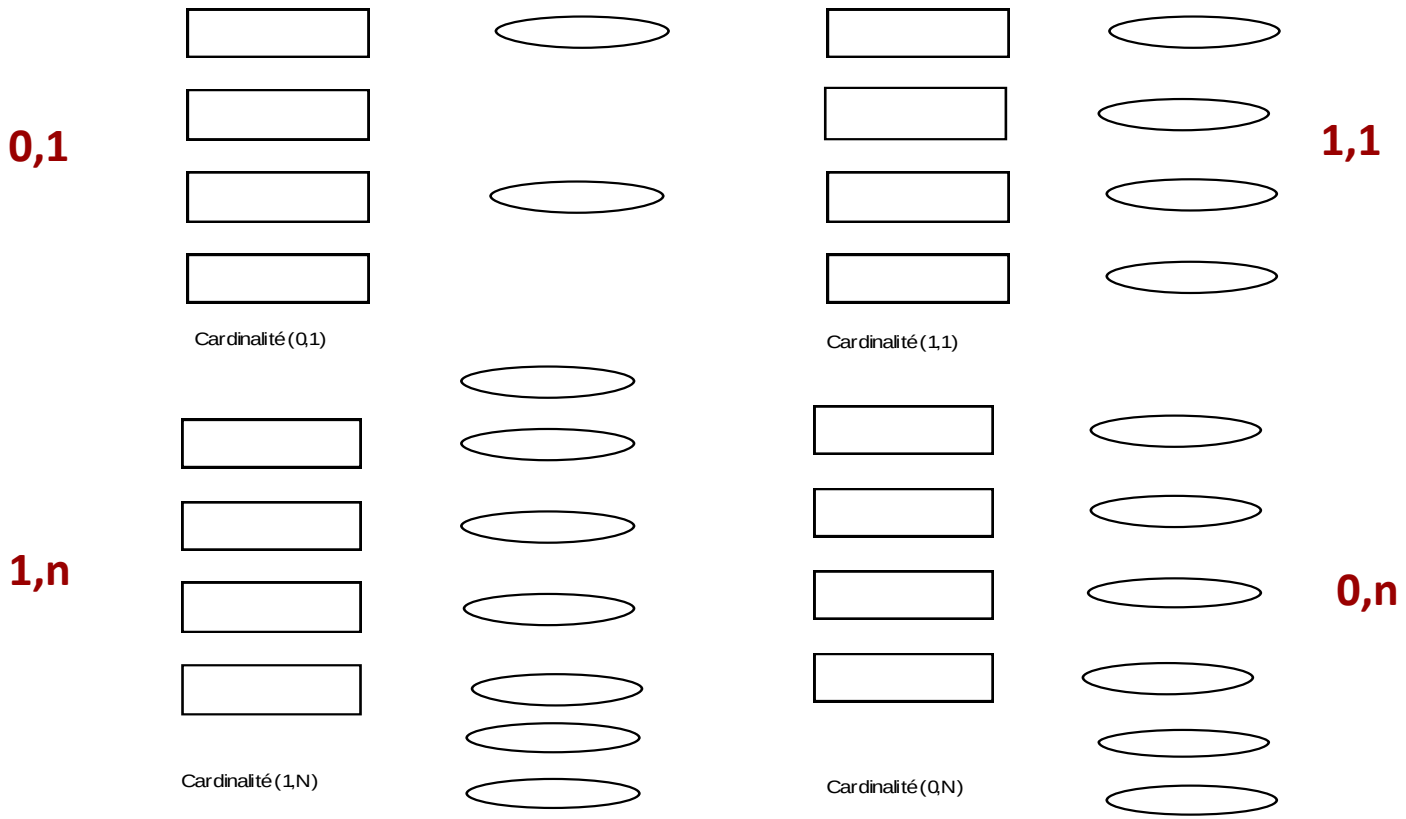
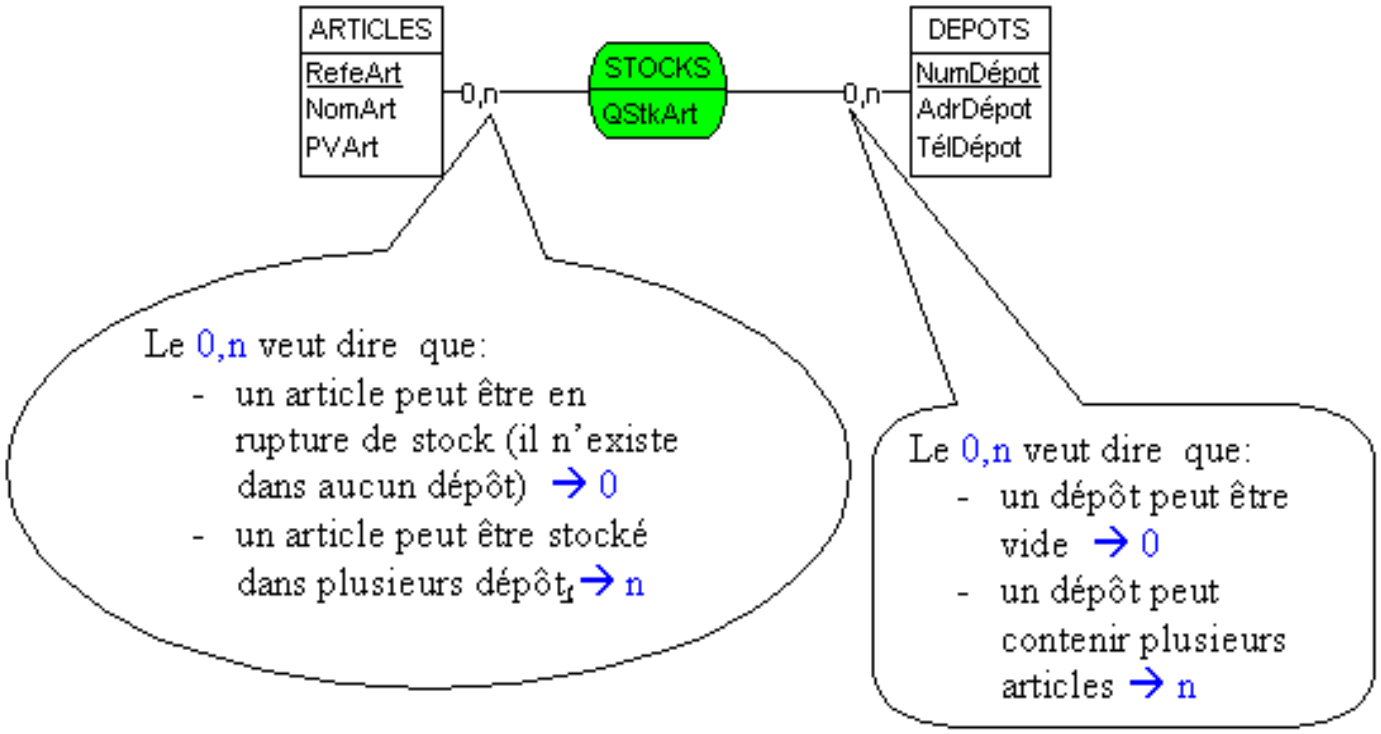


Diagramme Entité/Association

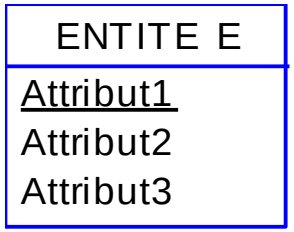
Un **diagramme Entité/Association (DEA)** est une représentation graphique, d'un domaine donnée du réel, qui permet de structurer les informations sur les objets du monde considéré.

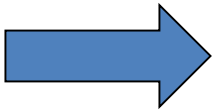


Modèle relationnel

Il existe 3 règles pour passer du modèle Entité-Association au modèle rationnel.

1. Tout type d'entité sera traduit en une table (relation)

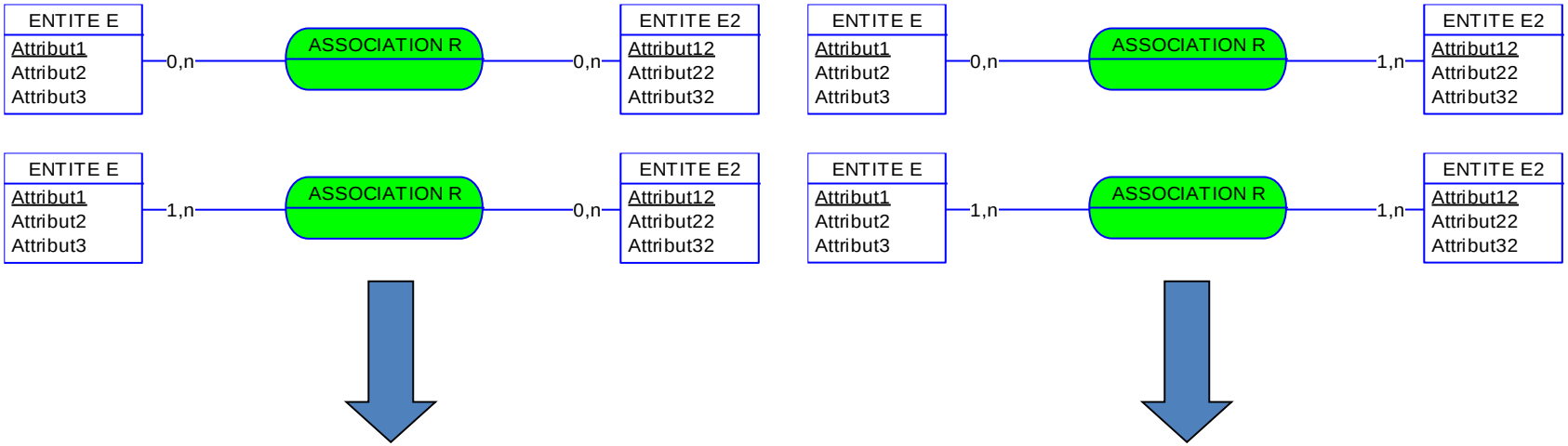




Table/Relation E (Attribut1, Attribut2, Attribut3)

Règles de passage

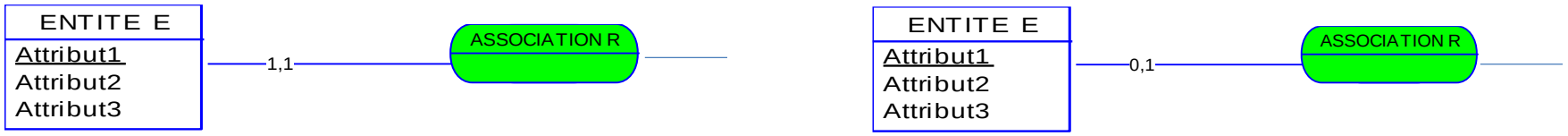
2. Tout type d'association sera traduit en une relation (table) si toutes cardinalités sont du type 0,n ou 1,n



Table/Relation R (Attribut1↑, Attribut12 ↑, …)

Règles de passage

3. **Tout type d'association sera traduit en une colonne (attribut) supplémentaire dans une table existante s'il y a une cardinalité du type 0,1 ou 1,1**



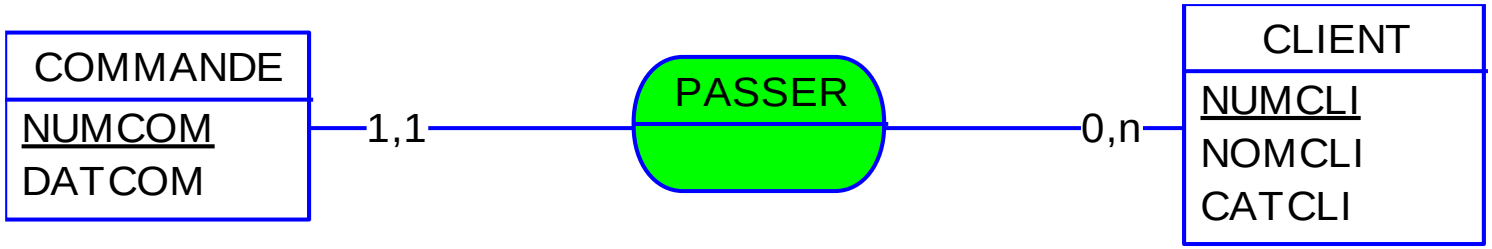
Table/Relation E (Attribut1, Attribut2, Attribut3, CléEtrangère↑)

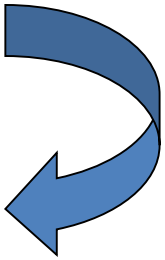
Règles de passage

- **Tout type d'entité E est traduit en une relation R**
  - La clé primaire de R est l'identifiant de E
  - Les attributs de R sont ceux de E

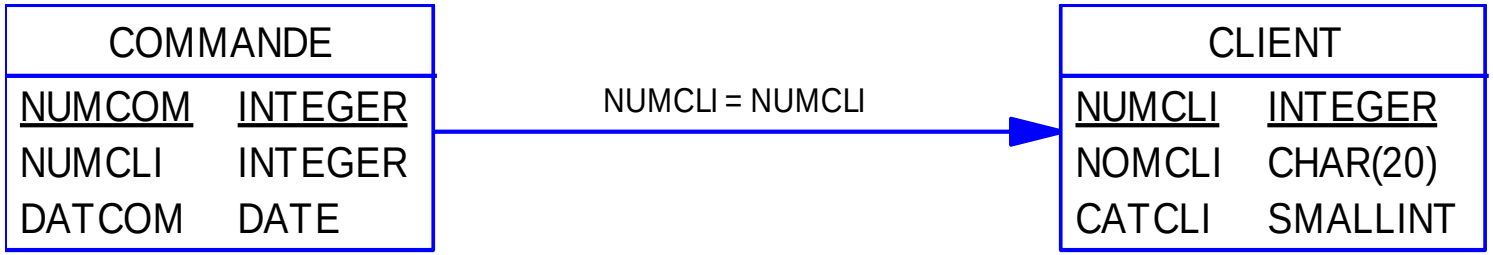
- **Tout type d'association est traduit :**
  - En une clé étrangère si la cardinalité est 1,1 ou 0,1
  - En une nouvelle relation sinon

Exemple 1

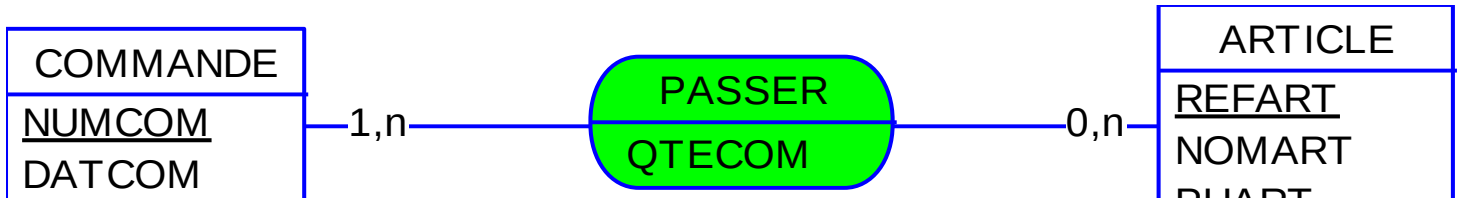




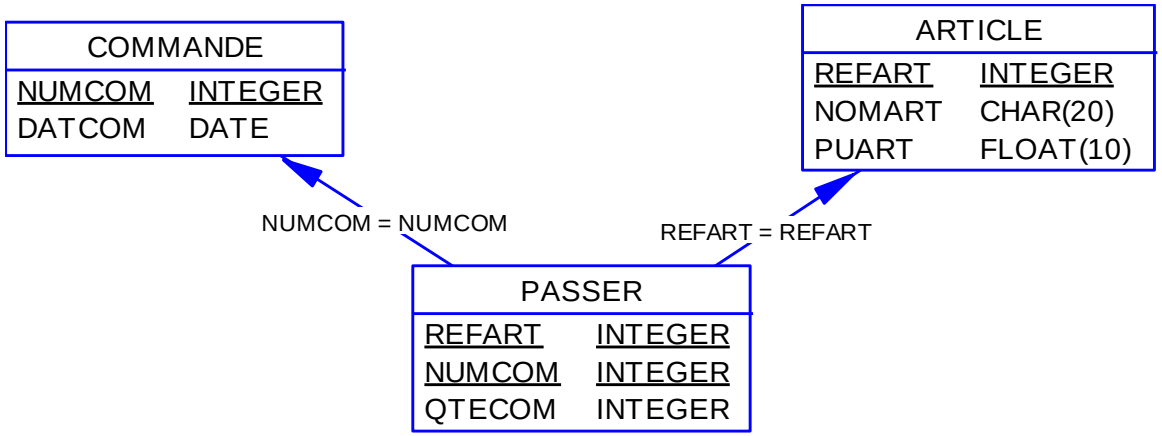
CLIENT (NUMCLI, NOMCLI, CATCLI)
COMMANDE (NUMCOM, DATCOM, NUMCLI↑)



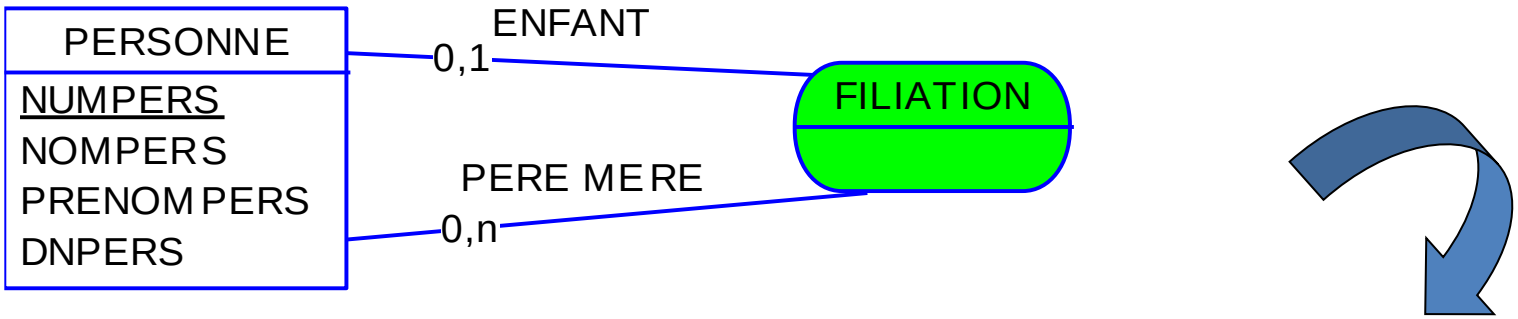
Exemple 2



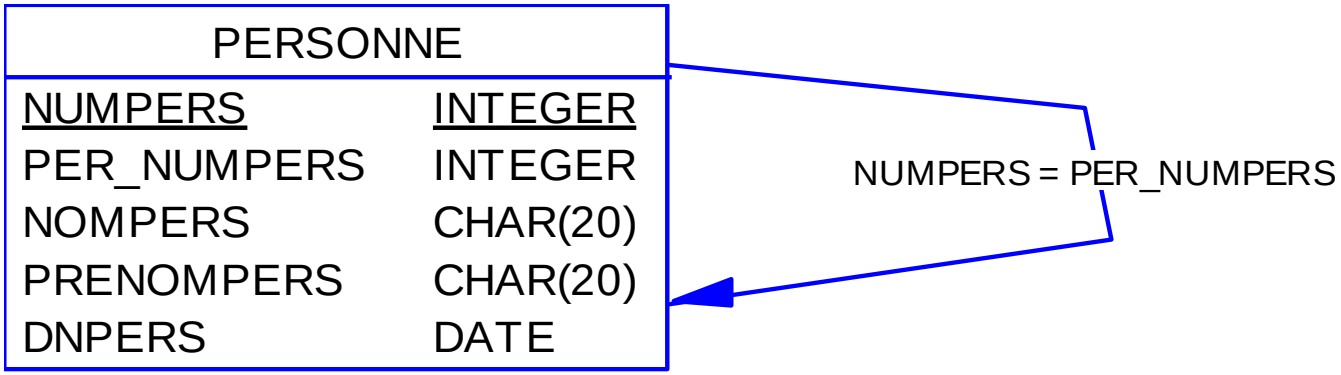
ARTICLE (REFART, NOMART, PUART)
COMMANDE (NUMCOM, DATCOM)
PASSER (NUMCOM ↑, REFART ↑, QTECOM)



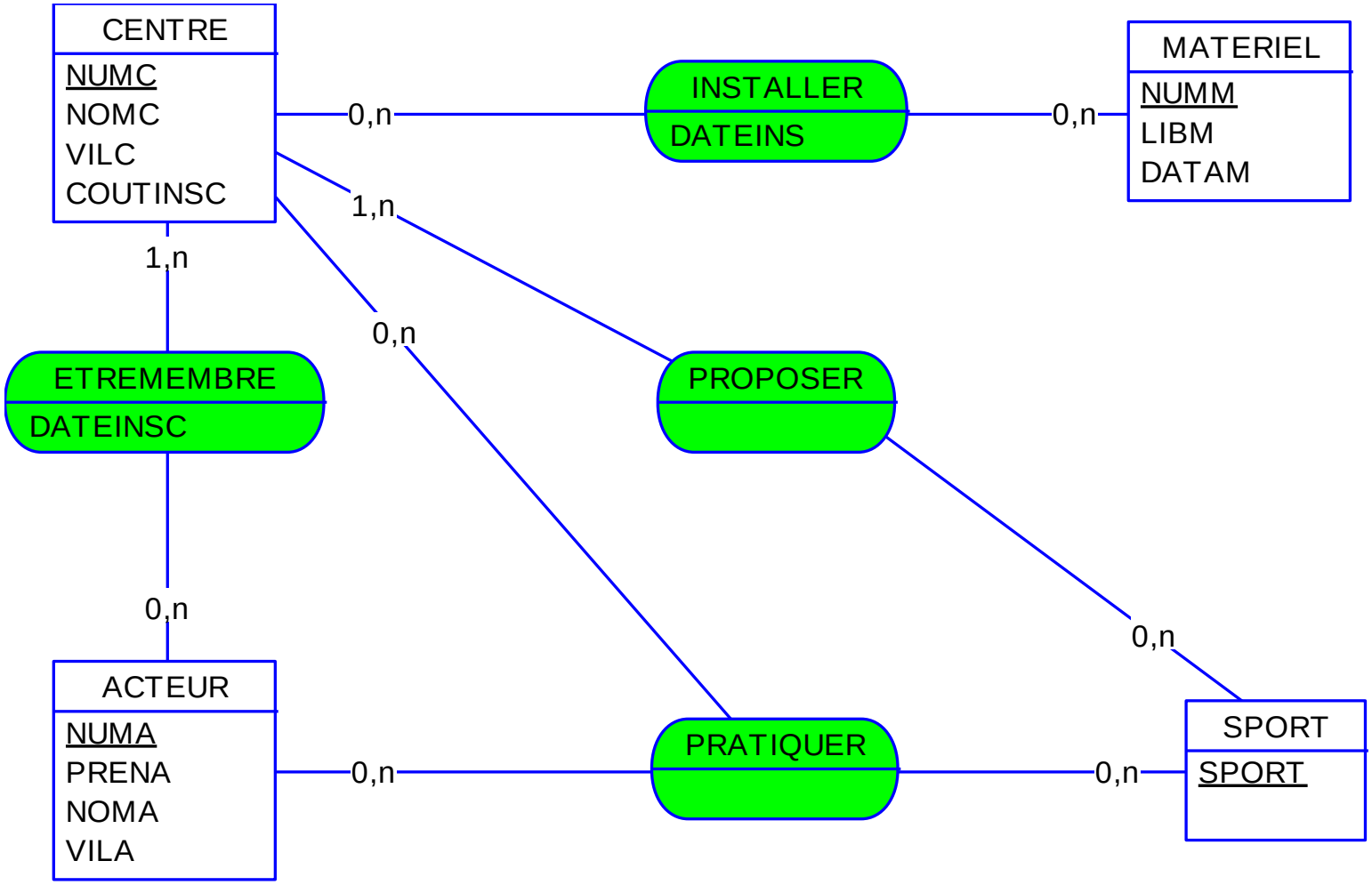
Exemple 2



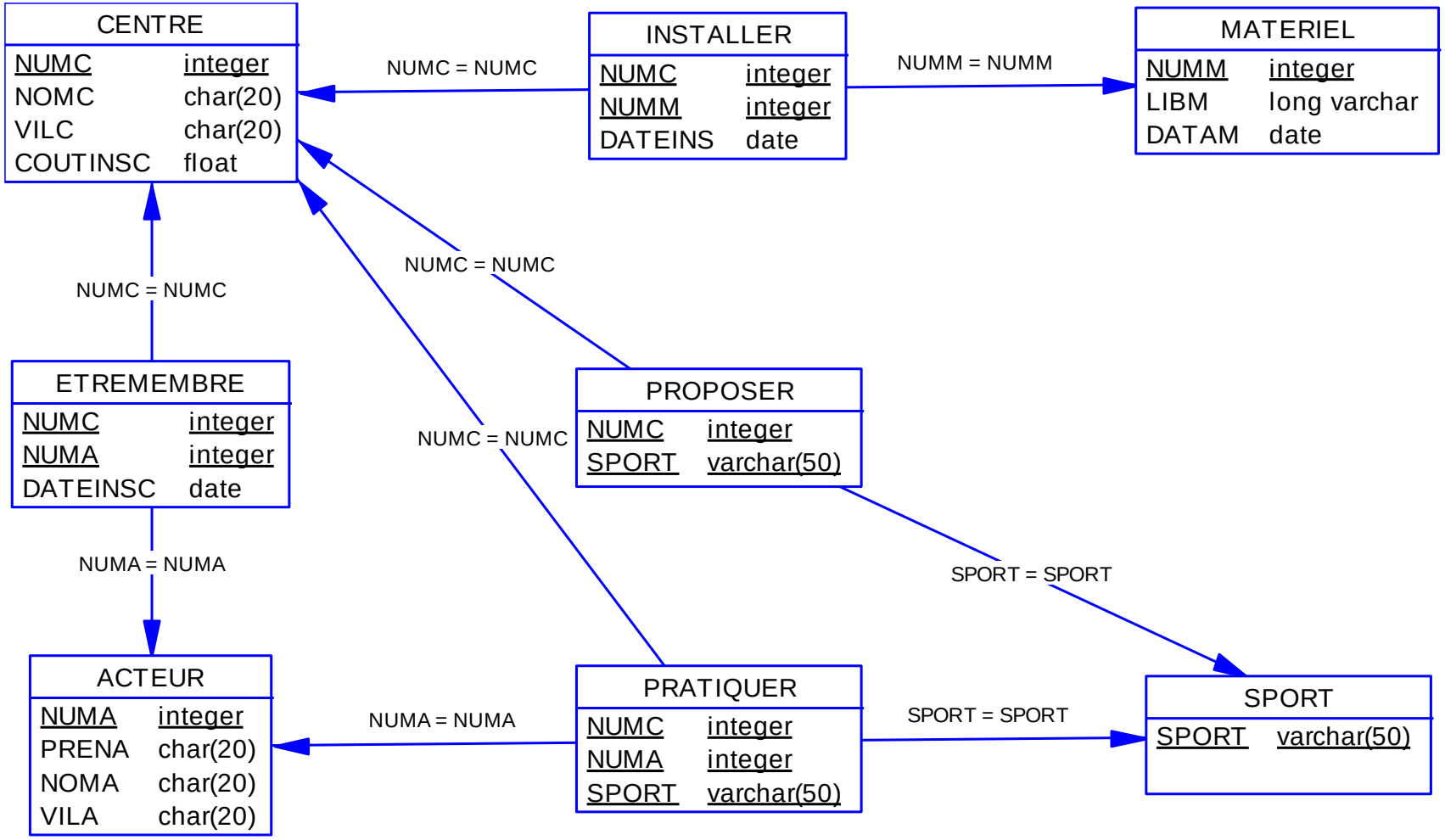
PERSONNE (NUMPERS, NOMPERS, PRENOMPERS, DNPERS, NUMPERSpar↑)



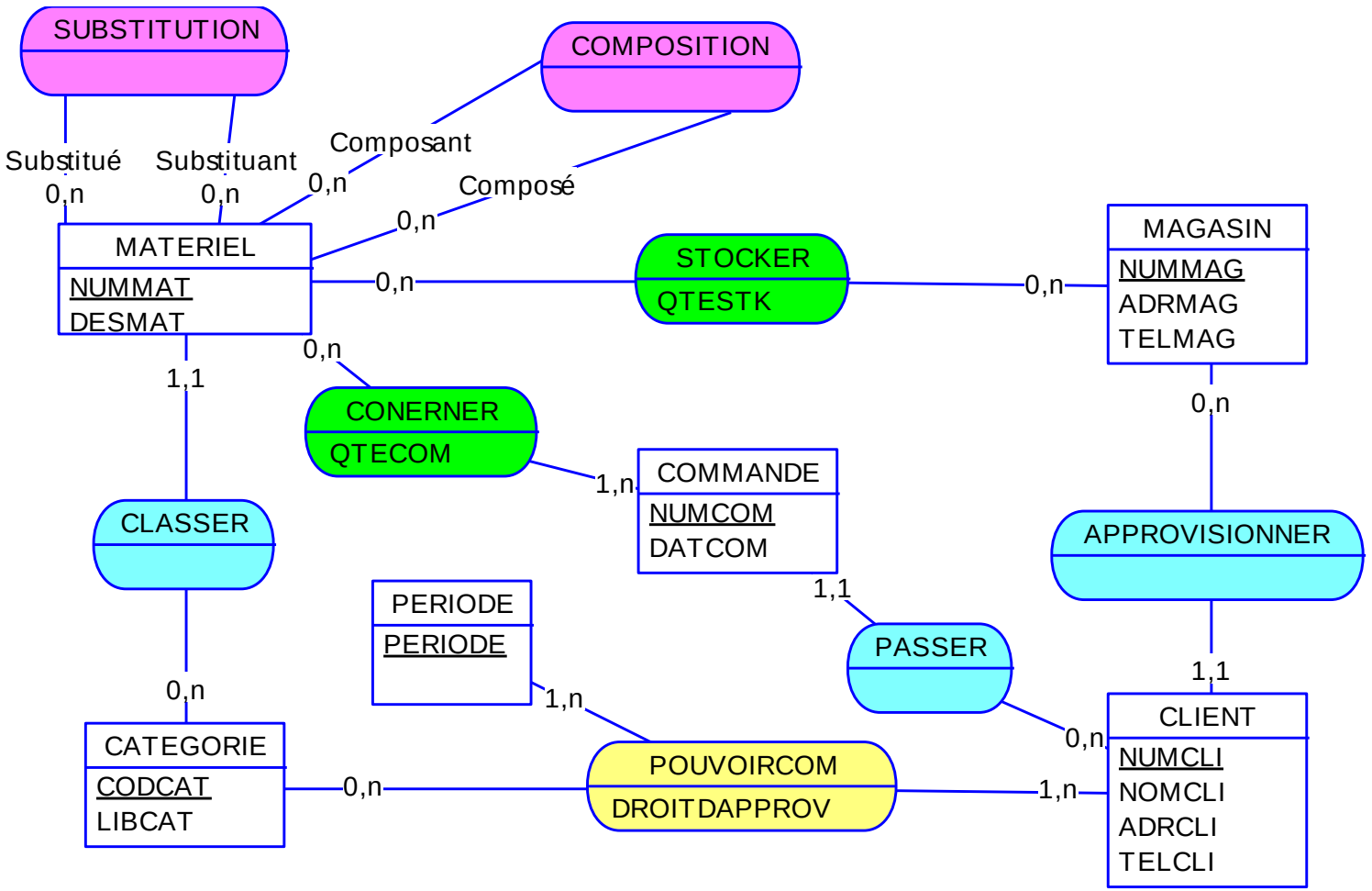
Exemple 2


CENTRE
NUMC
NOMC
VILC
COUTINSC
INSTALLER
DATEINS
MATERIEL
NUMM
LIBM
DATAM
0,n
0,n
1,n
1,n
0,n
ETREMEMBRE
DATEINSC
PROPOSER
0,n
0,n
ACTEUR
NUMA
PRENA
NOMA
VILA
PRATIQUER
SPORT
SPORT
0,n
0,n

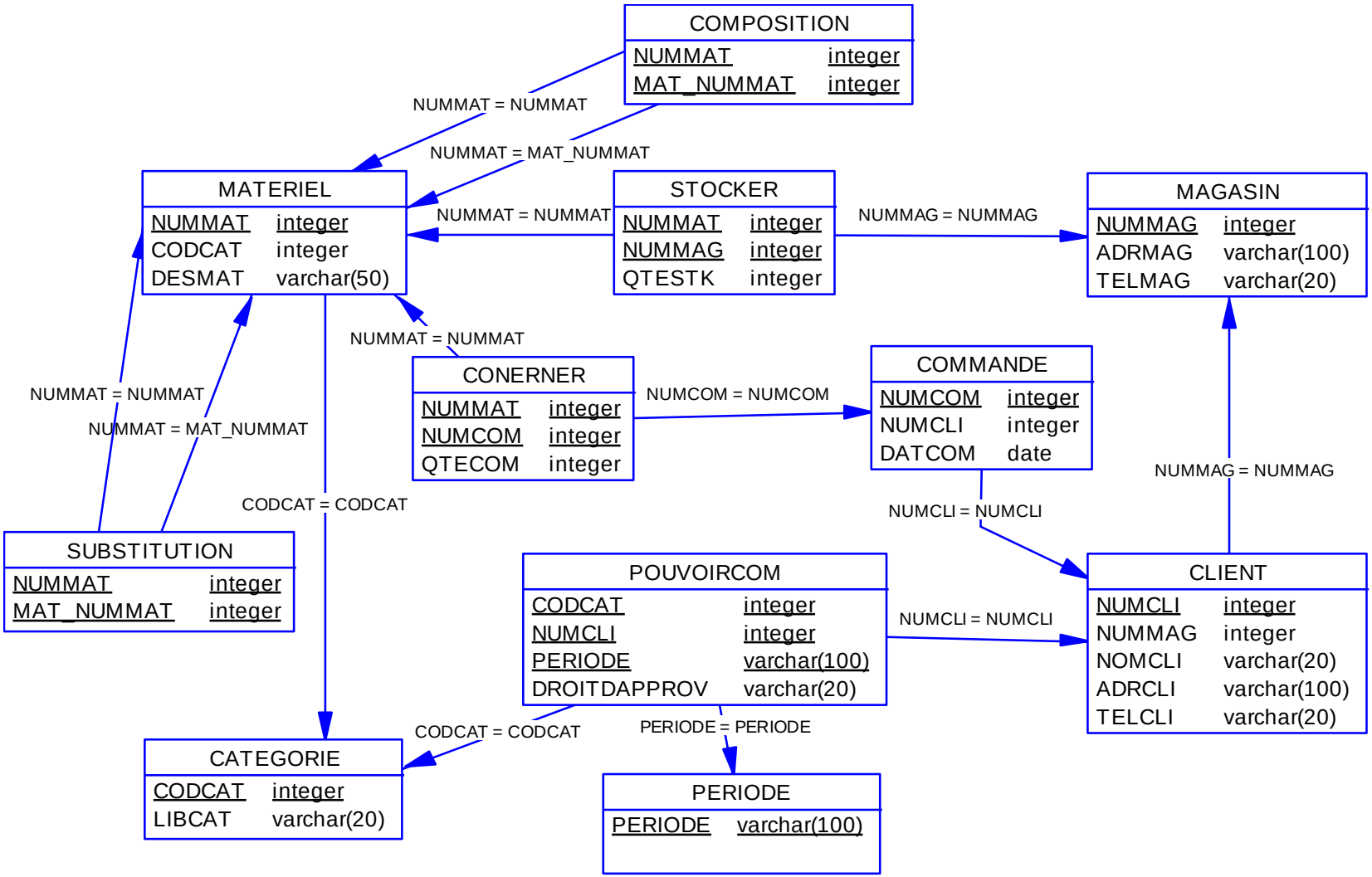
Exemple 2


CENTRE
NUMC integer
NOMC char(20)
VILC char(20)
COUTINSC float
INSTALLER
NUMC integer
NUMM integer
DATEINS date
MATERIEL
NUMM integer
LIBM long varchar
DATAM date
NUMC = NUMC
NUMM = NUMM
NUMC = NUMC
NUMC = NUMC
ETREMEMBRE
NUMC integer
NUMA integer
DATEINSC date
PROPOSER
NUMC integer
SPORT varchar(50)
NUMC = NUMC
NUMA = NUMA
SPORT = SPORT
ACTEUR
NUMA integer
PRENA char(20)
NOMA char(20)
VILA char(20)
PRATIQUER
NUMC integer
NUMA integer
SPORT varchar(50)
SPORT
SPORT varchar(50)
NUMA = NUMA
SPORT = SPORT

Exemple 2


SUBSTITUTION
COMPOSITION
Substitué
0,n
Substituant
0,n
Composant
0,n
Composé
0,n
MATERIEL
NUMMAT
DESMAT
0,n
STOCKER
QTESTK
0,n
MAGASIN
NUMMAG
ADRMAG
TELMAG
1,1
0,n
CONERNER
QTECOM
1,n
COMMANDE
NUMCOM
DATCOM
0,n
CLASSER
APPROVISIONNER
PERIODE
PERIODE
1,1
PASSER
1,1
0,n
1,n
CATEGORIE
CODCAT
LIBCAT
0,n
POUVOIRCOM
DROITDAPPROV
1,n
0,n
CLIENT
NUMCLI
NOMCLI
ADRCLI
TELCLI

Exemple 2


COMPOSITION
NUMMAT integer
MAT_NUMMAT integer
NUMMAT = NUMMAT
NUMMAT = MAT_NUMMAT
MATERIEL
NUMMAT integer
CODCAT integer
DESMAT varchar(50)
NUMMAT = NUMMAT
STOCKER
NUMMAT integer
NUMMAG integer
QTESTK integer
NUMMAG = NUMMAG
MAGASIN
NUMMAG integer
ADRMAG varchar(100)
TELMAG varchar(20)
NUMMAT = NUMMAT
CONERNER
NUMMAT integer
NUMCOM integer
QTECOM integer
NUMCOM = NUMCOM
COMMANDE
NUMCOM integer
NUMCLI integer
DATCOM date
NUMMAT = NUMMAT
NUMMAT = MAT_NUMMAT
CODCAT = CODCAT
NUMCLI = NUMCLI
NUMMAG = NUMMAG
SUBSTITUTION
NUMMAT integer
MAT_NUMMAT integer
POUVOIRCOM
CODCAT integer
NUMCLI integer
PERIODE varchar(100)
DROITDAPPROV varchar(20)
NUMCLI = NUMCLI
CLIENT
NUMCLI integer
NUMMAG integer
NOMCLI varchar(20)
ADRCLI varchar(100)
TELCLI varchar(20)
CODCAT = CODCAT
PERIODE = PERIODE
CATEGORIE
CODCAT integer
LIBCAT varchar(20)
PERIODE
PERIODE varchar(100)

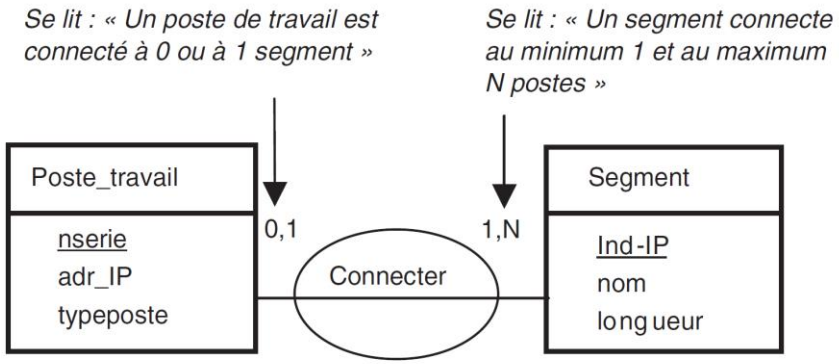
Différences E/A - UML

Entité-association	UML
Entité	Classe
Association (Relation)	Association (Relation)
Occurrence	Objet
Cardinalité	Multiplicité
Modèle conceptuel de donnés (Merise)	Diagramme de classes

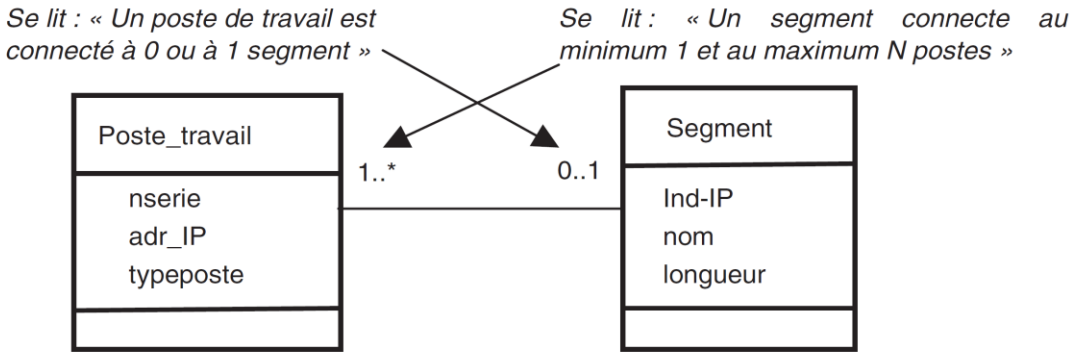
Cardinalités	Multiplicités d'UML
0,1	0..1
1,1	1
0,N	0..* ou *
1,N	1..*

Différences E/A - UML

E/A :



UML :



Les formes normales

Il existe de nombreuses formes normales, qui assurent certaines propriétés d'une relation.

Cependant, seules les premières sont d'un intérêt pratique :

- La **Première Forme Normale** (**1FN**)
- La **Deuxième Forme Normale** (**2FN**)
- La **Troisième Forme Normale** (**3FN**)
- La **Forme normale de Boyce-Codd** (**FNBC**)

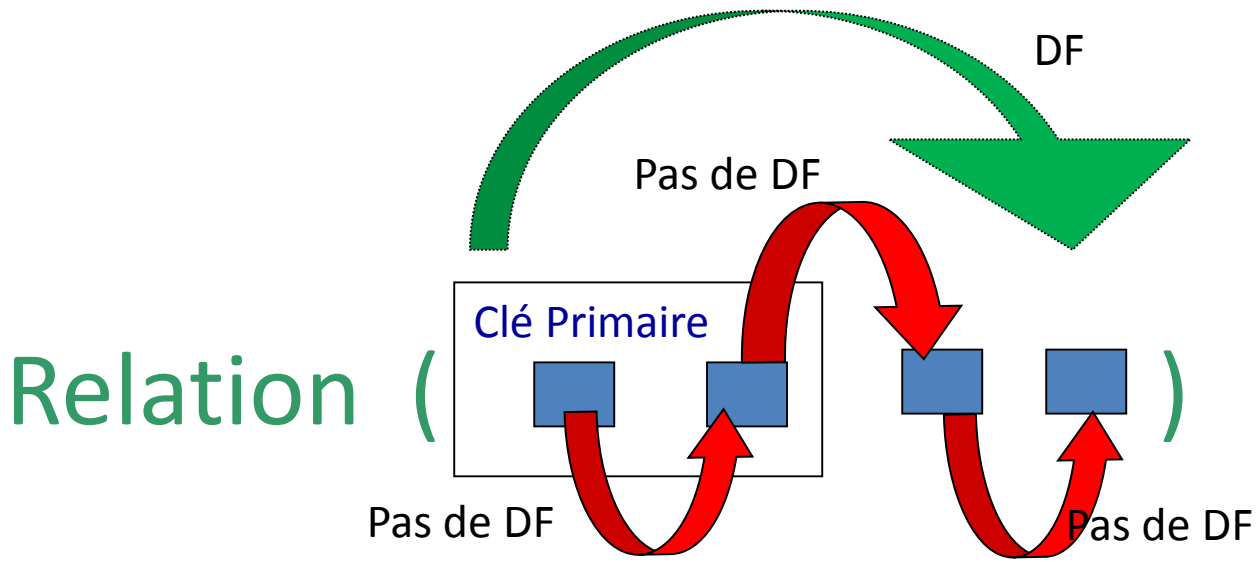
Concepts de base

Notion de dépendance fonctionnelle :

On dit qu'il existe une dépendance fonctionnelle entre deux attributs X et Y si la connaissance de la valeur de X détermine celle de Y

Concepts de base

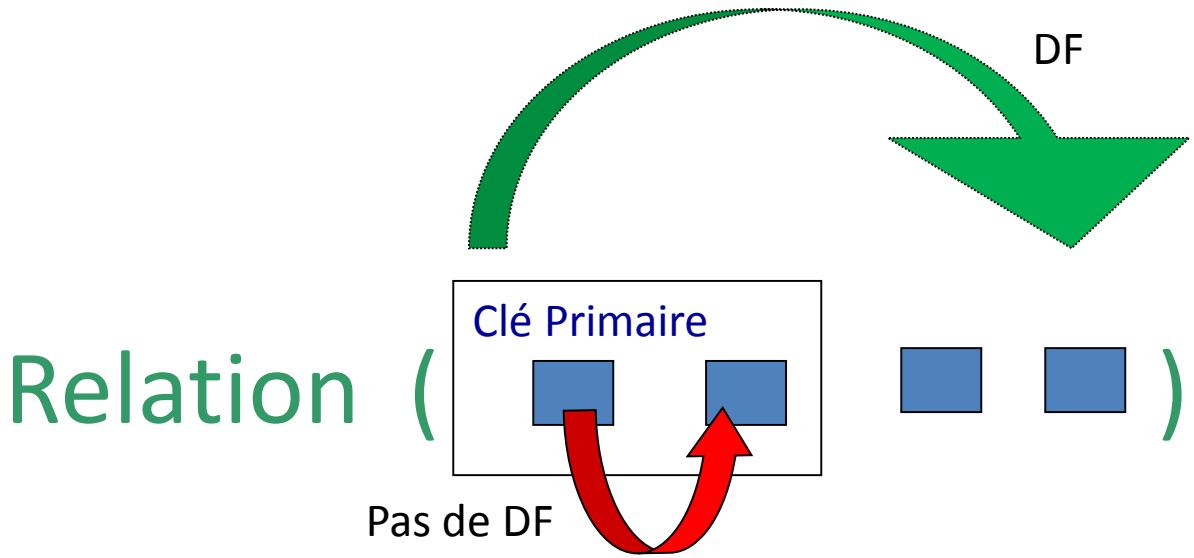
Format d'une relation « bien conçue » :



Première forme normale (1FN)

Une relation est en 1FN si

- **chaque attribut non clé dépend de la clé**
- **Les attribut contiennent des valeurs unitaires**



Première forme normale (1FN)

Par exemple les relations ci-dessous **ne sont pas en 1FN** :

NSS du PERE	LISTE des PRENOMS de ses ENFANTS
1	ALADIN
2	ADAM, EVE
3	
4	JEAN, CAROLE, PAUL, CECILE

Une Personne
EST-PERE-DE
enfant(s)

Un centre
sportif
PROPOSE
des sports

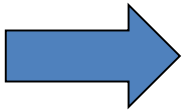
NUMC	LISTE des SPORTS proposés par le centre
1O1	FOOTBALL, KARTING, TENNIS
1O2	NATATION
1O3	FOOTBALL, HANDBALL, BASKETBALL

Première forme normale (1FN)

Transformations en première forme normale :

EST-PERE-DE **NON 1FN**

NSS du PERE	LISTE des PRENOMS de ses ENFANTS
1	ALADIN
2	ADAM, EVE
3	
4	JEAN, CAROLE, PAUL, CECILE



EST-PERE-DE **1FN**

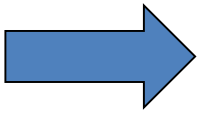
NSS du PERE	PRENOM de l'ENFANT
1	ALADIN
2	ADAM
2	EVE
3	? (valeur nulle)
4	CAROLE
4	CECILE
4	JEAN
4	PAUL

Première forme normale (1FN)

Transformations en première forme normale :

PROPOSE **NON 1FN**

NUMC	LISTE des SPORTS proposés par le centre
1O1	FOOTBALL, KARTING, TENNIS
1O2	NATATION
1O3	FOOTBALL, HANDBALL, BASKETBALL

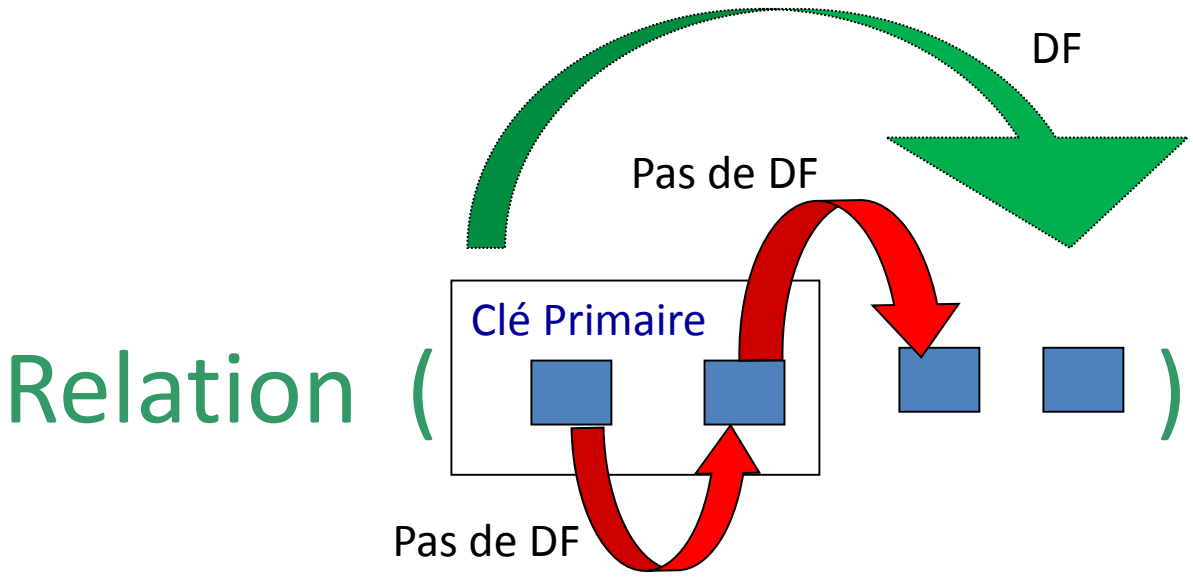


PROPOSE **1FN**

NUMC	SPORT
1O1	FOOTBALL
1O1	KARTING
1O1	TENNIS
1O2	NATATION
1O3	FOOTBALL
1O3	HANDBALL
1O3	BASKETBALL

Deuxième forme normale (2FN)

- **Est en 1FN**
- **Aucun attribut hors clé ne dépend que d'une partie de la clé**



Deuxième forme normale (2FN)

Exemples:

La relation **ARTICLE** ci-dessous **n'est pas en deuxième forme normale** :

NumA	NumD	NomA	PrixA	NomD	TélD
A100	D1	Ballon de Foot	125	BELAIR	0145750679
A200	D1	Ballon de Basket	85	BELAIR	0145750679
A300	D1	Ballon de Hand	63	BELAIR	0145750679
A100	D2	Ballon de Foot	125	VASPORT	0149495051
A400	D2	Chaussures de Foot	92	VASPORT	0149495051

Le schéma de la relation ARTICLE est :

ARTICLE (**NuméroArticle, NuméroDépot**, NomArticle, PrixArticle, NomDépot, TéléphoneDépot)

Deuxième forme normale (2FN)

La relation **ARTICLE** ci-dessous **n'est pas en deuxième forme normale** :

NumA	NumD	NomA	PrixA	NomD	TélD
A100	D1	Ballon de Foot	125	BELAIR	0145750679
A200	D1	Ballon de Basket	85	BELAIR	0145750679
A300	D1	Ballon de Hand	63	BELAIR	0145750679
A100	D2	Ballon de Foot	125	VASPORT	0149495051
A400	D2	Chaussures de Foot	92	VASPORT	0149495051

Le nom de l'article ne dépend que d'une partie de la clé (NuméroArticle).
Il en est de même pour le prix de l'article.

Le nom et le téléphone du dépôt dépendent fonctionnellement du Numéro du Dépot.

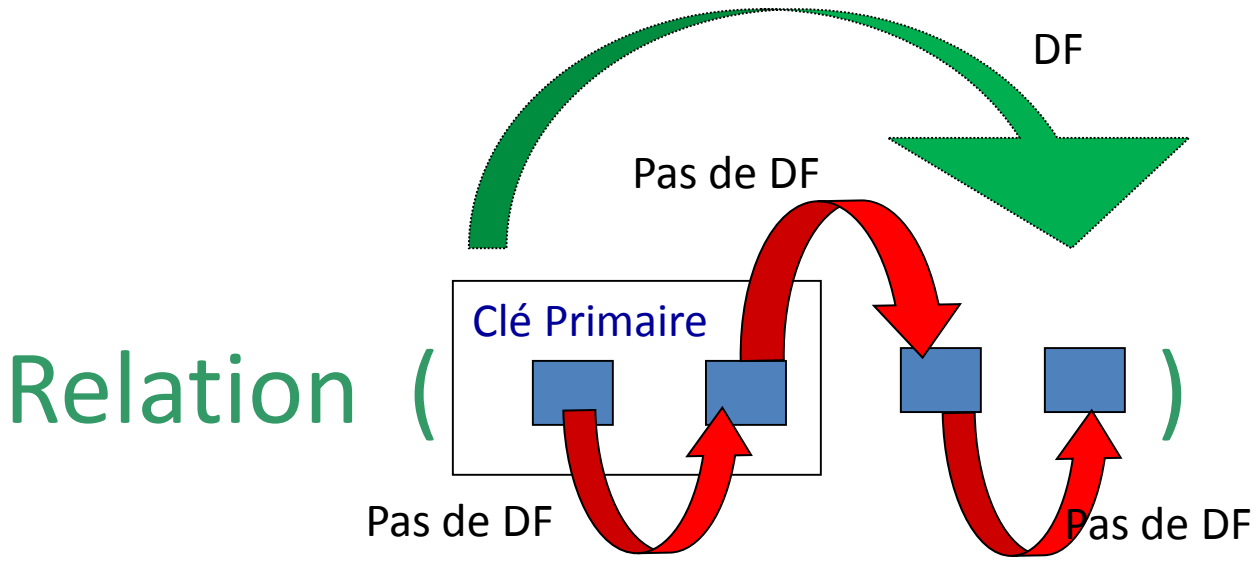
Deuxième forme normale (2FN)

Remarques :

- Une relation en 1FN dont la clé primaire est composée d'un seul attribut est forcément en 2FN
- Une relation en 2FN présente des anomalies de mises à jour.

Troisième forme normale (3FN)

1. **Est en 2NF**
2. **Tout attribut non clé ne dépend pas d'un autre attribut non clé**



Troisième forme normale (3FN)

Exemple : Étant donné la relation Professeur, en 2FN (non 3FN), décrite comme suit :

Professeur (**NumProf**, NomProf, IdCours, VolumeHorC, CoeffC)

Avec NumProf comme clé primaire et les DF ci-dessous :

NumProf → NomProf

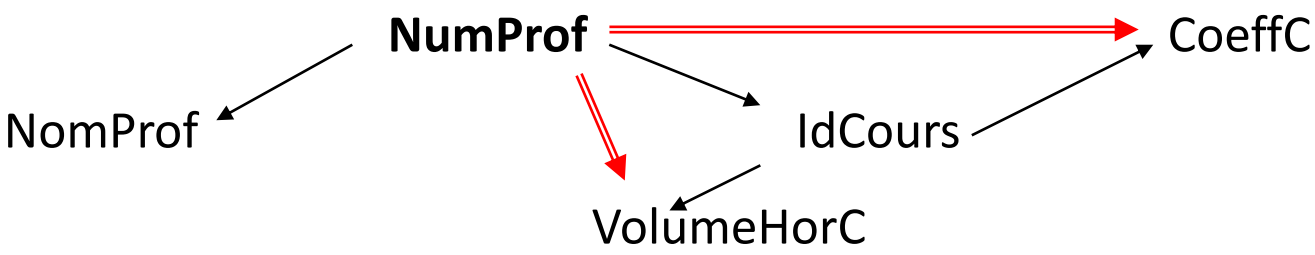
NumProf → IdCours

NumProf → VolumeHorC

NumProf → CoeffC

IdCours → VoulmeHorC

IdCours → CoeffC



Troisième forme normale (3FN)

Exemple : Étant donné la relation Professeur, en 2FN (non 3FN), décrite comme suit :

Professeur (**NumProf**, NomProf, IdCours, VolumeHorC, CoeffC)

Avec NumProf comme clé primaire et les DF ci-dessous :

NumProf → NomProf

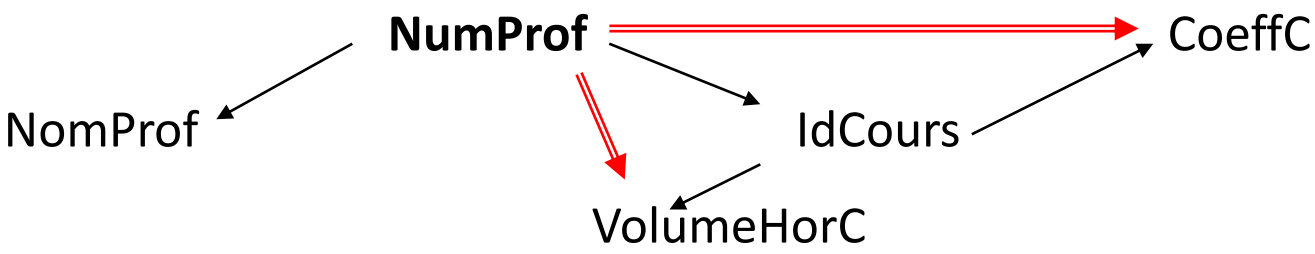
NumProf → IdCours

NumProf → VolumeHorC

NumProf → CoeffC

IdCours → VoulmeHorC

IdCours → CoeffC

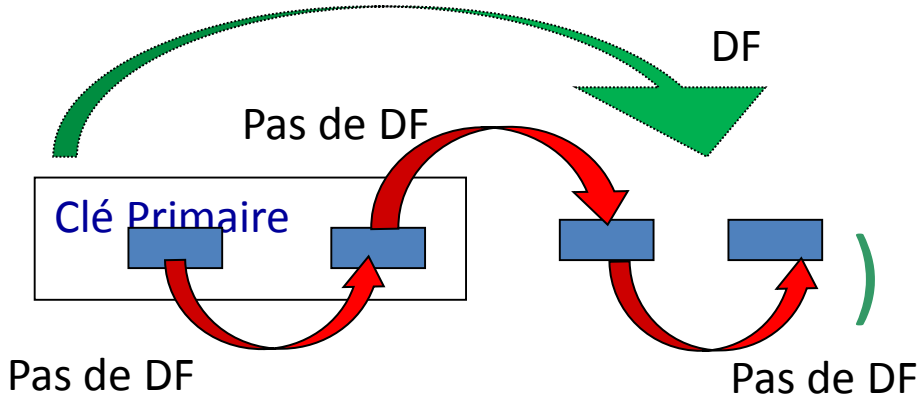


La relation *Professeur* n'est pas 3FN car la DF NumProf → CoefC n'est pas directe. La relation *Professeur* présente des anomalies de mise à jour.

Méthode pour être en 3FN

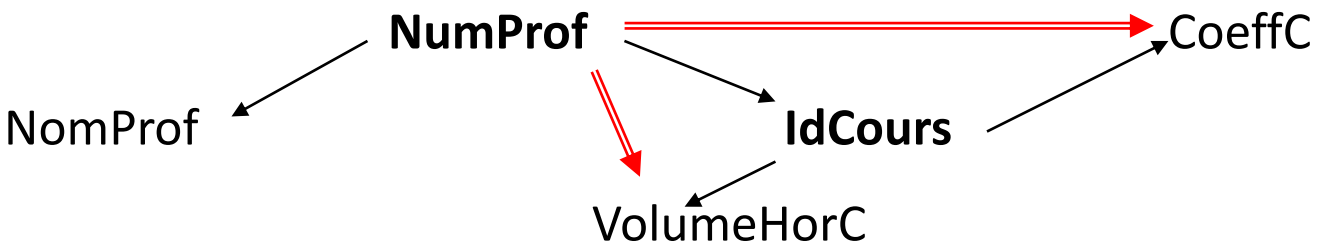
1. **Vérifier que les attribut contiennent des valeurs unitaires**
2. **Définir une clé unique (la créer si besoin: identifiant)**
3. **Si il reste des dépendances entre attributs non clé, diviser en plusieurs tables, chacune avec une clé unique.**

Relation (



)

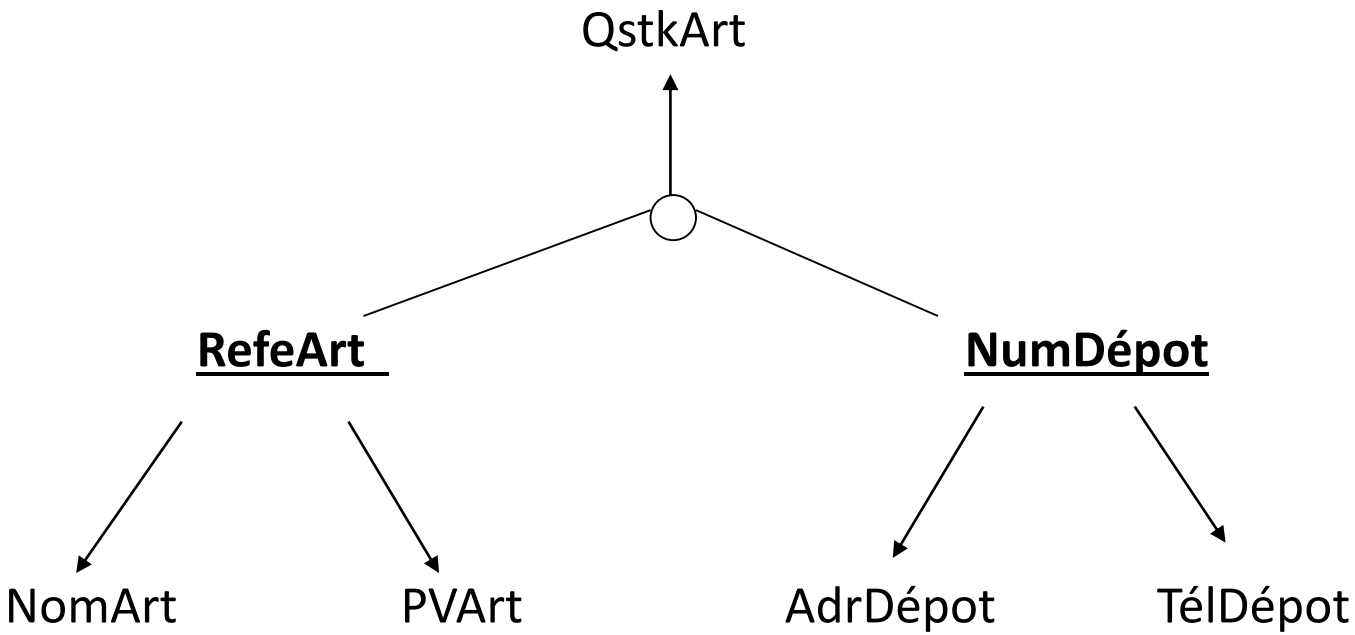
Méthode graphique



Professeur (**NumProf**, NomProf, IdCours*)

Cours(**IdCours**, VolumeHorC, CoeffC

Méthode graphique



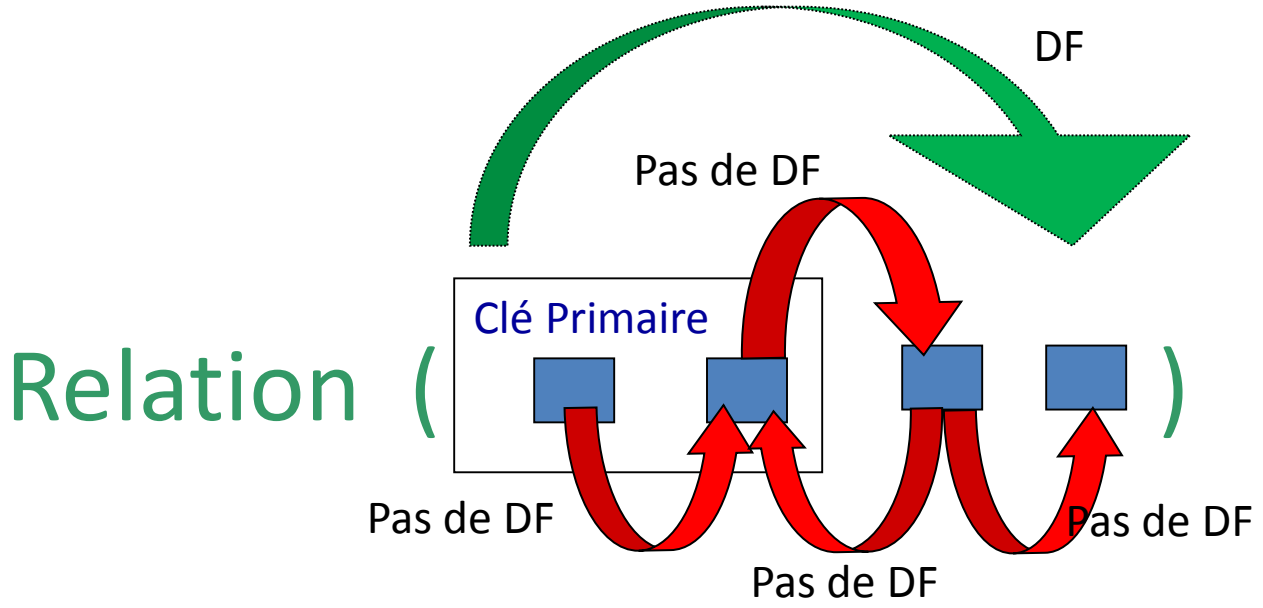
Produit (**RefArt**, NomArt, PVArt)

Dépôt (**NumDépot**, AdrDépot, TélDépot)

Stock (**RefArt, NumDépot**, QstkArt)

Forme normale Boyce-Codd (FNBC)

1. **Est en 3FN**
2. **Chaque attribut hors clé dépend uniquement et entièrement de la clé primaire**
3. **Aucun attribut de la clé ne dépend d'un autre attribut**



Forme normale Boyce-Codd (FNBC)

Exemple en 3FN mais pas en FNBC :

Vin(Cru, Pays, Région) avec Région → Pays

Une décomposition en FNBC possible ?

Forme normale Boyce-Codd (FNBC)

Exemple en 3FN mais pas en FNBC :

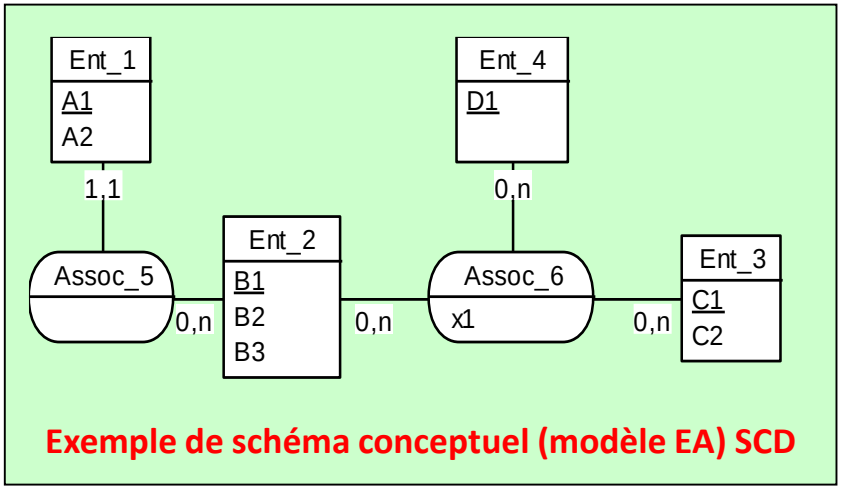
Vin(Cru, Pays, Région) avec Région → Pays

Une décomposition en FNBC possible :

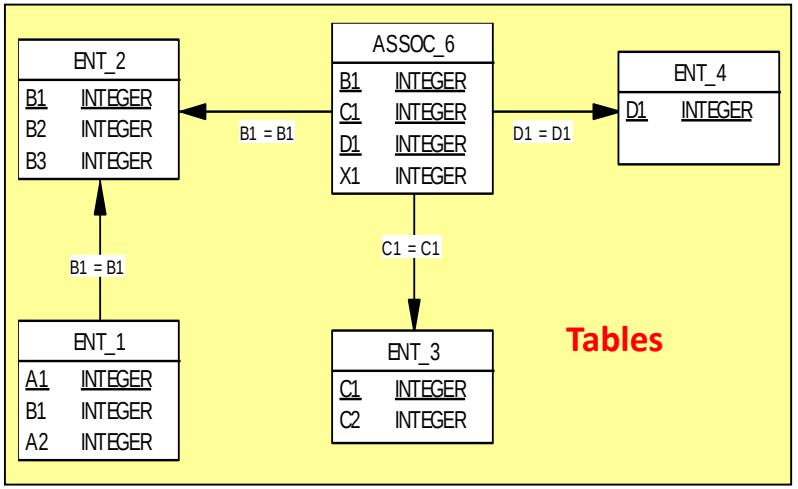
Régions(Région, Pays)
Crus(Cru, Région)

Pas de perte d'information mais perte de dépendance fonctionnelle (Cru, Pays → Région).
Parfois il est préférable de rester en 3FN…

Du concept au modèle logique



Exemple de schéma conceptuel (modèle EA) SCD



R1_ENT_2 (**B1**, B2, B3)
R2_ENT_1 (**A1**, A2, *B1* *)
R3_ENT_3 (**C1**, C2)
R4_ENT_4 (**D1**)

R5_ASSOC_6 (**B1*, C1*, D1***, X1)

Exemple de schéma logique (modèle relationnel) SLD

Implémentation du modèle logique

Les SGBD vérifient l'intégrité des données via :

- La déclaration de contraintes (*constraints*)
- La programmation de :
  - fonctions (*functions*)
  - procédures (*procedures*)
  - déclencheurs (*triggers*)

Le principe étant d'assurer la cohérence de la base après chaque mise à jour.

Le langage d'interaction avec le SGBD est le SQL

Le langage SQL

- Les Systèmes de Gestion de Bases de Données (SGBD) présentent une interface externe sous forme de langage de requêtes.
- Celui-ci permet de spécifier les données à sélectionner ou à mettre à jour.
- Il n'est pas nécessaire de spécifier comment retrouver les informations (le SGBD s'en charge) mais seulement de quelles informations on a besoin.

Le langage SQL

Le langage SQL ne présente intentionnellement qu'un nombre limité de verbes ou mots-clés, qui se répartissent en trois familles fonctionnellement distinctes :

1. Le **Langage de Définition de Données** (**LDD** ou DDL) permet la description de la structure de la base de données (tables, vues, attributs, index...). Les mots clés utilisés sont:

   CREATE, DROP et ALTER.

Le langage SQL

Le langage SQL ne présente intentionnellement qu'un nombre limité de verbes ou mots-clés, qui se répartissent en trois familles fonctionnellement distinctes :

2. Le **Langage de Manipulation de Données** (**LMD** ou DML) permet la manipulation des tables et des vues avec ses quatre verbes correspondant aux opérations fondamentales sur les données:

   INSERT, DELETE, UPDATE et SELECT

Le langage SQL

Le langage SQL ne présente intentionnellement qu'un nombre limité de verbes ou mots-clés, qui se répartissent en trois familles fonctionnellement distinctes :

3. Le **Langage de Contrôle de Données** (**LCD** ou DCL) contient les primitives de gestion des transactions:

   COMMIT et ROLLBACK

   et des privilèges d'accès aux données

   GRANT et REVOKE

Le langage SQL

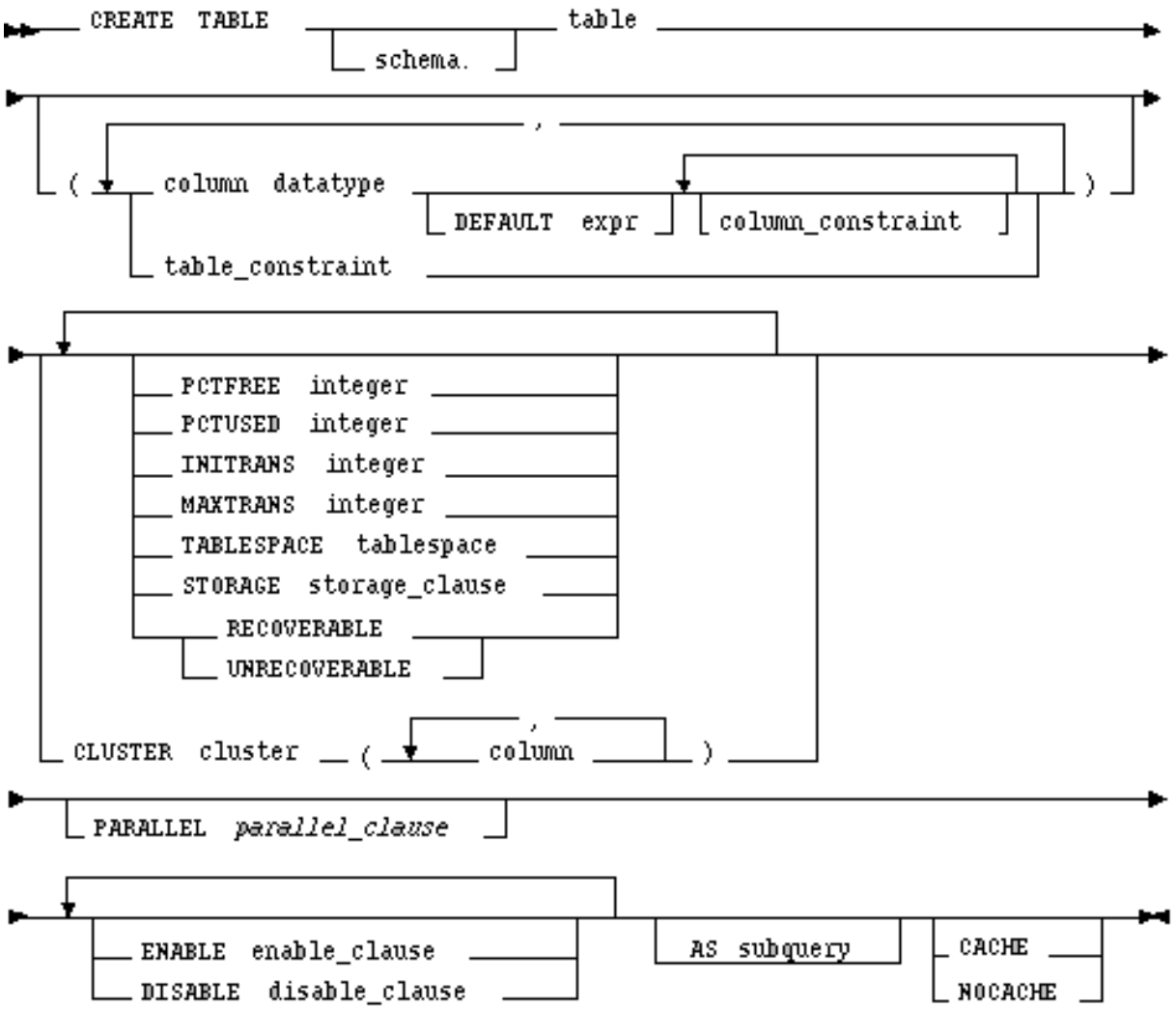
CREATE ALTER DROP RENAME TRUNCATE	Langage de Définition de Données (LDD)
INSERT UPDATE DELETE SELECT	Langage de Manipulation de Données (LMD)
GRANT REVOKE COMMIT ROLLBACK SAVEPOINT	Langage de Contrôle de Données (LCD)

Partie III

-

Définition et manipulation de données

Création de tables



Création de tables

Création simple:

CREATE TABLE nom_table (Nom_col1 TYPE1, Nom_col2 TYPE2, …);

Les **domaines** (types de données) les plus utiles dans ORACLE sont :

- **NUMBER(x)** pour les entiers (optionnel : de longueur x)
- **NUMBER(x,y)** réels (optionnel : de longueur totale x dont y décimales)
- **CHAR(n)** chaines de caractères de longueur n
- **VARCHAR2(n)** chaines de caractères de longueur variable, n max
- **DATE** date, de la forme 01/11/81 ou 01-nov-81
- **TIMESTAMP** date et heure

Création de tables

Valeurs par défaut:

CREATE TABLE nom_table (Nom_col1 TYPE1 DEFAULT valeur, …);

Exemple

```
CREATE TABLE films
(
        idFilm NUMBER,
        Titre VARCHAR2(255),
        Réalisateur VARCHAR2(255) DEFAULT 'Unknown'
);
```

Création de tables

On peut insérer des données dans une table lors de sa création :

CREATE TABLE Nom_table (Nom_col1 TYPE1, Nom_col2 TYPE2, …)
AS SELECT …..

Création de tables : contraintes

Le **contrôle de la validité des données** se fait par la définition de contraintes :

```
CREATE TABLE nom_table (
nom_col_1 type_1                [CONSTRAINT nom_1_1] contrainte_de_colonne_1_1
                                [CONSTRAINT nom_1_2] contrainte_de_colonne_1_2
                                ...
                                [CONSTRAINT nom_1_m] contrainte_de_colonne_2_m,
nom_col_2 type_2                [CONSTRAINT nom_2_1] contrainte_de_colonne_2_1
                                [CONSTRAINT nom_2_2] contrainte_de_colonne_2_2
                                ...
                                [CONSTRAINT nom_2_m] contrainte_de_colonne_2_m,
...
nom_col_n type_n                [CONSTRAINT nom_n_1] contrainte_de_colonne_n_1
                                [CONSTRAINT nom_n_2] contrainte_de_colonne_n_2
...
                                [CONSTRAINT nom_n_m] contrainte_de_colonne_n_m,

[CONSTRAINT nom_1] contrainte_de_table_1,
[CONSTRAINT nom_2] contrainte_de_table_2,
...
[CONSTRAINT nom_p] contrainte_de_table_p
)
```

Création de tables : contraintes

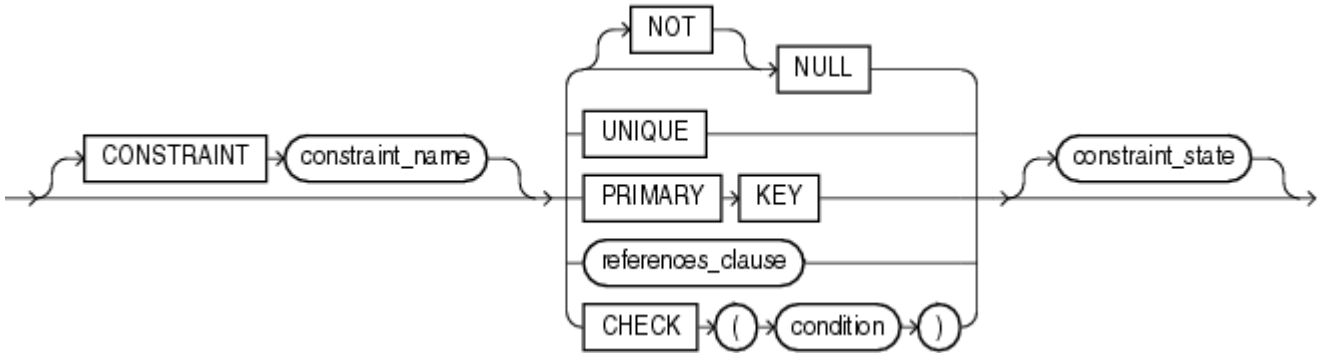
Il y a deux types de contraintes :

- Les contraintes de colonne portent sur une seule colonne
- Les contraintes de table portent sur un ensemble de colonnes

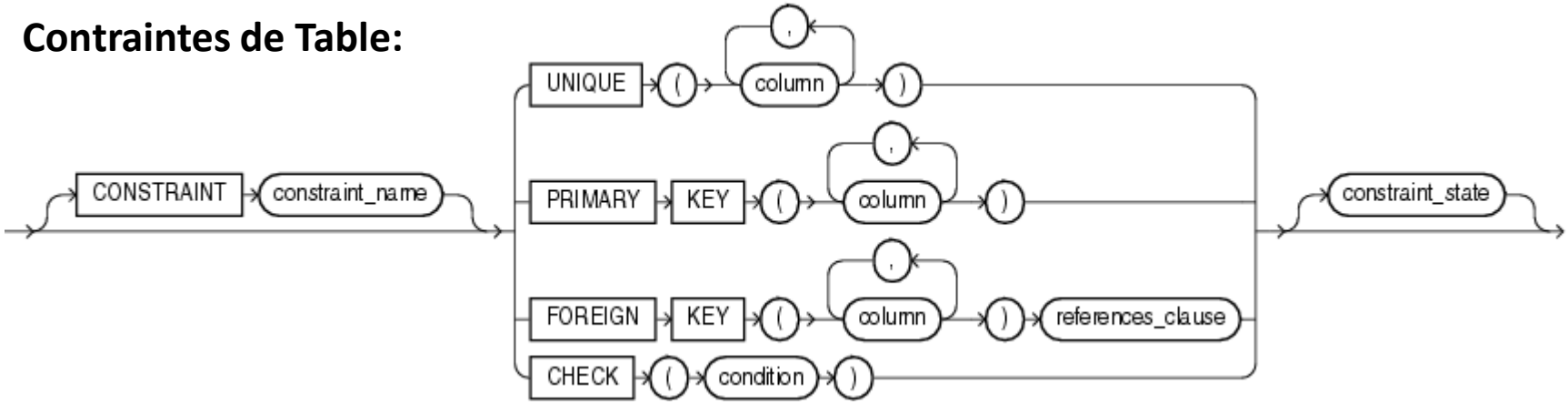
Il est cependant souvent possible d'écrire une contrainte de colonne sous forme de contrainte de table !

Création de tables : contraintes

Contraintes de colonne :



Contraintes de Table:



Création de tables : contraintes

NOT NULL :

Par défaut, il n'est pas obligatoire de donner une valeur à chaque attribut pour une ligne de donnée. L'attribut prend la valeur NULL dans ce cas si aucune valeur par défaut n'a été spécifiée.

id	domain_id	name	type	content	ttl	prio	change_date
1	1	test.com	SOA	localhost ahu@ds9a.nl 1	86400	*NULL*	*NULL*
2	1	test.com	NS	dns-us1.powerdns.net	86400	*NULL*	*NULL*
3	1	test.com	NS	dns-eu1.powerdns.net	86400	*NULL*	*NULL*
4	1	www.test.com	A	199.198.197.196	120	*NULL*	*NULL*
5	1	mail.test.com	A	195.194.193.192	120	*NULL*	*NULL*
6	1	localhost.test.com	A	127.0.0.1	120	*NULL*	*NULL*
7	1	test.com	MX	mail.test.com	120	25	*NULL*

Création de tables : contraintes

NOT NULL :

Si on veux interdire des valeurs NULL pour un attribut, on utilise la contrainte NOT NULL:

```
CREATE TABLE supplier
(
        supplier_id     NUMBER(10)      NOT NULL,
        supplier_name   VARCHAR2(50)    NOT NULL,
        contact_name    VARCHAR2(50)
);
```

Création de tables : contraintes

UNIQUE :

Si on veux que la valeur d'un attribut soit unique pour chaque donnée, on utilise la contrainte UNIQUE:

```
CREATE TABLE supplier
(
        supplier_id     NUMBER(10)      NOT NULL        UNIQUE ,
        supplier_name   VARCHAR2(50)    NOT NULL,
        contact_name    VARCHAR2(50)
);
```

Création de tables : contraintes

UNIQUE :

Si on veux que la valeur d'un attribut soit unique pour chaque donnée, on utilise la contrainte UNIQUE.

Un attribut NOT NULL UNIQUE peut être choisi comme clé primaire !

Création de tables : contraintes

PRIMARY KEY:

Si on veux définir la clé primaire de la table, on utilise la contrainte PRIMARY KEY :

```
CREATE TABLE supplier
(
        supplier_id     NUMBER(10)      PRIMARY KEY,
        supplier_name   VARCHAR2(50)    NOT NULL,
        contact_name    VARCHAR2(50)
);
```

Création de tables : contraintes

REFERENCES :

Si on veux définir une clé étrangère de la table, on utilise la contrainte REFERENCES table [(colonne)] [ON DELETE CASCADE | SET NULL] :

```
CREATE TABLE supplier
(
        supplier_id     NUMBER(10)      PRIMARY KEY,
        supplier_name   VARCHAR2(50)    NOT NULL,
        contact_id      NUMBER(10)      REFERENCES contact(ID)
        contact_name    VARCHAR2(50)
);
```

Création de tables : contraintes

REFERENCES :

Si on veux définir une clé étrangère de la table, on utilise la contrainte REFERENCES table [(colonne)] [ON DELETE CASCADE | SET NULL].

Remarques:

1. **Une clé étrangère pointe toujours vers un attribut (ou un ensemble d'attributs) unique.**
2. **Si le nom de colonne n'est pas précisé, c'est la clé primaire de la table de destination qui est choisie par défaut.**
3. **ON DELETE supprime (CASCADE) les lignes ou les attributs clé étrangère (SET NULL) ne pointant plus vers une ligne qui vient d'être supprimée de la table de destination.**

Création de tables : contraintes

CHECK :

Si on veux définir une condition a respecter lors de l'ajout d'une ligne, on utilise la contrainte CHECK (condition) :

```
CREATE TABLE supplier
(
        supplier_id     NUMBER(10)      PRIMARY KEY,
        supplier_name   VARCHAR2(50)    NOT NULL,
        contact_id      NUMBER(10)      REFERENCES contact(ID)
        contact_name    VARCHAR2(50)
        contact_age     NUMBER(3)       CHECK (contact_age>18)
);
```

Création de tables : contraintes

CHECK :

Si on veux définir une condition a respecter lors de l'ajout d'une ligne, on utilise la contrainte CHECK (conditions) :

- **Comparaison** : >, <, =, <=, >=, !=, …
- **Operateurs** : +, *, /, -, ||, …
- **Fonctions** : SQRT, COS, ROUND, POW, ….
- **Pattern matching** : LIKE (%), …
- **Intervalle** : <Nom_d'attribut> BETWEEN <Borne_Inf> AND <Borne_Sup>
- **Liste** : <Nom_d'attribut> IN (<Valeur$_1$> , <Valeur$_2$>, … <Valeur$_n$>)
- **Éléments logiques** : AND, OR, NOT, …

Création de tables : contraintes

CHECK :

Si on veux définir une condition a respecter lors de l'ajout d'une ligne, on utilise la contrainte CHECK (conditions) :

- **Comparaison** : prix <= 200, type != 'inconnu', date_naiss < '01/01/1985',…
- **Operateurs** : (prix*prix)/(prix-50) < 100
- **Fonctions** : SQRT(prix) <= 50
- **Pattern matching** : categorie LIKE 'Cat_%', …
- **Intervalle** : prix BETWEEN 100 AND 200
- **Liste** : jours IN ('lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi')
- **Éléments logiques** : Prix>10 OR Prix=0

Création de tables : contraintes

Si les contraintes portent sur plusieurs attributs on utilise des **Contraintes de table**. La syntaxe reste la même.

Clé primaire multiple :
PRIMARY KEY (att1, att2, …)

Clé étrangère pointant sur une clé primaire multiple :
FOREIGN KEY (att1, att2,…) REFERENCES table2 (att21, att22, …) ON DELETE…

Attribut dont la combinaison doit être unique :
UNIQUE (att1, att2,…)

Condition entre plusieurs attributs :
CHECK (condition)

Création de tables : contraintes

Il est très important de nommer les contraintes, pour des raisons d'administration des tables et de compréhension des messages d'erreurs !

CONSTRAINT Nom_contrainte <contrainte>

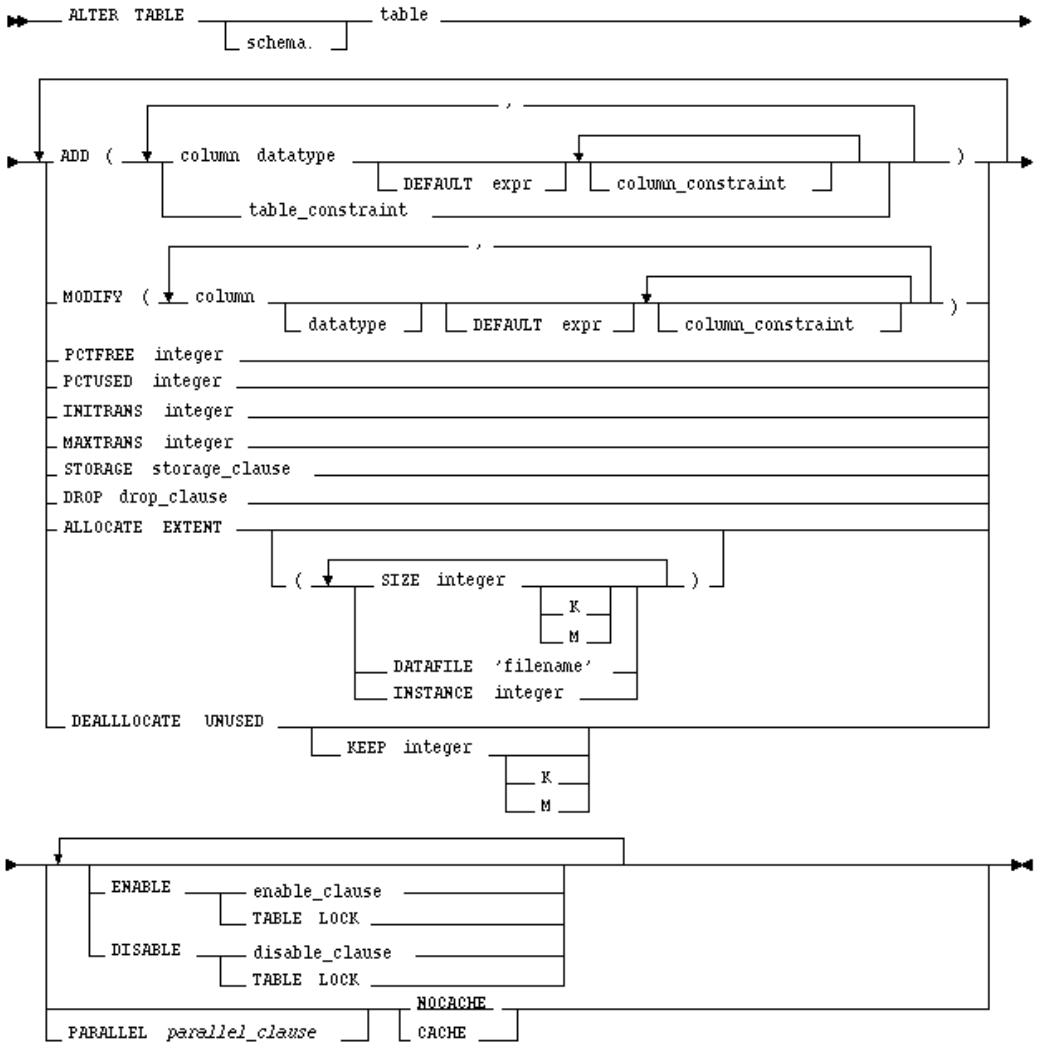
Notation utilisée concernant les noms des contraintes :

- **Contrainte clé primaire :** **pk_table**
- **Contrainte clé étrangère :** **fk_table1_colonne_table2**
- **Contrainte CHECK :** **ck_ table_colonne**
- **Contrainte NOT NULL :** **nn_ table_colonne**
- **Contrainte UNIQUE :** **unique_ table_colonne**

Création de tables : contraintes

```
CREATE TABLE jedi
(
   nom              VARCHAR2(50)      CONSTRAINT  nn_jedi_nom NOT NULL,
   prenom           VARCHAR2(50)      CONSTRAINT  nn_jedi_prenom NOT NULL,
   age              NUMBER(3),        CONSTRAINT  ck_jedi_age CHECK age>18,
   CONSTRAINT pk_jedi PRIMARY KEY (nom, prenom),
   CONSTRAINT ck_jedi_nom_prenom CHECK (nom != prenom),
);
```

Modification de tables



Modification de tables

La commande **ALTER TABLE** permet d'ajouter/de modifier une ou plusieurs **colonnes** dans une tables existante :

ALTER TABLE table **ADD/MODIFY**
(
attribut_1 [<type> <contraintes>],
...
attribut_n [<type> <contraintes>]
);

Remarques :

- Les valeurs de la nouvelle colonne sont des valeurs NULL.
- NOT NULL n'est pas autorisé dans ALTER TABLE.
- Si il y a des données dans la tables, elles doivent respecter les contraintes.

Modification de tables

La commande **ALTER TABLE** permet d'ajouter/de modifier une ou plusieurs **colonnes** dans une tables existante :

```
ALTER TABLE jedi ADD
(
        Adresse         VARCHAR2(50),
        Poids           NUMBER                  CHECK Poids>40
);
```

```
ALTER TABLE jedi MODIFY
(
        age             NUMBER(4)               CHECK age > 6
);
```

Modification de tables

La commande **ALTER TABLE** permet aussi d'ajouter une ou plusieurs **contraintes** dans une tables existante :

ALTER TABLE table **ADD** <contraintes>

ALTER TABLE apprentissage **ADD** CONSTRAINT fk_app_nomprenom_jedi FOREIGN KEY (master_nom, master_prenom) REFERENCES jedi (nom, prenom)

Modification de tables

La commande **ALTER TABLE** permet aussi d'ajouter ou de supprimer des **valeurs par défaut** dans une tables existante :

ALTER TABLE table **MODIFY** attribut **SET** valeur;

ALTER TABLE table **MODIFY** attribut **DROP DEFAULT**;

Modification de tables

La commande **ALTER TABLE** permet de supprimer des **colonnes** ou des contraintes dans une tables existante :

ALTER TABLE table **DROP** (<list col_names>);

ALTER TABLE table **DROP** PRIMARY KEY;

ALTER TABLE table **DROP** FOREIGN KEY fk_name;

ALTER TABLE table **DROP** CONSTRAINT constraint_name;

Les contraintes peuvent être désactivées et réactivées avec **DISABLE** et **ENABLE** à la place de **DROP**.

Destruction de tables

La commande **DROP TABLE** permet de supprimer une table inutilisée.

Lors de sa destruction, les données qu'elle contient sont perdues et ne peuvent être restaurées.

L'ensemble des objets (index, vues, etc…) associés à cette table sont également supprimés.

DROP TABLE <nom_de_la_table> [CASCADE CONSTRAINTS];

L'option CASCADE CONSTRAINTS permet de supprimer les clés étrangères des autres tables qui pointent sur la table à supprimer.

Insertion de lignes

L'insertion des ligne (**tuples)** dans la BD peut se faire en utilisant la commande ci-dessous :

INSERT INTO nom_table	(attribut$_1$, attribut$_2$, ... attribut$_n$)
VALUES	(valeur$_1$, valeur$_2$,....., valeur$_n$);

Insertion de lignes

Exemples :

INSERT INTO CENTRE (NUMC, NOMC, VILC, COUTINSC)
VALUES (‘103’, ’Pleine Forme’, ‘PARIS’, 400);

INSERT INTO CENTRE
VALUES (‘107’, ’Pleine Forme’, ‘PARIS’, 420);

INSERT INTO CENTRE (NUMC, NOMC, COUTINSC)
VALUES (‘111’, ’Le nouveau’, 400);

INSERT INTO PROPOSE (NUMC, SPORT)
VALUES (‘103’, ’FOOTBALL’);

Modification de lignes

La modification des **tuples** dans la BD peut se faire en utilisant la commande ci-dessous :

```
UPDATE   table
SET      attribut_1      =      valeur_1,
         attribut_2      =      valeur_2,
         …
         attribut_n      =      valeur_n
[WHERE   <condition>] ;
```

Modification de lignes

Exemples :

Augmenter le coût d' inscription, pour tous les centres, de 5% :

```
UPDATE CENTRE
SET    COUTINSC = COUTINSC * 105%;
```

Baisser le coût d' inscription, dans le centre 101, de 20% :

```
UPDATE CENTRE
SET    COUTINSC = COUTINSC * 80%
WHERE  NUMC = '101';
```

Suppression de lignes

La suppression des **tuples** dans la BD peut se faire en utilisant la commande ci-dessous :

```
DELETE FROM     table
[WHERE          <condition>] ;
```

Suppression de lignes

Exemples :

Supprimer de la BD les coordonnées de l'acteur de numéro 003

```
DELETE FROM     ACTEUR
WHERE           NUMA = '003';
```

Supprimer de la BD tous les acteurs

```
DELETE FROM     ACTEUR;
```

Différence avec DROP TABLE ?

Partie IV

-

Interrogation d’une table

SELECT

La commande SELECT constitue, à elle seule, le langage permettant d'interroger une base de données.

Elle permet de :

- Sélectionner certaines colonnes d'une table (projection) ;
- Sélectionner certaines lignes d'une table en fonction de leur contenu (restricton) ;
- Combiner des informations venant de plusieurs tables (jointure, union, intersection, différence et division) ;
- Combiner entre elles ces différentes opérations.

SELECT

La commande SELECT constitue, à elle seule, le langage permettant d'interroger une base de données.

Une requête (*i.e.* une interrogation) est une combinaison d'opérations portant sur des tables (relations) et dont le résultat est lui-même une table dont l'existence est éphémère (le temps de la requête).

Syntaxe simplifiée de SELECT

```
SELECT [ ALL | DISTINCT ] { * | attribut [, ...] }
FROM nom_table [, ...]
[ WHERE condition ] ;
```

Syntaxe simplifiée de SELECT

```
SELECT [ ALL | DISTINCT ] { * | attribut [, ...] }
FROM nom_table [, ...]
[ WHERE condition ] ;
```

- La clause **SELECT** permet de spécifier les attributs que l'on désire voir apparaître dans le résultat de la requête
- La clause **FROM** spécifie les tables sur lesquelles porte la requête
- La clause **WHERE** (facultative), énonce une condition que doivent respecter les lignes (n-uplets) sélectionnés

Syntaxe simplifiée de SELECT

La clause **SELECT** permet de réaliser la projection, la clause **FROM** le produit cartésien et la clause **WHERE** la sélection.

Exemple, sélectionner toutes les lignes de la table *film* :

```
SELECT * FROM film ;
```

Syntaxe générale de SELECT

```
SELECT [ ALL | DISTINCT ] { * | expression [ AS nom_affiché ] } [, ...]
        FROM nom_table [ [ AS ] alias ] [, ...]
        [ WHERE condition ]
        [ GROUP BY expression [, ...] ]
        [ HAVING condition [, ...] ]
        [ {UNION | INTERSECT | EXCEPT [ALL]} requête ]
        [ ORDER BY expression [ ASC | DESC ] [, ...] ]
;
```

Syntaxe générale de SELECT

SELECT est composé de 7 clauses dont 5 sont optionnelles :

- **SELECT :** permet de spécifier les attributs que l'on désire voir apparaître dans le résultat de la requête
- **FROM :** spécifie les tables sur lesquelles porte la requête
- **WHERE :** permet de filtrer les lignes en imposant une condition à remplir
- **GROUP BY :** permet de définir des groupes (*i.e.* sous-ensemble).
- **HAVING :** permet de spécifier un filtre (condition de regroupement)
- **UNION, INTERSECT et EXCEPT :** permet d'effectuer des opérations ensemblistes entre plusieurs résultats de requête (*i.e.* entre plusieurs **SELECT**)
- **ORDER BY :** Cette clause permet de trier les lignes du résultat.

L'opérateur de projection

La projection est l'opération qui permet de sélectionner certaines colonnes d'une table, en *éliminant les doublons*.

Elle se note, en algèbre relationnelle :

$$\Pi_{(Colonne\_1, \ldots, Colonne\_n)}(Table)$$

L'opérateur de projection

$$\Pi_{(Colonne\_1, \ldots, Colonne\_n)}(Table)$$

- Cet opérateur ne porte que sur 1 table
- Il permet de ne retenir que certains attributs (colonnes)
- On obtient toutes les lignes de la table **à l'exception des doublons**

Projection en SQL

Sélectionner toutes les lignes de la table :

```
SELECT * FROM nom_table;
```

Exemple :

```
SELECT * FROM Champignons;
```

Espèce	Catégorie	Conditionnement	Nombre
Rosé des prés	Conserve	Bocal	12
Rosé des prés	Sec	Verrine	10
Coulemelle	Frais	Boîte	6
Rosé des prés	Sec	Sachet plastique	2

Projection en SQL

Sélectionner certaines lignes de la table :

SELECT Col1, Col2, … FROM nom_table;

Exemple :

SELECT Espèce, Catégorie FROM Champignons;

Espèce	Catégorie
Rosé des prés	Conserve
Rosé des prés	Sec
Coulemelle	Frais
Rosé des prés	Sec

Projection en SQL

Sélectionner certaines lignes de la table, sans doublons :

SELECT DISTINCT Col1, Col2, … FROM nom_table;

Exemple :

SELECT DISTINCT Espèce, Catégorie FROM Champignons;

Espèce	Catégorie
Rosé des prés	Conserve
Rosé des prés	Sec
Coulemelle	Frais

Projection avec opérations

Il est possible de faire des opérations sur les colonnes :

SELECT expression FROM nom_table;

Exemple :

SELECT DISTINCT Espèce || ' : ' || Catégorie FROM Champignons;

Espèce||':'||Catégorie
Rosé des prés : Conserve
Rosé des prés : Sec
Coulemelle : Frais

Projection avec opérations

Il est possible de nommer les colonnes sélectionnées à afficher :

SELECT expression AS nom_affiché FROM nom_table;

Exemple :

SELECT Conditionnement, Nombre AS Nombre_Total, Nombre/2+1 AS Nombre_A_Vendre FROM Champignons;

Conditionnement	Nombre_Total	Nombre_A_Vendre
Bocal	12	7
Verrine	10	6
Boîte	6	4
Sachet plastique	2	2

Projection avec opérations

Les opérations portent sur une ou plusieurs colonnes

- **Operateurs numériques** : +, *, /, -, (,), …
- **Operateurs de concaténation** : ||
- **Fonctions** :
  - SQRT, POW : Racine carrée et puissances
  - COS, SIN, etc. : Trigonométrie
  - ROUND : Arrondis
  - …

L'opérateur de restriction

La restriction (ou sélection) est l'opération qui permet de sélectionner certaines lignes d'une table qui vérifient une condition.

Elle se note, en algèbre relationnelle :

$$\sigma(_{condition})(Table)$$

L'opérateur de restriction

$$\sigma(_{condition})(Table)$$

- Cet opérateur ne porte que sur 1 table
- Il permet de ne retenir que certaines lignes (n-uplets)
- Les lignes retenues doivent vérifier la *condition*

Restriction en SQL

Syntaxe générale:

```
SELECT  * FROM nom_table WHERE condition;
```

La condition peut porter sur une ou plusieurs colonnes

Définition des Conditions

Syntaxe générale:

```
SELECT  * FROM nom_table WHERE condition;
```

- **Une condition est un prédicat logique**
- **Elle est soit vraie, soit fausse**
- **Seules les lignes qui vérifient la condition sont sélectionnées**

Définition des Conditions

De nombreux operateurs et fonctions peuvent être utilisés pour définir une condition, on peut distinguer trois types :

- **Comparaison** entre attributs (colonnes) et/ou valeurs, au moins une par condition, doit être vraie ou fausse pour chaque ligne (n-uplet)
- **Transformation** des attributs avant comparaison
- **Combinaison** logique de comparaisons

Comparaisons

Operateurs arithmétiques de comparaison :

- **A > B** : vrai si A plus grand que B
- **A < B** : vrai si A plus petit que B
- **A = B** : vrai si A et B sont identiques
- **A <= B** : vrai si A plus grand ou identique à B
- **A >= B** : vrai si A plus petit ou identique à B
- **A != B** : vrai si A est B ne sont pas identiques

Ces operateurs peuvent être utilisés pour comparer des valeurs numériques, des dates ou des chaines de caractères

Comparaisons

Table Champignons

Espèce	Catégorie	Conditionnement	Nombre
Rosé des prés	Conserve	Bocal	12
Rosé des prés	Sec	Verrine	10
Coulemelle	Frais	Boîte	6
Rosé des prés	Sec	Sachet plastique	2

```
SELECT * FROM Champignon WHERE Espèce = 'Rosé des prés';
```

Espèce	Catégorie	Conditionnement	Nombre
Rosé des prés	Conserve	Bocal	12
Rosé des prés	Sec	Verrine	10
Rosé des prés	Sec	Sachet plastique	2

Comparaisons

L'opérateur LIKE permet d'effectuer une comparaison entre une *chaine de caractère* et un modèle particulier :

- **LIKE '%a'** : se termine par un « a »
- **LIKE 'a%'** : commence par un « a »
- **LIKE '%a%'** : contient le caractère « a » (n'importe où)
- **LIKE 'pa%on'** : commence par « pa », se termine par « on »
- **LIKE 'a_c'** : contient une seule lettre entre « a » et « c »

- « % » remplace tous les autres caractères
- « _ » remplace un caractère uniquement
- LIKE est sensible à la casse

Comparaisons

Table Clients

id	nom	ville
1	Léon	Lyon
2	Odette	Nice
3	Vivien	Nantes
4	Etienne	Lille

```
SELECT * FROM Clients WHERE ville LIKE 'N%';
```

id	nom	ville
2	Odette	Nice
3	Vivien	Nantes

Comparaisons

L'opérateur BETWEEN permet de comparer une valeur a un intervalle, par exemple :

- Prix BETWEEN 10 AND 100
- Date BETWEEN '01/11/2001' AND '01/11/2013'
- Nom BETWEEN 'A' AND 'G'

Comparaisons

L'opérateur BETWEEN permet de comparer une valeur a un intervalle

- Les bornes sont incluses sous ORACLE
- Les valeurs peuvent être numériques, des dates ou des chaines de caractères
- Pour les chaines de caractères : ordre alphabétique

Comparaisons

Table utilisateurs

id	nom	date_inscription
1	Maurice	2012-03-02
2	Simon	2012-03-05
3	Chloé	2012-04-14
4	Marie	2012-04-15
5	Clémentine	2012-04-26

```
SELECT * FROM utilisateurs
WHERE date_inscription BETWEEN '2012-04-01' AND '2012-04-20';
```

id	nom	date_inscription
3	Chloé	2012-04-14
4	Marie	2012-04-15

Comparaisons

L'opérateur IN permet de comparer une valeur a une liste de valeurs possibles, par exemple :

- Année IN (1, 2, 3)
- Date IN ('01/11/2001' , '01/11/2013')
- Jours IN ('Lundi' , 'Mardi', 'Mercredi', 'Jeudi')

Comparaisons

L'opérateur IN permet de comparer une valeur a une liste de valeurs possibles.

- est valide pour les valeur numériques, les dates et les chaines de caractères
- est sensible à la casse

Comparaisons

Table addresses

id	id_utilisateur	addr_rue	addr_code_postal	addr_ville
1	23	35 Rue Madeleine Pelletier	25250	Bournois
2	43	21 Rue du Moulin Collet	75006	Paris
3	65	28 Avenue de Cornouaille	27220	Mousseaux-Neuville
4	67	41 Rue Marcel de la Provoté	76430	Graimbouville
5	68	18 Avenue de Navarre	75009	Paris

```
SELECT * FROM addresses WHERE addr_ville IN ('Paris', Graimbouville');
```

id	id_utilisateur	addr_rue	addr_code_postal	addr_ville
2	43	21 Rue du Moulin Collet	75006	Paris
4	67	41 Rue Marcel de la Provoté	76430	Graimbouville
5	68	18 Avenue de Navarre	75009	Paris

Comparaisons

L'opérateur IS NULL permet de comparer une valeur à la valeur NULL :

- Année IS NULL
- Date IS NULL
- Jours IS NULL

Comparaisons

Table utilisateurs

id	nom	date_inscription	livraison_id	facturation_id
23	Grégoire	2013-02-12	12	12
24	Sarah	2013-02-17	NULL	10
25	Anne	2013-02-21	13	14
26	Frédérique	2013-03-02	NULL	NULL

```
SELECT * FROM utilisateurs WHERE livraison_id IS NULL;
```

id	nom	date_inscription	livraison_id	facturation_id
24	Sarah	2013-02-17	NULL	10
26	Frédérique	2013-03-02	NULL	NULL

Transformation

- **Operateurs de transformation de valeurs numériques**:
  - + : addition
  - - : soustraction
  - * : multiplication
  - / : division

```
SELECT * FROM produits WHERE prix_vente > prix_achat  + prix_achat * 20/100;
```

- **Operateurs de transformation de chaines de caractères**: ||

```
SELECT * FROM clients WHERE nom||prenom LIKE '%David%';
```

Transformation

Fonctions de transformation de valeurs numériques :

- SQRT(N) : racine carrée de N
- ABS (N) : valeur absolue de N
- COS(N), SIN(N), TAN(N), … : fonctions trigonométriques, N en radiant
- ROUND (N), FLOOR (N), CEILING (N) : arrondi de N, entier inf, entier sup
- POW(N, P), EXP (N) : A puissance P, exponentiel de N
- …

```
SELECT * FROM produits WHERE ABS (gain) > SQRT (prix_achat );
```

Transformation

Fonctions de transformation de chaines de caractères:

- LOWER(C) : met C en minuscule
- UPPER (C) : met C en majuscule
- SOUNDEX (C) : transforme C en phonétique
- LENGTH (C) : calcule le nombre de caractères de C
- TO_CHAR (N) : transforme le nombre N en caractères
- …

```
SELECT * FROM clients WHERE LOWER (Nom) = 'dupuis';
```

Combinaison

On peut combiner plusieurs conditions à l'aide d'éléments logiques :

- **Condition1 AND Condition2** : vrai si les deux conditions sont vraies
- **Condition1 OR Condition2** : vrai si au moins une des conditions est vraie
- **NOT Condition1** : vrai si la condition est fausse
- **( )** : les parenthèses permettent une infinité de combinaisons

NOT peut aussi se mettre devant un operateur de comparaison :

- Colonne NOT IN (liste) ⇔ NOT (Colonne IN (liste))
- Colonne NOT LIKE expr ⇔ NOT (Colonne LIKE expr)
- Colonne IS NOT NULL ⇔ NOT (Colonne IS NULL)

Combinaison

Table produits

id	nom	catégorie	stock	prix
1	ordinateur	informatique	5	950
2	clavier	informatique	32	35
3	souris	informatique	16	30
4	crayon	fourniture	147	2

```
SELECT * FROM produit WHERE ( catégorie = 'informatique' AND stock < 20 )
OR  catégorie = 'fourniture';
```

id	nom	catégorie	stock	prix
1	ordinateur	informatique	5	950
3	souris	informatique	16	30
4	crayon	fourniture	147	2

Tri de lignes

Il est possible de trier les lignes affichés par la commande SELECT. Il faut pour cela utiliser la clause ORDER BY.

```
SELECT liste_colonnes_select
        FROM nom_table
        [ WHERE condition ]
        ORDER BY liste_colonnes_order [ DESC ]
;
```

Tri de lignes

Il est possible de trier les lignes affichés par la commande SELECT. Il faut pour cela utiliser la clause ORDER BY.

- Il est possible de trier sur une ou plusieurs colonnes, par ordre de priorité
- Par défaut, le tri se fait par ordre croissant, mais il est possible de trier par ordre décroissant (option DESC)
- Tous les types d'attributs peuvent être triés

Tri de lignes

Table utilisateur

id	nom	prenom	date_inscription	tarif_total
1	Durand	Maurice	2012-02-05	145
2	Dupond	Fabrice	2012-02-07	65
3	Durand	Fabienne	2012-02-13	90
4	Dubois	Chloé	2012-02-16	98
5	Dubois	Simon	2012-02-23	27

SELECT * FROM utilisateur ORDER BY nom, date_inscription DESC ;

id	nom	prenom	date_inscription	tarif_total
5	Dubois	Simon	2012-02-23	27
4	Dubois	Chloé	2012-02-16	98
2	Dupond	Fabrice	2012-02-07	65
3	Durand	Fabienne	2012-02-13	90
1	Durand	Maurice	2012-02-05	145

Tri de lignes

Remarque : Le nom de la colonne peut être remplacé par un chiffre qui correspond à son numéro de colonne dans la projection. C'est utile en particulier pour les colonnes issues d'un calcul, mais il existe toujours d'autres solutions.

Exemple:

```
SELECT nom_produit, prix_vente-prix_achat
FROM produits ORDER BY 2 DESC ;

SELECT nom_produit, prix_vente-prix_achat
FROM produits ORDER BY prix_vente-prix_achat DESC ;

SELECT nom_produit, prix_vente-prix_achat AS benefice
FROM produits ORDER BY benefice DESC ;
```

Combiner restriction et projection

En SQL il est très facile de combiner projection et restriction.

En algèbre relationnelle :

$$\Pi(_{Col\_1, \ldots, Col\_n})(\sigma(_{condition})(Table))$$

En SQL:

```
SELECT DISTINCT Col_1, …, Col_n FROM Table
WHERE condition;
```

Exemple

Table produits

id	nom	catégorie	stock	prix
1	ordinateur	informatique	5	950
2	clavier	informatique	32	35
3	souris	informatique	16	30
4	crayon	fourniture	147	2

SELECT nom, catégorie, prix FROM produits WHERE stock <= 30 ;

nom	catégorie	prix
ordinateur	informatique	950
souris	informatique	30

Exemple

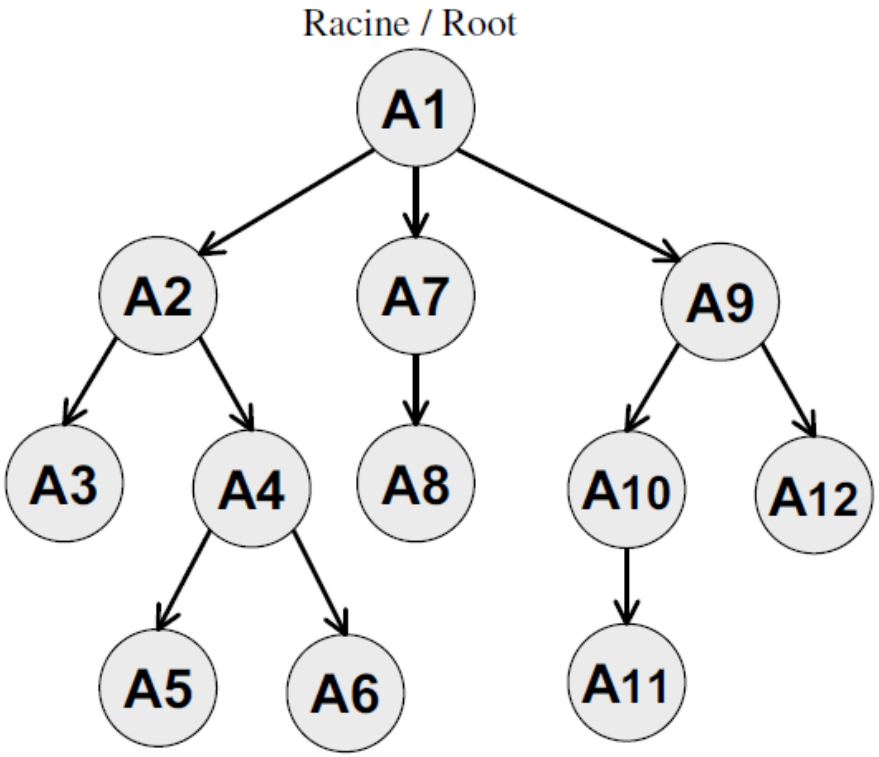
Table utilisateur

id	nom	prenom	date_inscription	tarif_total
1	Durand	Maurice	2012-02-05	145
2	Dupond	Fabrice	2012-02-07	65
3	Durand	Fabienne	2012-02-13	90
4	Dubois	Chloé	2012-02-16	98
5	Dubois	Simon	2012-02-23	27

SELECT nom, prenom FROM utilisateurs ORDER BY tarif_total ;

nom	prenom
Dubois	Simon
Dupond	Fabrice
Durand	Fabienne
Dubois	Chloé
Durand	Maurice

Arbres



Lorsque les données sont structurées en arbres ou graphes, il existe des outil SQL pour simplifier les requêtes.

Arbres : Syntaxe

```
Select colonnes [, LEVEL] FROM  table
START WITH condition
CONNECT BY expression avec PRIOR ;
```

- **START WITH** : indique le nœud à partir duquel l'arbre est parcouru.
- **CONNECT BY PRIOR** : règle de connexion entre les nœuds
- **LEVEL** : profondeur du nœud par rapport à la racine

PRIOR indique si l'arbre doit être parcouru de la racine vers les feuilles ou des feuilles vers la racine. Il précède l'enfant si on parcourt l'arbre de la racine vers les feuilles. Il précède le parent si on parcourt l'arbre des feuilles vers la racine.

Arbres : Syntaxe

Select colonnes [, LEVEL] FROM table
START WITH condition
CONNECT BY expression avec PRIOR ;

Remarques :

- Une restriction dans une clause supprime les individus, mais n'arrête pas la construction de l'arbre.
- Un select avec gestion d'arborescence ne peut être une jointure.
- ORDER BY est exécuté après le tri arborescent.

Arbres : Exemple

Nous allons nous intéresser à une structure de données permettant de créer un Forum de discussion, matérialisé par la table suivante :

messages (id, id_parent, titre, auteur, heure, texte);

ID	ID_P	TITRE	AUTEUR	HEURE
1		Combien d'oeufs dans la pate à crêpes ?	John	16:58:20
2		Sondage : votre marque de lait préférée	John	16:58:37
3	1	Re : Combien d'oeufs dans la pate à crêpes ?	Martine	16:58:51
4	3	C'est sûrement ça	John	16:59:00
5	2	Re : Sondage : votre marque de lait préférée	Fifi	16:59:08
6	2	Re : Sondage : votre marque de lait préférée	Petrus	16:59:14
7	4	Re : C'est sûrement ça	Martine	16:59:20
8	2	Re : Sondage : votre marque de lait préférée	Fifi	16:59:26
9	2	Re : Sondage : votre marque de lait préférée	Enrico	16:59:31
10	8	Re : Sondage : votre marque de lait préférée	John	16:59:45
11		Qui a vu le match de volley hier soir ?	Petrus	16:59:51
12	10	Re : Sondage : votre marque de lait préférée	Fifi	16:59:59
13	3	Re : Combien d'oeufs dans la pate à crêpes ?	Fifi	17:00:03

Arbres : Exemple

Notre but est donc d'obtenir une hiérarchie de la forme :

```
ID=1
   ID=3
      ID=4
         ID=7
      ID=13
ID=2
   ID=5
   ID=6
   ID=8
      ID=10
         ID=12
   ID=9
ID=11
```

Le lien entre un message et son parent est bien sûr fait avec les colonnes ID et ID_PARENT.

Arbres : Exemple

```
SELECT id, id_parent, titre FROM messages
START WITH id_parent IS NULL CONNECT BY id_parent = PRIOR id;
```

ID	ID_PARENT	TITRE
1		Combien d'oeufs dans la pate à crêpes ?
3	1	Re : Combien d'oeufs dans la pate à crêpes ?
4	3	C'est sûrement ça
7	4	Re : C'est sûrement ça
13	3	Re : Combien d'oeufs dans la pate à crêpes ?
2		Sondage : votre marque de lait préférée
5	2	Re : Sondage : votre marque de lait préférée
6	2	Re : Sondage : votre marque de lait préférée
8	2	Re : Sondage : votre marque de lait préférée
10	8	Re : Sondage : votre marque de lait préférée
12	10	Re : Sondage : votre marque de lait préférée
9	2	Re : Sondage : votre marque de lait préférée
11		Qui a vu le match de volley hier soir ?

Notre objectif est atteint : nous avons les messages dans le bon ordre. Une requête hiérarchique permet donc de trier des données en utilisant des notions d'arborescence.

Arbres : Exemple

Afin de mieux voir la profondeur de hiérarchie, nous allons utiliser la pseudo-colonne LEVEL, qui permet de savoir à quel niveau hiérarchique on se trouve :

```
SELECT level, titre, auteur FROM messages
START WITH id_parent IS NULL CONNECT BY id_parent = PRIOR id;
```

LEVEL	TITRE	AUTEUR
1	Combien d'oeufs dans la pate à crêpes ?	John
2	Re : Combien d'oeufs dans la pate à crêpes ?	Martine
3	C'est sûrement ça	John
4	Re : C'est sûrement ça	Martine
3	Re : Combien d'oeufs dans la pate à crêpes ?	Fifi
1	Sondage : votre marque de lait préférée	John
2	Re : Sondage : votre marque de lait préférée	Fifi
2	Re : Sondage : votre marque de lait préférée	Petrus
2	Re : Sondage : votre marque de lait préférée	Fifi
3	Re : Sondage : votre marque de lait préférée	John
4	Re : Sondage : votre marque de lait préférée	Fifi
2	Re : Sondage : votre marque de lait préférée	Enrico
1	Qui a vu le match de volley hier soir ?	Petrus

Arbres : Exemple

Pour plus de clarté, nous allons faire précéder chaque titre d'un nombre d'espaces égal au level qui lui correspond. Cela génèrera une indentation du plus bel effet !

```
SELECT id, id_parent, RPAD(' ', level-1) || titre AS titre , auteur FROM messages
START WITH id_parent IS NULL CONNECT BY id_parent = PRIOR id;
```

ID	ID_PARENT	TITRE	AUTEUR
1		Combien d'oeufs dans la pate à crêpes ?	John
3	1	Re : Combien d'oeufs dans la pate à crêpes ?	Martine
4	3	C'est sûrement ça	John
7	4	Re : C'est sûrement ça	Martine
13	3	Re : Combien d'oeufs dans la pate à crêpes ?	Fifi
2		Sondage : votre marque de lait préférée	John
5	2	Re : Sondage : votre marque de lait préférée	Fifi
6	2	Re : Sondage : votre marque de lait préférée	Petrus
8	2	Re : Sondage : votre marque de lait préférée	Fifi
10	8	Re : Sondage : votre marque de lait préférée	John
12	10	Re : Sondage : votre marque de lait préférée	Fifi
9	2	Re : Sondage : votre marque de lait préférée	Enrico
11		Qui a vu le match de volley hier soir ?	Petrus

Arbres : Exemple

La clause START WITH permet d'indiquer la condition qui détermine quels enregistrements sont au niveau 1. Si on n'avait voulu lister que les fils de discussion initiés par John, avec leurs réponses, voici la requête qu'il aurait fallu exécuter :

```
SELECT id, id_parent, RPAD(' ', level-1) || titre AS titre , auteur FROM messages
START WITH id_parent IS NULL AND auteur = 'John' CONNECT BY id_parent = PRIOR id;
```

ID	ID_PARENT	TITRE	AUTEUR
1		Combien d'oeufs dans la pate à crêpes ?	John
3	1	Re : Combien d'oeufs dans la pate à crêpes ?	Martine
4	3	C'est sûrement ça	John
7	4	Re : C'est sûrement ça	Martine
13	3	Re : Combien d'oeufs dans la pate à crêpes ?	Fifi
2		Sondage : votre marque de lait préférée	John
5	2	Re : Sondage : votre marque de lait préférée	Fifi
6	2	Re : Sondage : votre marque de lait préférée	Petrus
8	2	Re : Sondage : votre marque de lait préférée	Fifi
10	8	Re : Sondage : votre marque de lait préférée	John
12	10	Re : Sondage : votre marque de lait préférée	Fifi
9	2	Re : Sondage : votre marque de lait préférée	Enrico

Arbres : Exemple

```
SELECT id, id_parent, RPAD(' ', level-1) || titre AS titre , auteur
FROM messages
START WITH id_parent IS NULL AND auteur = 'John'
CONNECT BY id_parent = PRIOR id;
```

- Le 13ème message n'a pas été sélectionné car il ne fait pas partie d'un fil de discussion initié par John.
- Les requêtes hiérarchiques sont donc non seulement un moyen de trier les enregistrements, mais aussi un moyen de sélectionner des enregistrements (même sans utilisation de la clause WHERE).

L'agrégation

Il est possible de faire des calculs de type agrégat sur des ensembles de ligne (des groupes) puis de les afficher avec SELECT.

Exemple de fonctions d'agrégat :

- Moyenne
- Somme
- Minimum
- Maximum
- Comptage

Fonctions d'agrégat en SQL

- **SUM(attribut)** : total des valeurs (numériques) d'un attribut
- **AVG(attribut)** : moyenne des valeurs (num) d'un attribut
- **MIN(attribut)** : plus petite valeur (num, date) d'un attribut
- **MAX(attribut)** : plus grande valeur (num, date) d'un attribut
- **COUNT(*)** : nombre de lignes
- **COUNT(DISTINCT attribut)** : nombre de valeurs différentes de l'attribut

Remarque : les valeurs NULL sont ignorées.

Utilisation des fonctions d'agrégat

La fonction d'agrégat s'utilise dans la clause SELECT à la place d'un nom de colonne.

Exemples:

- Prix moyen des articles :

  SELECT AVG(prix) FROM articles ;

- Note maximale :

  SELECT MAX(note) FROM resultats ;

- Nombre d'étudiants :

  SELECT COUNT(*) FROM etudiants ;

Le regroupement de lignes

Il est possible de calculer l'agrégation non pas sur toutes les lignes mais par catégories.

Syntaxe simple:

```
SELECT fonction_agregation
        FROM nom_table
        GROUP BY liste_attributs;
```

Les lignes sont regroupées si elles ont les mêmes valeurs sur le ou les attributs de la clause GROUP BY.

Le regroupement de lignes

Table Commandes

ID_client	ID_fournisseur	Prix
C1	F1	25
C1	F2	35
C2	F1	64
C2	F1	120
C2	F3	26

```
SELECT ID_client, ID_fournisseur, MAX(Prix) AS Maxi FROM Commandes
GROUP BY ID_client, ID_fournisseur ;
```

ID_client	ID_fournisseur	Maxi
C1	F1	25
C1	F2	35
C2	F1	120
C2	F3	26

Condition sur les groupes

Il est aussi possible de ne sélectionner que certains des groupes, avec la clause HAVING.

Syntaxe simple:

```
SELECT fonction_agregation
        FROM nom_table
        GROUP BY liste_attributs
        HAVING conditions ;
```

La condition de HAVING porte sur les groupes à afficher, alors que celle de WHERE porte sur les lignes à prendre en compte lors du calcul d'agrégation. Les conditions de HAVING s'écrivent de la même façon que dans WHERE.

Condition sur les groupes

Il est aussi possible de ne sélectionner que certains des groupes, avec la clause HAVING.

Exemple : Liste des UE pour lesquelles les étudiants de Master ont obtenu une moyenne générale > 10

```
SELECT nom_UE FROM resultats
        WHERE niveau = 'Master'
        GROUP BY nom_UE
        HAVING AVG(note) > 10 ;
```

Condition sur les groupes

Table Commandes

Commande	CodeProduit	Quantité	Prix
96008	A10	10	83
96008	B20	35	32
96009	A10	20	83
96010	A15	4	110
96010	B20	55	32

```
SELECT Commande, SUM(Quantité*Prix) AS Montant FROM Commandes
GROUP BY Commande HAVING Montant > 1700
ORDER BY Montant DESC ;
```

Commande	Montant
96010	2200
96008	1950

Partie V

-

Interrogation de plusieurs tables

Introduction

En algèbre relationnelle, il existe un certain nombre d'operateurs permettant de fusionner les informations de deux tables

Jointures (attributs différents) **:**

- Produit cartésien — $R = R1 \times R2$
- Jointure (cas général) — $R = R1 \rhd\lhd_E R2$
- Équijointure — $R = R1 \rhd\lhd_{A1,A2} R2$
- Jointure naturelle — $R = R1 \rhd\lhd R2$

Opération ensemblistes (mêmes attributs) **:**

- Union — $R = R1 \cup R2$
- Intersection — $R = R1 \cap R2$
- Différence — $R = R1 - R2$
- Division — $R = R1 \div R2$

Produits cartésien

Le produit cartésien construit une relation regroupant exclusivement toutes les possibilités de combinaison des occurrences de deux relations.

$$R = R1 \times R2$$

Produits cartésien

Le produit cartésien construit une relation regroupant exclusivement toutes les possibilités de combinaison des occurrences de deux relations.

Exemple :

Relation *Cadeau*

Article	Prix
livre	45
poupée	25
montre	87

Relation *Amie*

Nom	Prénom
Fourt	Lisa
Juny	Carole

Relation *R* = *Amie* × *Cadeau*

Nom	Prénom	Article	Prix
Fourt	Lisa	livre	45
Fourt	Lisa	poupée	25
Fourt	Lisa	montre	87
Juny	Carole	livre	45
Juny	Carole	poupée	25
Juny	Carole	montre	87

Produits cartésien

Le produit cartésien construit une relation regroupant exclusivement toutes les possibilités de combinaison des occurrences de deux relations.

- Le résultat du produit cartésien est une nouvelle relation qui a tous les attributs de R_1 et tous ceux de R_2.
- Le nombre d'occurrences de la relation qui résulte du produit cartésien est le nombre d'occurrences de R_1 multiplié par le nombre d'occurrences de R_2.

Jointure

La jointure est une opération qui construit une relation regroupant exclusivement toutes les possibilités de combinaison des occurrences de deux relations qui satisfont une expression logique E.

$$R = R1 \triangleright\!\!\triangleleft_E R2$$

Jointure

La jointure est une opération qui construit une relation regroupant exclusivement toutes les possibilités de combinaison des occurrences de deux relations qui satisfont une expression logique E.

Exemple :

Relation *Famille*

Nom	Prénom	Age
Fourt	Lisa	6
Juny	Carole	42
Fidus	Laure	16

Relation *Cadeau*

AgeC	Article	Prix
99	livre	30
6	poupée	60
20	baladeur	45

Relation *R= Famille* $\triangleright\!\triangleleft_{(Age \leq AgeC)}$ *Cadeau*

Nom	Prénom	Age	AgeC	Article	Prix
Fourt	Lisa	6	99	livre	30
Fourt	Lisa	6	20	poupée	60
Fourt	Lisa	6	20	baladeur	45
Juny	Carole	42	99	livre	30
Fidus	Laure	16	99	poupée	60
Fidus	Laure	16	20	baladeur	45

Jointure

La jointure est une opération qui construit une relation regroupant exclusivement toutes les possibilités de combinaison des occurrences de deux relations qui satisfont une expression logique E.

En fait, la jointure n'est rien d'autre qu'un produit cartésien suivi d'une sélection :

$$R_1 \triangleright\!\!\triangleleft_E R_2 = \sigma_E (R_1 \times R_2)$$

Equijointure

Une équijointure est une jointure dans laquelle l'expression E est un test d'égalité entre un attribut A_1 de la relation R_1 et un attribut A_2 de la relation R_2.

$$R = R1 \triangleright\!\!\triangleleft_{A1,A2} R2$$

Equijointure

Une équijointure est une jointure dans laquelle l'expression E est un test d'égalité entre un attribut A_1 de la relation R_1 et un attribut A_2 de la relation R_2.

Exemple :

Relation *Famille*

Nom	Prénom	AgeF
Fourt	Lisa	6
Juny	Carole	40
Fidus	Laure	20
Choupy	Emma	6

Relation *Cadeau*

AgeC	Article	Prix
40	livre	45
6	poupée	25
20	montre	87

Relation *R = Famille* $\triangleright\!\triangleleft_{AgeF,\ AgeC}$ *Cadeau*

Nom	Prénom	AgeC	AgeF	Article	Prix
Fourt	Lisa	6	6	poupée	25
Juny	Carole	40	40	livre	45
Fidus	Laure	20	20	montre	87
Choupy	Emma	6	6	poupée	25

Equijointure

Une équijointure est une jointure dans laquelle l'expression E est un test d'égalité entre un attribut A_1 de la relation R_1 et un attribut A_2 de la relation R_2.

Relation *R* = *Famille* $\triangleright\!\triangleleft_{AgeF,\ AgeC}$ *Cadeau*

Nom	Prénom	AgeC	AgeF	Article	Prix
Fourt	Lisa	6	6	poupée	25
Juny	Carole	40	40	livre	45
Fidus	Laure	20	20	montre	87
Choupy	Emma	6	6	poupée	25

Dans une équijointure les deux attributs A_1 et A_2 apparaissent !

Jointure naturelle

Une jointure naturelle est une jointure dans laquelle l'expression logique E est un test d'égalité entre les attributs qui portent le même nom dans les deux relations.

$$R = R_1 \bowtie R_2$$

Ou en précisant les attributs communs à utiliser :

$$R = R_1 \bowtie_{A_1,\dots A_n} R_2$$

Jointure naturelle

Une jointure naturelle est une jointure dans laquelle l'expression logique E est un test d'égalité entre les attributs qui portent le même nom dans les deux relations.

Exemple :

Relation *Famille*

Nom	Prénom	Age
Fourt	Lisa	6
Juny	Carole	40
Fidus	Laure	20
Choupy	Emma	6

Relation *Cadeau*

Age	Article	Prix
40	livre	45
6	poupée	25
20	montre	87

Relation *R* = *Famille* ⋈ *Cadeau*

Nom	Prénom	Age	Article	Prix
Fourt	Lisa	6	poupée	25
Juny	Carole	40	livre	45
Fidus	Laure	20	montre	87
Choupy	Emma	6	poupée	25

Jointure naturelle

Une jointure naturelle est une jointure dans laquelle l'expression logique E est un test d'égalité entre les attributs qui portent le même nom dans les deux relations.

- Dans la relation construite, ces attributs ne sont pas dupliqués, mais fusionnés en une seule colonne par couple d'attributs.

- Généralement, R_1 et R_2 n'ont qu'un attribut en commun. Dans ce cas, une jointure naturelle est équivalente à une *équijointure* dans laquelle l'attribut de R_1 et celui de R_2 sont justement les deux attributs qui portent le même nom.

- Il vaut mieux écrire $R_1 \triangleright\!\triangleleft_{A_1} R_2$ que $R_1 \triangleright\!\triangleleft R_2$. En effet, si R_1 et R_2possèdent deux attributs portant un nom commun, A_1 et A_2, $R_1 \triangleright\!\triangleleft_{A_1} R_2$ est bien une jointure naturelle sur l'attribut A_1, mais $R_1 \triangleright\!\triangleleft R_2$ est une jointure naturelle sur le couple d'attributs (A_1, A_2), ce qui produit un résultat très différent !

Jointures en SQL

Sans compter l'opérateur **CROSS JOIN**, voici les trois syntaxes possibles de l'expression d'une jointure dans la clause **FROM** en SQL :

table_1 [INNER | { { LEFT | RIGHT | FULL } [OUTER] }] JOIN table_2
ON predicat […]

table_1 [INNER | { { LEFT | RIGHT | FULL } [OUTER] }] JOIN table_2
USING (colonnes) […]

table_1 NATURAL [INNER | { { LEFT | RIGHT | FULL } [OUTER] }] JOIN table_2 […]

Jointures en SQL

```
table_1 JOIN table_2 ON predicat
```

- **ON :** La clause **ON** correspond à la condition de jointure la plus générale. Le prédicat **predicat** est une expression logique de la même nature que celle de la clause **WHERE**.

Exemple :

```
SELECT * FROM film JOIN realisateur ON film.id_real = realisateur.id_real ;
```

Jointures en SQL

```
table_1 JOIN table_2 USING (colonnes)
```

- **USING :** La clause **USING** permet de définir sur quelle(s) colonne(s) se fait la jointure. La condition de jointure sera l'égalité des colonnes, qui doivent avoir le même nom dans les deux tables. Les paires de colonnes seront fusionnées en une colonne unique.

Exemple :

```
SELECT * FROM film JOIN realisateur  USING (id_real) ;
```

Jointures en SQL

```
table_1 NATURAL JOIN table_2
```

- **NATURAL :** Il s'agit d'une notation abrégée de la clause **USING** dans laquelle la liste de colonnes est implicite et correspond à la liste des colonnes communes aux deux tables participant à la jointure. Les colonnes communes n'apparaissent qu'une fois dans la table résultat.

Exemple :

```
SELECT * FROM film NATURAL JOIN realisateur ;
```

Jointures en SQL

table_1 INNER JOIN table_2

- **INNER JOIN :** La table résultat est constituée de toutes les juxtapositions possibles d'une ligne de la table **table_1** avec une ligne de la table **table_2** qui satisfont la condition de jointure. **INNER** est toujours optionnel, c'est le comportement par défaut de **JOIN**.

Les deux sont équivalents :

```
SELECT * FROM film JOIN realisateur  USING (id_real) ;

SELECT * FROM film INNER JOIN realisateur  USING (id_real) ;
```

Jointures en SQL

table_1 LEFT [OUTER] JOIN table_2

- **LEFT JOIN :** Dans un premier temps, une jointure interne (**INNER JOIN**) est effectuée. Ensuite, les lignes de **table_1** qui n'apparaissent pas dans la table résultat de la jointure interne sont ajoutées à la table résultats. Les attributs correspondant à la table **table_2**, pour cette ligne, sont affectés de la valeur **NULL**. **OUTER** est toujours optionnel, il est implicite dans **LEFT**.

Les deux sont équivalents :

```
SELECT * FROM film LEFT JOIN realisateur USING (id_real) ;

SELECT * FROM film LEFT OUTER JOIN realisateur USING (id_real) ;
```

Jointures en SQL

table_1 RIGHT [OUTER] JOIN table_2

- **RIGHT JOIN :** Dans un premier temps, une jointure interne est effectuée. Ensuite, les lignes de **table_2** qui n'apparaissent pas dans la table résultat de la jointure interne sont ajoutées à la table résultats. Les attributs correspondant à la table **table_1**, pour cette ligne, sont affectés de la valeur **NULL**. **OUTER** est toujours optionnel, il est implicite dans **RIGHT**.

Les deux sont équivalents :

```
SELECT * FROM film RIGHT JOIN realisateur  USING (id_real) ;

SELECT * FROM film RIGHT OUTER JOIN realisateur  USING (id_real) ;
```

Jointures en SQL

```
table_1 FULL [OUTER] JOIN  table_2
```

- **FULL JOIN :** La jointure externe bilatérale est la combinaison des deux opérations précédentes (**LEFT OUTER JOIN** et **RIGHT OUTER JOIN**) afin que la table résultat contienne au moins une occurrence de chacune des lignes des deux tables impliquées dans l'opération de jointure. **OUTER** est toujours optionnel, il est implicite dans **FULL**.

Les deux sont équivalents :

```
SELECT * FROM film FULL JOIN realisateur  USING (id_real) ;

SELECT * FROM film FULL OUTER JOIN realisateur  USING (id_real) ;
```

Jointures en SQL

Remarques

- Des jointures de n'importe quel type peuvent être chaînées les unes derrière les autres.
- Les jointures peuvent également être imbriquées étant donné que les tables **table_1** et **table_2** peuvent très bien être elles-mêmes le résultat de jointures de n'importe quel type.
- Les opérations de jointures peuvent être parenthésées afin de préciser l'ordre dans lequel elles sont effectuées.
- En l'absence de parenthèses, les jointures s'effectuent de gauche à droite.

Exemple :

```
SELECT * FROM CITIES JOIN (FLIGHTS NATURAL JOIN COUNTRIES) ON
CITIES.AIRPORT = FLIGHTS. AIRPORT ;
```

Exemple

Réalisateurs		
id_real	**nom**	**prenom**
1	von Trier	Lars
4	Tarantino	Quentin
3	Eastwood	Clint
2	Parker	Alan

Films		
id_film	**id_real**	**titre**
1	1	Dogville
2	1	Breaking the waves
3	5	Faux-Semblants
4	5	Crash
5	3	Chasseur blanc, cœur noir

A partir de ces deux tables, on peut faire une jointure naturelle, une équijointure et des jointures gauches, droites et bilatérales.

Équijointure

Table obtenue :

Équijointure					
id_film	**id_real**	**titre**	**id_real**	**nom**	**prenom**
1	1	Dogville	1	von Trier	Lars
2	1	Breaking the waves	1	von Trier	Lars
5	3	Chasseur blanc, cœur noir	3	Eastwood	Clint

L'**Équijointure** entre les tables **film** et **réalisateur** peut s'écrire indifféremment de l'une des manières suivantes :

```
SELECT * FROM film, realisateur WHERE film.id_real = realisateur.id_real;
SELECT * FROM film JOIN realisateur ON film.id_real = realisateur.id_real;
SELECT * FROM film INNER JOIN realisateur ON film.id_real = realisateur.id_real;
```

Jointure naturelle

Table obtenue :

Jointure naturelle				
id_film	**id_real**	**titre**	**nom**	**prenom**
1	1	Dogville	von Trier	Lars
2	1	Breaking the waves	von Trier	Lars
5	3	Chasseur blanc, cœur noir	Eastwood	Clint

La **jointure naturelle** entre les tables **film** et **réalisateur** peut s'écrire indifféremment de l'une des manières suivantes :

```
SELECT * FROM film NATURAL JOIN realisateur
SELECT * FROM film NATURAL INNER JOIN realisateur;
SELECT * FROM film JOIN realisateur USING (id_real);
SELECT * FROM film INNER JOIN realisateur USING (id_real);
```

Jointures externes gauches

Table obtenue :

Jointure externe gauche				
id_film	**id_real**	**titre**	**nom**	**prenom**
1	1	Dogville	von Trier	Lars
2	1	Breaking the waves	von Trier	Lars
3	5	Faux-Semblants		
4	5	Crash		
5	3	Chasseur blanc, cœur noir	Eastwood	Clint

La **jointure externe gauche** entre les tables **film** et **réalisateur** peut s'écrire indifféremment de l'une des manières suivantes :

```
SELECT * FROM film NATURAL LEFT JOIN realisateur;
SELECT * FROM film NATURAL LEFT OUTER JOIN realisateur;
SELECT * FROM film LEFT JOIN realisateur USING (id_real);
SELECT * FROM film LEFT OUTER JOIN realisateur USING (id_real);
```

Jointures externes gauches

Table obtenue, avec colonne dupliquée:

Jointure externe gauche					
id_film	**id_real**	**titre**	**id_real**	**nom**	**prenom**
1	1	Dogville	1	von Trier	Lars
2	1	Breaking the waves	1	von Trier	Lars
3	5	Faux-Semblants	5		
4	5	Crash	5		
5	3	Chasseur blanc, cœur noir	3	Eastwood	Clint

La **jointure externe gauche** avec **colonne dupliquée** entre les tables **film** et **réalisateur** peut s'écrire indifféremment de l'une des manières suivantes :

```
SELECT * FROM film LEFT JOIN realisateur
        ON film.id_real = realisateur.id_real;
```

```
SELECT * FROM film LEFT OUTER JOIN realisateur
        ON film.id_real = realisateur.id_real;
```

Jointures externes droites

Table obtenue :

Jointure externe droite				
id_real	**id_film**	**titre**	**nom**	**prenom**
1	1	Dogville	von Trier	Lars
1	2	Breaking the waves	von Trier	Lars
2			Parker	Alan
3	5	Chasseur blanc, cœur noir	Eastwood	Clint
4			Tarantino	Quentin

La **jointure externe droite** entre les tables **film** et **réalisateur** peut s'écrire indifféremment de l'une des manières suivantes :

```
SELECT * FROM film NATURAL RIGHT JOIN realisateur;
SELECT * FROM film NATURAL RIGHT OUTER JOIN realisateur;
SELECT * FROM film RIGHT JOIN realisateur USING (id_real);
SELECT * FROM film RIGHT OUTER JOIN realisateur USING (id_real);
```

Jointures externes bilatérales

Table obtenue :

Jointure externe bilatérale				
id_real	**id_film**	**titre**	**nom**	**prenom**
1	1	Dogville	von Trier	Lars
1	2	Breaking the waves	von Trier	Lars
2			Parker	Alan
3	5	Chasseur blanc, cœur noir	Eastwood	Clint
4			Tarantino	Quentin
5	3	Faux-Semblants		
5	4	Crash		

La **jointure externe bilatérale** entre les tables **film** et **réalisateur** peut s'écrire indifféremment de l'une des manières suivantes :

```
SELECT * FROM film NATURAL FULL JOIN realisateur;
SELECT * FROM film NATURAL FULL OUTER JOIN realisateur;
SELECT * FROM film FULL JOIN realisateur USING (id_real);
SELECT * FROM film FULL OUTER JOIN realisateur USING (id_real);
```

Traduction du produit cartésien

L'opérateur de produit cartésien *R1* × *R2* se traduit en SQL par la requête :

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

ou

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

Traduction du produit cartésien

- **Le produit cartésien n'est rien d'autre qu'une jointure dans laquelle l'expression logique *E* est toujours vraie : $R_1 \times R_2 = R_1 \bowtie_{true} R_2$**

- **Les quatre écritures suivantes sont équivalentes :**

```
SELECT * FROM R1, R2 ;
SELECT * FROM R1 CROSS JOIN R2 ;
SELECT * FROM R1 JOIN R2 ON TRUE ;
SELECT * FROM R1 INNER JOIN R2 ON TRUE ;
```

Dans la pratique, seule la première est utilisée !

Remarques

Dans la mesure du possible il est préférable d'utiliser un opérateur de jointure (mot-clé JOIN) pour effectuer une jointure :

- Les jointures faites dans **WHERE** ne permettent pas de faire la distinction entre sélection et jointure .

- L'optimisation d'exécution de la requête est souvent meilleure lorsque l'on utilise l'opérateur **JOIN**.

- La suppression de la clause **WHERE** à des fins de tests pose des problèmes sans **JOIN**.

Requêtes imbriqués

Une requête renvoyant une table peut être imbriqué dans la clause FROM d'une autre requête.

Exemple :

```
SELECT *
FROM
        ( SELECT TIT_CODE, CLI_NOM, CLI_PRENOM
          FROM MaTable ) AS TableReponse
;
```

TableReponse est le renommage en table du résultat de la requête, car la clause FROM doit porter sur des tables nommées.

Sous-requêtes renvoyant une valeur

La plupart du temps, nous avons l'assurance de ne renvoyer qu'une valeur unique si nous utilisons une requête dont l'unique colonne est le résultat d'un calcul statistique (agrégation) comme les **MAX**, **MIN**, **AVG**, **COUNT** et **SUM**.

C'est pourquoi on trouvera souvent ce genre d'expression dans les requêtes imbriquées des filtres **WHERE** et **HAVING**, mais aussi parfois dans la clause **SELECT**.

Sous-requêtes renvoyant une valeur

Dans la clause SELECT

- **On peut placer dans la clause SELECT, à la place de colonnes, des sous-requêtes.**

Exemple 1 : Un hôtelier veux connaître l'évolution du prix moyen de ses chambres par rapport à son tarif de référence au premier janvier 2000.

CHAMBRE(Date, Prix);

Sous-requêtes renvoyant une valeur

Exemple 1 : Un hôtelier veux connaître l'évolution du prix moyen de ses chambres par rapport à son tarif de référence au premier janvier 2000.

```
SELECT      Date, AVG(Prix) –  ( SELECT AVG(Prix)
                                 FROM CHAMBRE
                                 WHERE Date = '2000-01-01‘
                               ) AS MOYENNE
FROM        CHAMBRE
GROUP BY    Date ;
```

Remarques :

- Nous n'avons plus besoin de nommer les colonnes de la sous-requête.
- La sous-requête a été placée dans une paire de parenthèses.

Sous-requêtes renvoyant une valeur

Dans les filtres WHERE et HAVING

- C'est l'endroit le plus classique pour placer une sous-requête.

Exemple 2 : Quelles sont les chambres au 01/01/2000 qui ont un prix voisin à + ou - 10 € de la moyenne des prix au 01/01/2000 ?

CHAMBRE(ID, Date, Prix);

Sous-requêtes renvoyant une valeur

Exemple 2 : Quelles sont les chambres au 01/01/2000 qui ont un prix voisin à + ou - 10 € de la moyenne des prix au 01/01/2000 ?

CHAMBRE(ID, Date, Prix);

```
SELECT   ID, Prix
FROM     CHAMBRE
WHERE    Prix – (SELECT AVG(Prix)
                 FROM CHAMBRE
                 WHERE Date = '2000-01-01‘ )
         BETWEEN -10 AND 10
         AND Date = '2000-01-01‘ ;
```

ID	Prix
2	300
6	305
9	300
16	302
19	299

Sous-requêtes renvoyant une valeur

Exemple 3 : Quels sont les mois pour lesquels le taux d'occupation de son hôtel a dépassé les 2/3 ?

RESERV(Jour, Mois, Annee);

CHAMBRE(Id);

```
SELECT Annee, Mois, COUNT(*) AS Nombre
FROM RESERV
GROUP BY Annee, Mois
HAVING COUNT(*) > ( SELECT COUNT(*) * 30 * 0.66
                    FROM CHAMBRE)
ORDER BY Annee, Mois ;
```

Annee	Mois	Nombre
1999	12	404
2000	1	400
2000	2	420
2000	3	401
2000	4	412

Sous-requêtes renvoyant une liste

Une liste de valeurs, c'est à dire une colonne, ne peut être utilisée comme critère de comparaison que par des opérateurs spécialisés.

C'est le cas de l'opérateur **IN**, mais aussi des opérateurs **ALL** et **ANY**.

Sous-requêtes renvoyant une liste

- **Dans le prédicat IN**

L'opérateur **IN** est utilisable dans tous les prédicats, c'est pourquoi on le retrouve dans les filtres **WHERE** et **HAVING**.

Pour alimenter une liste de valeur pour le prédicat **IN**, il suffit de placer une requête ne renvoyant qu'une seule colonne.

Sous-requêtes renvoyant une liste

- **Dans le prédicat IN**

Exemple 4 : Monsieur BOUVIER vient réserver une chambre, et comme il s'y prend à l'avance, il aimerait prendre une chambre dans laquelle il n'a jamais dormi au cours de l'année 2001.

```
SELECT IdCh
FROM CHAMBRE
WHERE IdCh NOT IN ( SELECT DISTINCT C.IdCh
                    FROM RESERV NATURAL JOIN CLIENT C
                    WHERE C.NOM ='BOUVIER' AND Annee = 2001) ;
```

IdCh
5
8

Sous-requêtes renvoyant une liste

- **Dans le prédicat IN**

Beaucoup de requêtes utilisant le **IN** peuvent être simplifiées en utilisant des jointures. Le **IN** par des jointures internes, le **NOT IN** par des jointures externes avec une clause **HAVING COUNT**(...) = 0. En général les performances seront meilleures.

```
SELECT DISTINCT IdCh
FROM      CHAMBRE H
          NATURAL LEFT JOIN RESERV J
          LEFT JOIN T_CLIENT C ON J.IdCli = C. IdCli AND CLI_NOM ='BOUVIER'
                    AND Annee =  ’2001’
GROUP BY H.IdCh, J. IdCh
HAVING COUNT(C. IdCli) = 0
ORDER BY H. IdCh
```

Sous-requêtes renvoyant une liste

L'opérateur **IN** que nous venons de voir, ne permet qu'une comparaison avec une stricte égalité. Il arrive que l'on soit confronté au cas où l'on souhaite que le critère de comparaison des deux ensembles soit une inégalité. Par exemple, pour trouver une valeur supérieure ou égale à toutes les valeurs d'un ensemble donné.

Ceci est possible à l'aide des opérateurs **ALL** et **ANY**. Les opérateurs **ANY** et **ALL** permettent de comparer des ensembles de valeurs de manière globale.

- **ALL** compare toutes les valeurs pour que le prédicat soit vrai.
- **ANY** est vrai si au moins une valeur de l'ensemble répond vrai à la comparaison.

Sous-requêtes renvoyant une liste

- **Dans le prédicat ALL**

Exemple 5: Quel est l'étage qui permet de coucher le maximum de personnes?

CHAMBRE(IdCh, NbLit, Etage);

```
SELECT Etage
FROM CHAMBRE
GROUP BY Etage
HAVING SUM(NbLit) >= ALL (SELECT SUM(NbLit)
                          FROM CHAMBRE
                          GROUP BY Etage )
;
```

Sous-requêtes renvoyant une liste

- **Dans le prédicat ANY**

Exemple 6: Tous les étages sauf celui dont le couchage est le plus petit ?

CHAMBRE(IdCh, NbLit, Etage);

```
SELECT Etage
FROM CHAMBRE
GROUP BY Etage
HAVING SUM(NbLit) > ANY (SELECT SUM(NbLit)
                         FROM CHAMBRE
                         GROUP BY Etage )
;
```

Sous-requêtes renvoyant une table

N'importe quelle requête est capable de renvoyer une table, car un résultat de requête est bien une table. C'est l'essence même de la fonction d'une requête.

Il est possible de placer une sous-requête dans la clause **FROM** de n'importe quelle requête à la place d'un nom de table !

Lorsque l'on place une sous-requête en tant que table dans la clause FROM d'une requête, il faut lui donner systématiquement un nom.

Sous-requêtes renvoyant une table

Exemple 6: Quel est le maximum de la somme des couchages des étages ?

CHAMBRE(IdCh, NbLit, Etage);

```
SELECT      MAX(Couchage) AS Max_Couchage
FROM        (SELECT SUM(NbLit) AS Couchage
              FROM CHAMBRE
              GROUP BY Etage) T
;
```

Remarque : **SELECT MAX(SUM(NbLit))** n'est pas accepté en SQL.

Sous-requêtes vide, non vide

Dans ce cas, si la sous-requête renvoie un résultat quelconque, alors le prédicat vaut vrai. Si le sous-requête ne renvoi aucune ligne, le prédicat vaut faux.

On peut utiliser pour cela deux prédicats spécialisés qui sont **EXISTS** et **UNIQUE**.

Remarques (1) :

- Le prédicat **EXISTS** est en général plus rapide que le prédicat **IN**.
- Le prédicat **EXISTS** n'a aucun intérêt sans une sous-requête corrélée.
- Il convient de toujours utiliser l'étoile comme unique contenu de la clause **SELECT** de la sous-requête pour un traitement performant.

Sous-requêtes vide, non vide

Dans ce cas, si la sous-requête renvoie un résultat quelconque, alors le prédicat vaut vrai. Si le sous-requête ne renvoi aucune ligne, le prédicat vaut faux.

On peut utiliser pour cela deux prédicats spécialisés qui sont **EXISTS** et **UNIQUE**.

Remarques (2) :

- Le prédicat **UNIQUE** est en général beaucoup plus rapide que d'autre solution équivalentes.
- Le prédicat **UNIQUE** n'a aucun intérêt sans une sous-requête corrélée.
- Contrairement à **EXISTS**, il convient de toujours spécifier les colonnes visées dans la clause **SELECT**.

Sous-requêtes vide, non vide

- **Dans le prédicat EXISTS**

Le prédicat EXISTS permet de tester l'existence de données dans la sous-requête. Si la sous-requête renvoie au moins une ligne, même remplie de marqueurs NULL, le prédicat est vrai.

Exemple 7 : Nous voulons obtenir le total du couchage de l'hôtel, toutes chambres confondues, à condition qu'il y ait au moins une chambre dotée d'un couchage pour au moins 3 personnes :

CHAMBRE(IdCh, NbLit, Etage);

Sous-requêtes vide, non vide

- **Dans le prédicat EXISTS**

Exemple 7 : Nous voulons obtenir le total du couchage de l'hôtel, toutes chambres confondues, à condition qu'il y ait au moins une chambre dotée d'un couchage pour au moins 3 personnes :

CHAMBRE(IdCh, NbLit, Etage);

```
SELECT SUM(NbLit) AS TOTAL_COUCHAGE
FROM CHAMBRE
WHERE EXISTS (SELECT * FROM CHAMBRE WHERE NbLit>= 3) ;
```

Sous-requêtes vide, non vide

Dans le prédicat UNIQUE

UNIQUE est un raffinement du prédicat EXISTS. UNIQUE vaut faux si au moins deux lignes renvoyées par la sous-requête comporte les mêmes données.

Exemple 8 : Nous voulons obtenir le total du couchage de l'hôtel, toutes chambres confondues, à condition qu'il n'y ait qu'une seule chambre dotée d'un couchage pour exactement 5 personnes :

CHAMBRE(IdCh, NbLit, Etage);

Sous-requêtes vide, non vide

Dans le prédicat UNIQUE

Exemple 8 : Nous voulons obtenir le total du couchage de l'hôtel, toutes chambres confondues, à condition qu'il n'y ait qu'une seule chambre dotée d'un couchage pour exactement 5 personnes :

CHAMBRE(IdCh, NbLit, Etage);

```
SELECT SUM(NbLit) AS TOTAL_COUCHAGE
FROM CHAMBRE
WHERE UNIQUE    (SELECT NbLit
                 FROM CHAMBRE
                 WHERE NbLit = 5)
;
```

Les sous-requêtes corrélées

Une sous-requête corrélée est une sous-requête qui s'exécute pour chaque ligne de la requête principale et non une fois pour toute. Pour cela, il suffit de faire varier une condition (en général un prédicat) en rappelant dans la sous-requête la valeur d'une colonne de la requête principale.

Exemple 9 : Trouver les clients qui ont un prénom en commun :

CLIENT(IdCli, Nom, Prenom);

```
SELECT IdCli, Nom, Prenom
FROM CLIENT C1
WHERE Prenom IN (  SELECT Prenom
                   FROM CLIENT C2
                   WHERE C1. IdCli != C2. IdCli) ;
```

Les sous-requêtes corrélées

```
SELECT IdCli, Nom, Prenom
FROM CLIENT C1
WHERE Premon IN (  SELECT Prenom
                   FROM CLIENT C2
                   WHERE C1. IdCli != C2. IdCli) ;
```

- Ici, la corrélation se fait dans la clause WHERE : **C1. IdCli != C2. IdCli**.
- Notons que pour obtenir cette corrélation, il faut donner des surnoms aux tables.

Sous-requêtes ou jointures ?

Il est souvent possible de trouver un équivalent a une sous-requête avec des jointures, mais toutes les sous-requêtes ne peuvent pas trouver leur équivalent sous forme de jointures.

Voici quelques exemples de sous-requêtes ne possédant aucun équivalent sous forme de jointure :

```
SELECT * FROM   TABLE_1
WHERE  COLONNE_1 + 3 = (SELECT MAX(COLONNE_2) FROM TABLE_2)
```

```
SELECT * FROM   TABLE_1
        JOIN (SELECT MAX(COLONNE_1) AS MAX_COL_1 FROM TABLE_2) TABLE_2
        ON TABLE_1.COLONNE_1 + 4 = TABLE_2.COLONNE_1
```

Opérateurs ensemblistes SQL

Les résultats de deux requêtes peuvent être combinés en utilisant les opérateurs ensemblistes d'*union* (**UNION**), d'*intersection* (**INTERSECT**) et de *différence* (**MINUS**).

requête_1 { UNION | INTERSECT | MINUS } [ALL] requête_2

Remarques (1) :

- Il faut que les deux requêtes aient le même schéma : le même nombre de colonnes respectivement du même type.
- Les noms de colonnes (titres) sont ceux de la première requête (**requête_1**).
- On peut chaîner plusieurs opérations ensemblistes. L'expression est évaluée de gauche à droite, mais on peut modifier l'ordre d'évaluation en utilisant des parenthèses.

Opérateurs ensemblistes SQL

Les résultats de deux requêtes peuvent être combinés en utilisant les opérateurs ensemblistes d'*union* (**UNION**), d'*intersection* (**INTERSECT**) et de *différence* (**MINUS**).

requête_1 { UNION | INTERSECT | MINUS } [ALL] requête_2

Remarques (2) :

- Dans une requête on ne peut trouver qu'une seule instruction ORDER BY. Si elle est présente, elle doit être placée dans la dernière requête. La clause ORDER BY ne peut faire référence qu'aux numéros des colonnes et non pas à leurs noms.

- Contrairement à la commande SELECT, le comportement par défaut des opérateurs ensemblistes élimine les doublons. Pour les conserver, il faut utiliser le mot-clé ALL.

- Attention, il s'agit bien d'opérateurs portant sur des tables générées par des requêtes. On ne peut pas faire directement l'union de deux tables de la base de données.

Union

L'union est une opération portant sur deux relations ayant le même schéma et construisant une troisième constituée des n-uplets appartenant à chacune des deux relations sans doublon.

$$R = R1 \cup R2$$

Union

L'union est une opération portant sur deux relations ayant le même schéma et construisant une troisième constituée des n-uplets appartenant à chacune des deux relations sans doublon.

Exemple :

R_1	
Nom	**Prénom**
Durand	Caroline
Germain	Stan
Dupont	Lisa
Germain	Rose-Marie

R_2	
Nom	**Prénom**
Dupont	Lisa
Juny	Carole
Fourt	Lisa

$R = R1 \cup R2$	
Nom	**Prénom**
Durand	Caroline
Germain	Stan
Dupont	Lisa
Germain	Rose-Marie
Juny	Carole
Fourt	Lisa

Union

L'union est une opération portant sur deux relations ayant le même schéma et construisant une troisième constituée des n-uplets appartenant à chacune des deux relations sans doublon.

- R_1 et R_2 doivent avoir les mêmes attributs.
- Si une même occurrence existe dans R_1 et R_2, elle n'apparaît qu'une seule fois dans le résultat de l'union.
- Le résultat de l'union est une nouvelle relation qui a les mêmes attributs que R_1 et R_2.
- Si R_1 (respectivement R_2) est vide, la relation qui résulte de l'union est identique à R_2 (respectivement R_1).

Traduction de l'opérateur d'union

L'opérateur d'union $R_1 \cup R_2$ se traduit en SQL par la requête :

SELECT * FROM R_1 UNION SELECT * FROM R_2

Traduction de l'opérateur d'union

Exemple 1 :

Table1	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Truc	Patrick

U

Table2	
Nom	**Prénom**
Pouf	Jean
Chose	Jules

=

Résultat	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Pouf	Jean
Truc	Patrick

```
SELECT * FROM Table1 UNION SELECT * FROM Table2
```

```
SELECT Nom, Prénom FROM Table1 UNION SELECT Nom, Prénom FROM Table2
```

Traduction de l'opérateur d'union

Exemple 2 :

Table1	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Truc	Patrick

U

Table2	
LastName	**FirstName**
Pouf	Jean
Chose	Jules

=

Résultat	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Pouf	Jean
Truc	Patrick

```
SELECT * FROM Table1 UNION SELECT * FROM Table2
```

```
SELECT Nom, Prénom FROM Table1 UNION SELECT LastName, FirstName FROM Table2
```

Traduction de l'opérateur d'union

Exemple 3 :

Table1	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Truc	Patrick

U

Table2	
Nom	**Prénom**
Pouf	Jean
Chose	Jules

=

Résultat	
C1	**C2**
Chose	Jules
Machin	Pierre
Pouf	Jean
Truc	Patrick

```
SELECT Nom AS C1, Prénom AS C2 FROM Table1
UNION
SELECT Nom , Prénom FROM Table2
```

Traduction de l'opérateur d'union

Exemple 4 :

Table1	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Truc	Patrick

U

Table2	
Nom	**Prénom**
Pouf	Jean
Chose	Jules

=

Résultat	
Nom	**Prénom**
Machin	Pierre
Pouf	Jean
Truc	Patrick

```
SELECT * FROM Table1 WHERE Nom>"D"
UNION
SELECT * FROM Table2 WHERE Nom>"D"
```

Traduction de l'opérateur d'union

Exemple 5 :

Table1	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Truc	Patrick

U

Table2	
Nom	**Prénom**
Pouf	Jean
Chose	Jules

=

Résultat	
C1	**C2**
Chose	Jules
Machin	Pierre
Pouf	Jean
Truc	Patrick
Chose	Jules

```
SELECT * FROM Table1
UNION ALL
SELECT * FROM Table2
```

Intersection

L'intersection est une opération portant sur deux relations ayant le même schéma et construisant une troisième dont les n-uplets sont constitués de ceux communs aux deux relations.

$$R = R_1 \cap R_2$$

Intersection

L'intersection est une opération portant sur deux relations ayant le même schéma et construisant une troisième dont les n-uplets sont constitués de ceux communs aux deux relations.

Exemple :

Relation R_1	
Nom	**Prénom**
Durand	Caroline
Germain	Stan
Dupont	Lisa
Germain	Rose-Marie
Juny	Carole

Relation R_2	
Nom	**Prénom**
Dupont	Lisa
Juny	Carole
Fourt	Lisa
Durand	Caroline

Relation $R = R_1 \cap R_2$	
Nom	**Prénom**
Durand	Caroline
Dupont	Lisa
Juny	Carole

Intersection

L'intersection est une opération portant sur deux relations ayant le même schéma et construisant une troisième dont les n-uplets sont constitués de ceux communs aux deux relations.

- R_1 et R_2 doivent avoir les mêmes attributs.
- Le résultat de l'intersection est une nouvelle relation qui a les mêmes attributs que R_1 et R_2.
- Si R_1 ou R_2 ou les deux sont vides, la relation qui résulte de l'intersection est vide.

Traduction de l'intersection

L'opérateur d'intersection R1 ∩ R2 se traduit en SQL par la requête :

SELECT * FROM R_1 INTERSECT SELECT * FROM R_2

Traduction de l'intersection

Exemple :

Table1	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Truc	Patrick

∩

Table2	
Nom	**Prénom**
Pouf	Jean
Chose	Jules

=

Résultat	
Nom	**Prénom**
Chose	Jules

```
SELECT * FROM Table1 INTERSECT SELECT * FROM Table2
```

Traduction de l'intersection

Exemple :

Table1	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Truc	Patrick

∩

Table2	
Nom	**Prénom**
Pouf	Jean
Chose	Jules

=

Résultat	
Nom	**Prénom**
Chose	Jules

```
SELECT Nom, Prénom
FROM Table1
WHERE   Nom IN (SELECT Nom FROM Table2)
        AND
        Prénom IN (SELECT Prénom FROM Table2)
```

Différence

La différence est une opération portant sur deux relations ayant le même schéma et construisant une troisième relation dont les n-uplets sont constitués de ceux ne se trouvant que dans la relation R1.

Exemple :

Relation R_1	
Nom	**Prénom**
Durand	Caroline
Germain	Stan
Dupont	Lisa
Germain	Rose-Marie
Juny	Carole

Relation R_2	
Nom	**Prénom**
Dupont	Lisa
Juny	Carole
Fourt	Lisa
Durand	Caroline

Relation $R = R_1 - R_2$	
Nom	**Prénom**
Germain	Stan
Germain	Rose-Marie

Différence

La différence est une opération portant sur deux relations ayant le même schéma et construisant une troisième relation dont les n-uplets sont constitués de ceux ne se trouvant que dans la relation R1.

- R_1 et R_2 doivent avoir les mêmes attributs.
- Le résultat de la différence est une nouvelle relation qui a les mêmes attributs que R_1 et R_2.
- Si R_2 est vide, la différence est identique à R_1.

Traduction de la différence

L'opérateur de différence R1 – R2 se traduit en SQL par la requête :

```
SELECT * FROM R1 MINUS SELECT * FROM R2
```

Traduction de la différence

Exemple :

Table1	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Truc	Patrick

-

Table2	
Nom	**Prénom**
Pouf	Jean
Chose	Jules

=

Résultat	
Nom	**Prénom**
Machin	Pierre
Truc	Patrick

SELECT * FROM Table1 MINUS SELECT * FROM Table2

Traduction de la différence

Exemple :

Table1	
Nom	**Prénom**
Chose	Jules
Machin	Pierre
Truc	Patrick

-

Table2	
Nom	**Prénom**
Pouf	Jean
Chose	Jules

=

Résultat	
Nom	**Prénom**
Machin	Pierre
Truc	Patrick

```
SELECT Nom, Prénom
FROM Table1
WHERE   Nom NOT IN (SELECT Nom FROM Table2)
        AND Prénom NOT IN (SELECT Prénom FROM Table2)
```

Division

La division est une opération portant sur deux relations R1 et R2, telles que le schéma de R2 est strictement inclus dans celui de R1, qui génère une relation regroupant toutes les parties d'occurrences de la relation R1 qui sont associées à toutes les occurrences de la relation R2.

Exemple :

Relation *Enseignant*	
Enseignant	**Étudiant**
Germain	Dubois
Fidus	Pascal
Robert	Dubois
Germain	Pascal
Fidus	Dubois
Germain	Durand
Robert	Durand

Relation Étudiant
Nom
Dubois
Pascal

Relation *R* = *Enseignant* ÷ Étudiant
Enseignant
Germain
Fidus

Division

La division est une opération portant sur deux relations R1 et R2, telles que le schéma de R2 est strictement inclus dans celui de R1, qui génère une relation regroupant toutes les parties d'occurrences de la relation R1 qui sont associées à toutes les occurrences de la relation R2.

- Si $R = R_1 \div R_2$, le produit cartésien $R \times R_2$ est toujours inclut dans R_1.
- Si R_1 est vide, la relation qui résulte de la division est vide.
- La relation R_2 ne peut pas être vide et tous les attributs de R_2 doivent être présents dans R_1.
- R_1 doit posséder au moins un attribut de plus que R_2 (inclusion stricte).
- Le résultat de la division est une nouvelle relation qui a tous les attributs de R_1 sans aucun de ceux de R_2.

Traduction de la division

Il n'existe pas de commande SQL permettant de réaliser directement une division !!

Exemple :

Quels sont les acteurs qui ont joué dans tous les films de Lars von Trier ?

➔ *Quels sont les acteurs qui vérifient : quel que soit un film de Lars von Trier, l'acteur a joué dans ce film.*

∀ (pour tout) n'existe pas en SQL. Mais on peut utiliser ∃ (il existe) :

$$\forall\, x\, P(x) = \neg\, \exists\, x\, \neg\, P(x)$$

On peut donc reformuler le problème de la manière suivante :

➔ *Quels sont les acteurs qui vérifient : il est faux qu'il existe un film de Lars von Trier dans lequel l'acteur n'a pas joué.*

Traduction de la division

Quels sont les acteurs qui vérifient : il est faux qu'il existe un film de Lars von Trier dans lequel l'acteur n'a pas joué ?

```
SELECT DISTINCT nom, prenom
FROM individu AS acteur_lars
WHERE NOT EXISTS
        (SELECT *
        FROM    ( film JOIN individu ON num_realisateur =  num_individu
                AND nom = 'von Trier' AND prenom = 'Lars' ) AS film_lars
        WHERE NOT EXISTS
                (SELECT *
                FROM individu JOIN jouer ON num_individu = num_acteur
                AND num_individu = acteur_lars.num_individu
                AND num_film = film_lars.num_film ) ) ;
;
```

Traduction de la division

Il existe une autre solution :

➔ *Quels sont les acteurs qui vérifient : le nombre de films réalisés par Lars von Trier dans lequel l'acteur a joué est égal au nombre de films réalisés par Lars von Trier ?*

```
SELECT DISTINCT nom, prénom
FROM individu AS acteur_lars
WHERE
        (SELECT DISTINCT COUNT(*)
        FROM    jouer JOIN film ON jouer.num_film = film.num_film JOIN individu ON
                        num_realisateur = num_individu
        WHERE   nom = 'von Trier' AND prenom = 'Lars'
                AND jouer.num_acteur = acteur_lars.num_individu )
=
        (SELECT DISTINCT COUNT(*)
        FROM    film JOIN individu ON num_realisateur = num_individu
        WHERE   nom = 'von Trier' AND prénom = 'Lars' )
;
```

Partie VI

-

Les vues

Introduction

Une vue est une table virtuelle résultant d'une requête a laquelle on a donné un nom.

Les vues permettent de :

- Cacher aux utilisateurs certaines colonnes ou certaines lignes. Ceci fourni un niveau de confidentialité et de sécurité supplémentaire.
- Simplifier l'utilisation de tables comportant de nombreuses colonnes/lignes ou des noms complexes, en créant des vues avec des structures plus simples et des noms plus intelligibles.
- Nommer des requêtes fréquemment utilisées pour simplifier et accélérer l'écriture de requête y faisant référence.

Introduction

Une vue est une table virtuelle résultant d'une requête a laquelle on a donné un nom.

Remarques :

- La vue ne stocke pas les données, mais fait référence à une ou plusieurs tables d'origine à travers une requête **SELECT**, requête qui est exécutée chaque fois que la vue est référencée.
- De ce fait, toute modification de données dans les tables d'origine est immédiatement visible dans la vue dès que celle-ci est à nouveau référencée dans une requête.

Créer une vue

```
CREATE [ OR REPLACE ]
VIEW nom [ ( nom_colonne [, ...] ) ]
AS requête ;
```

CREATE OR REPLACE VIEW :

- définit une nouvelle vue, ou la remplace si une vue du même nom existe déjà, si la nouvelle requête génère un ensemble de colonnes identiques.

nom :

- Le nom de la vue à créer. Le nom de la vue doit être différent du nom des autres vues, tables, séquences ou index du même schéma.

nom_colonne :

- Une liste optionnelle de noms à utiliser pour les colonnes de la vue. Si elle n'est pas donnée, le nom des colonnes sera déduit de la requête.

requête :

- Une requête (c'est-à-dire une instruction **SELECT**) qui définit les colonnes et les lignes de la vue.

Créer une vue

```
CREATE [ OR REPLACE ]
VIEW nom [ ( nom_colonne [, ...] ) ]
AS requête ;
```

Exemple: on peut créer une vue qui ne « contient » que les cinémas parisiens :

```
CREATE VIEW ParisCinemas
AS SELECT * FROM Cinema WHERE ville = 'Paris' ;
```

On veux restreindre la vision des cinémas parisiens à leur nom et à leur nombre de salles :

```
CREATE VIEW SimpleParisCinemas
AS SELECT nom, COUNT(*) AS nbSalles
FROM Cinema c JOIN Salle s ON c.Nom = s.NomCinema
WHERE ville = 'Paris'
GROUP BY c.nom ;
```

Créer une vue

Un des intérêts des vues est de donner une représentation dénormalisée de la base, en regroupant des informations par des jointures.

Exemple : créer une vue Casting donnant explicitement les titres des films, leur année et les noms et prénoms des acteurs.

```
CREATE VIEW Casting (film, année, acteur_nom, acteur_prénom)
AS
SELECT titre, année, nom, prénom
FROM Film NATURAL JOIN Rôle NATURAL JOIN Acteur ;
```

Remarque : on a donné explicitement des noms d'attributs.

Se servir d'une vue

Le nom d'une vue peut être utilisé partout où on peut mettre le nom d'une table : SELECT, UPDATE, DELETE, INSERT, GRANT

Exemple : Quels acteurs ont tourné un film en 1997, en utilisant la vue « Casting » ?

```
SELECT acteur_nom, acteur_prénom
FROM Casting
WHERE année = 1997;
```

Se servir d'une vue

Le nom d'une vue peut être utilisé partout où on peut mettre le nom d'une table : SELECT, UPDATE, DELETE, INSERT, GRANT

Une vue peut aussi servir a assurer la confidentialité des données en ne donnant accès qu'a une partie des informations

Exemple:

```
CREATE VIEW IndividuPublic
SELECT NumIndividu, NomIndividu, PrenomIndividu FROM INDIVIDU;
```

Seul le créateur de IndividuPublic pourra avoir accès a des informations complètes, et il ne donnera accès aux autres qu'a la vue (noms et prénoms, mais pas adresse, téléphone,....).

Supprimer une vue

DROP VIEW nom [, ...] [CASCADE | RESTRICT];

DROP VIEW : Supprime une vue existante.

nom : Le nom de la vue à supprimer (qualifié ou non du nom du schéma).

CASCADE : Supprime automatiquement les objets qui dépendent de la vue (par exemple d'autres vues).

RESTRICT : Refuse de supprimer la vue si un objet en dépend. Ceci est la valeur par défaut.

La suppression d'une vue ne supprime pas les données !

Renommer une vue

RENAME <ancien nom> **TO** <nouveau nom> ;

Modifier les tables d'une vue

La modification des tables en utilisant une vue (UPDATE, DELETE, INSERT) est possibles seulement si :

- **On peut retrouver la ou les lignes de la table originale concerné.**

- **La vue contient pas :**
  - Un opérateur ensembliste (UNION, EXCEPT, INTERSECT).
  - Un opérateur DISTINCT.
  - Toute colonne non référencée doit pouvoir être mise à NULL ou disposer d'une valeur par défaut.
  - Une fonction d'agrégation comme attribut (COUNT, SUM, …).
  - Une clause GROUP BY, ORDER BY ou CONNECT BY.
  - Une jointure.

Modifier les tables d'une vue

Exemple 1 :

Imaginons que l'on souhaite insérer une ligne dans la vue Casting.

```
CREATE VIEW Casting (film, année, acteur_nom, acteur_prénom)
AS SELECT titre, année, nom, prénom
FROM Film NATURAL JOIN Rôle NATURAL JOIN Acteur ;

INSERT INTO CASTING (film, annee, acteur, prenom)
VALUES ('Titanic', 1998, 'DiCaprio', 'Leonardo');
```

Modifier les tables d'une vue

Exemple 1 :

Imaginons que l'on souhaite insérer une ligne dans la vue Casting.

```
CREATE VIEW Casting (film, année, acteur_nom, acteur_prénom)
AS SELECT titre, année, nom, prénom
FROM Film NATURAL JOIN Rôle NATURAL JOIN Acteur ;
```

~~INSERT INTO CASTING (film, annee, acteur, prenom)~~
~~VALUES ('Titanic', 1998, 'DiCaprio', 'Leonardo');~~ **ERROR !!**

Cet ordre s'adresse à une vue issue de trois tables. Il n'y a clairement pas assez d'information pour alimenter ces tables de manière cohérente, et l'insertion n'est pas possible (de même que toute mise à jour). De telles vues sont dites non modifiables.

Modifier les tables d'une vue

Exemple 2:

On souhaite insérer une ligne dans la table Film au travers de la vue ParisCinema.

```
CREATE VIEW ParisCinemas
AS SELECT * FROM Cinema
WHERE ville = 'Paris' ;

INSERT INTO ParisCinema
VALUES (1876, 'Breteuil', 12, 'Cite', 'Lyon');
```

Modifier les tables d'une vue

Exemple 2:

On souhaite insérer une ligne dans la table Film au travers de la vue ParisCinema.

```
CREATE VIEW ParisCinemas
AS SELECT * FROM Cinema
WHERE ville = 'Paris' ;

INSERT INTO ParisCinema
VALUES (1876, 'Breteuil', 12, 'Cite', 'Lyon');
```

OK !!!

On peut insérer dans une vue sans être en mesure de voir la ligne insérée au travers de la vue par la suite !

Modifier les tables d'une vue

On peut insérer dans une vue sans être en mesure de voir la ligne insérée au travers de la vue par la suite.

Afin d'éviter ce genre d'incohérence , SQL propose l'option WITH CHECK OPTION qui permet de garantir que toute ligne insérée dans la vue satisfait les critères de sélection de la vue.

```
CREATE VIEW ParisCinemas
AS SELECT * FROM Cinema
WHERE ville = 'Paris'
WITH CHECK OPTION ;
```

L'insertion précédente devient impossible.