

Définitions, modèles, histoires - PN maison

Définitions

Définitions générale

Déf génér. = Ensemble organisé de données ayant un objectif commun

Déf informatique = Un ensemble structuré et organisé permettant le stockage de grandes quantités de données afin d'en faciliter l'exploitation

DB = collection de données

Différent paradigmes

SGBD Système de gestion de base de données ou DBMS (DataBase Management System)

Logiciel permettant la gestion et l'accès à des bases de données

SGBDR ou **RDBMS**

SGBD dédié aux BD relationnelles (Le + utilisé, s'inspirant algèbre relationnelle)

- ▼ OLTP (Online Transaction Processing): traitement transactionnel en ligne

*Contexte d'utilisation typique des activités opérationnelles
(e-commerce, production, système de réservation, etc.)*

Système OLTP

- Répondre à un nombre élevé de transaction(ensemble solidaire de requêtes) par minutes
- Aussi performant en lecture que en écriture
- Garantir l'ACIDité des transactions

▼ Propriétés ACID

Ensemble de propriétés qui garantissent qu'une transaction est exécutée de façon fiable

- Atomicité: Une transaction se fait au complet ou pas du tout
- Cohérence: Assure que chaque transaction amènera le système d'un état valide à un autre état valide
- Isolation: Toute transaction doit s'exécuter comme si elle était la seule sur le système (pas de dépendance)
- Durabilité: Lorsqu'une transaction a été confirmée, elle demeure enregistrée même à la suite d'une panne

▼ OLAP (Online Analytical Processing): traitement analytique en ligne

*Contexte d'utilisation typique des secteurs décisionnels
(marketing, analyse financière, reporting, etc.)*

- Répondre rapidement à des requêtes d'interrogation souvent complexes(plus complexe qu'en OLTP)
- Traiter de "gros" volumes de données
- Permettre des ingestions massive de données

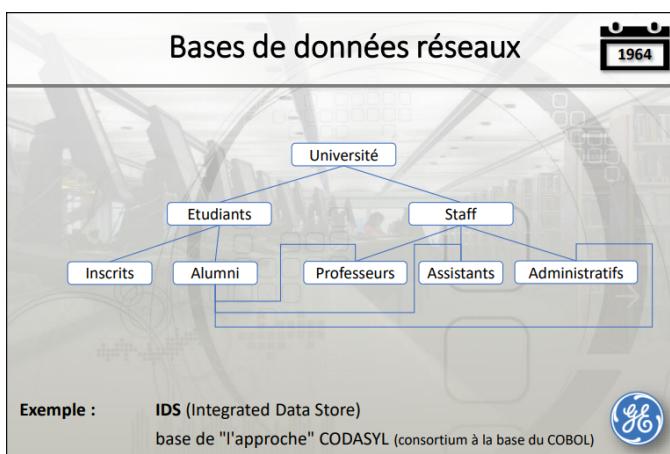
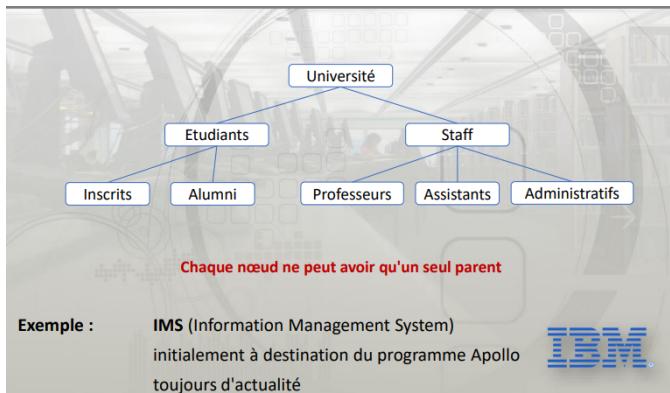
Histoire

▼ Histoire slide

Fichiers plats(flat files)

bases de données hiérarchiques

Chaque noeud = un parent



IDS (Integrated Data Store)

base de "l'approche" CODASYL (consortium à la base du COBOL)

Paradigmes hiérarchique et réseaux regroupés sous l'appellation
Navigational DBMS

Edgar Frank Codd propose le modèle relationnel, bcp critiquer au début

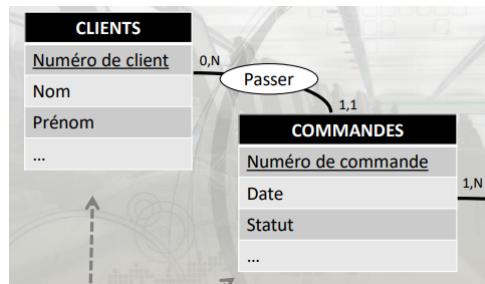
De nos jours, c'est le + utilisé et le + préféré

Modèle de DB

▼ Modèle relationnel

Composants:

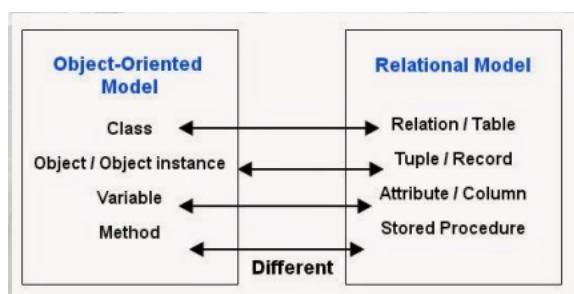
nom, ID, cardinalité, action qui s'effectue entre les deux Tables, dans DB plusieurs tables de données



▼ Modèle orientés objets

Avantages :

- Meilleure intégration avec les langages OO (plus facile pour le développeur OO)
- Plus grande flexibilité (types et relations)



Inconvénients :

- Cette flexibilité rend beaucoup de fonctionnalités "simples" (un tri par exemple) plus complexes à mettre en œuvre
- Les performances (pour de vrais cas d'utilisation)

- Les SGBDO ont quasi disparu
- En +, certaines SGBDR permettent une approche OO

▼ Bases de données semi-structurées

Très peu utilisées

Bases de données semi-structurées 2000's

```
<exempleXML>
<titre>Mes supers albums</titre>
<album>
<artiste>Rammstein</artiste>
<titre>Mutter</titre>
<annee>2001</annee>
</album>
<album annee="2017">
<artiste>Ultra Vomit</artiste>
<titre>Panzer Surprise !</titre>
</album>
</exempleXML>
```

Avantages :

- Absence de schéma
- Mieux adapté au stockage de documents

Inconvénients :

- Performances des recherches (nécessite de parser)
- Fonctionnalités peu nombreuses

▼ Bases de données NoSQL

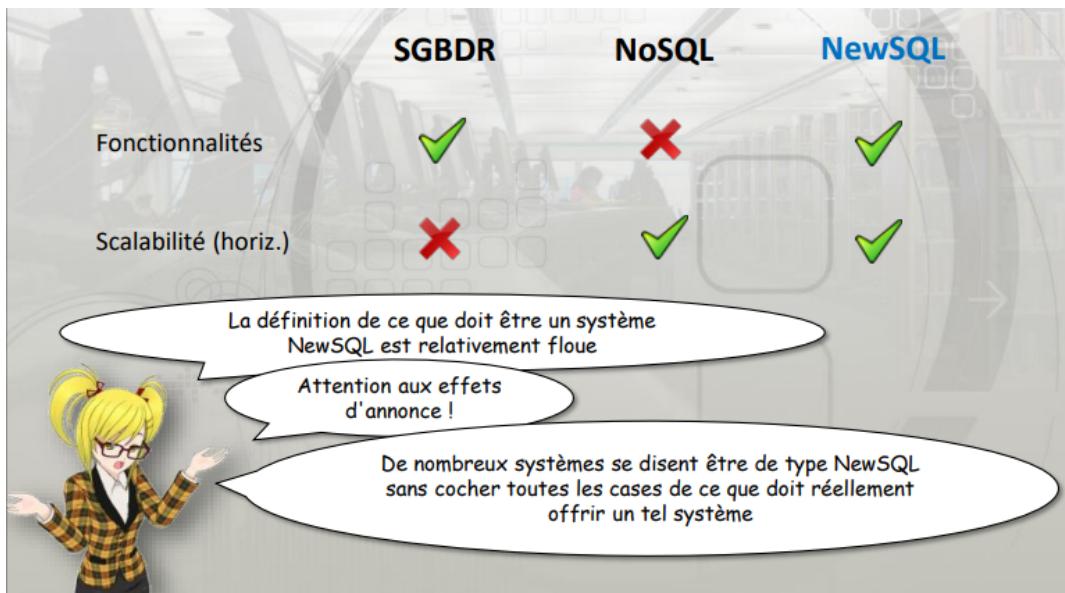
Not only SQL

Utiliser pour les bases de données géantes (Google, Amazon, Facebook ou eBay)

Simplicité

Les performances "restent bonnes" avec la montée en charge (scalabilité) en multipliant le nombre de serveurs

▼ New SQL



Normalisation

Dépendance fonctionnelle

ARTICLES	
<u>id_article</u>	
nom	
référence	
prix	
numéro_catégorie	
libellé_catégorie	
Emmental	
4578912	
1,25 €	
4	
Fromage	
Brie	
1256912	
0,98 €	
4	
Fromage	
Pink Lady	
4568941	
0,85 €	
7	
Fruits	

On dit qu'il existe une dépendance fonctionnelle entre un attribut A1 et un attribut A2, on note A1 → A2 si, connaissant une valeur de A1, on ne peut lui associer qu'une seule valeur de A2.

On dit aussi que A1 détermine A2.

A1 est la source de la dépendance fonctionnelle et A2 le but.

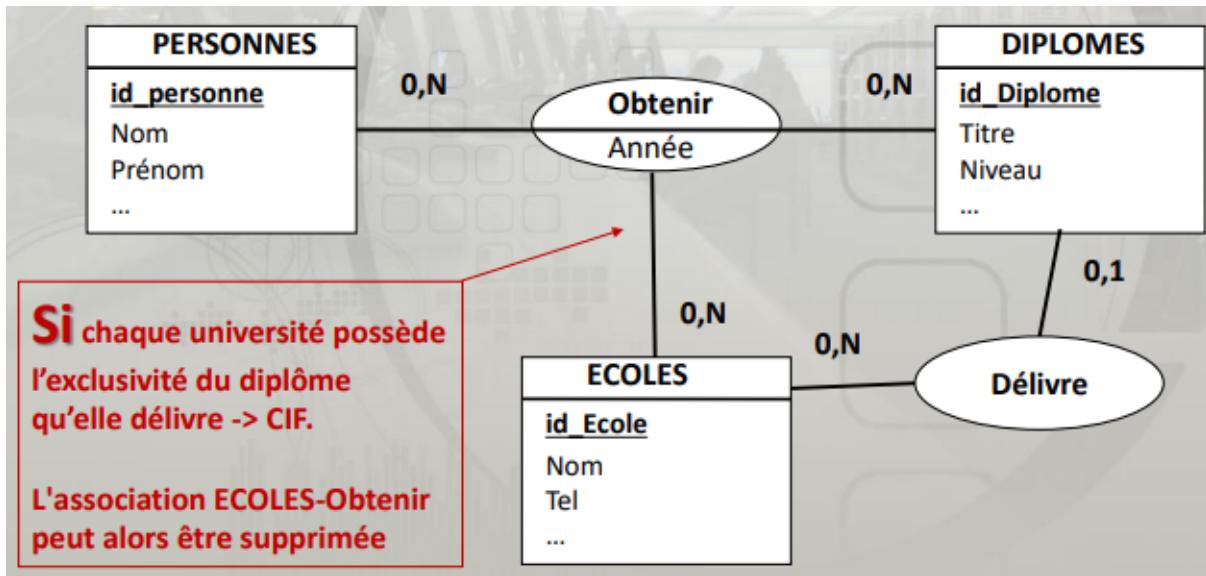
A1 = source de dépendance

A2 = but

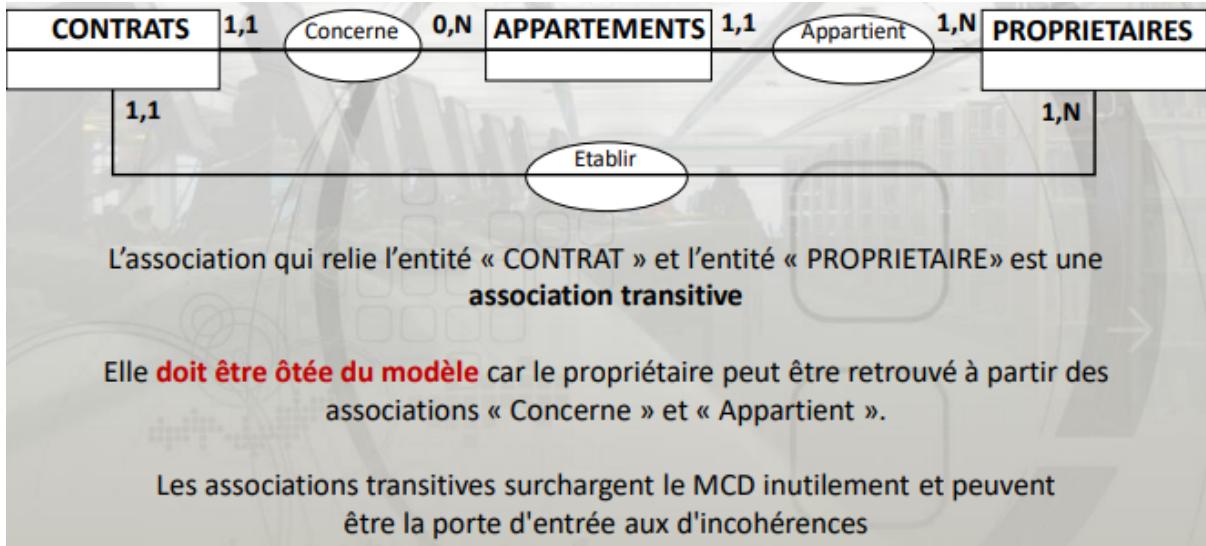
Avec A1 on connaît la valeur de A2

Contrainte d'intégrité fonctionnelle

Définit par le fait que l'une des entités participant à l'association est complètement déterminée par la connaissance d'une ou de plusieurs autres entités participant dans cette même association



Association transitive

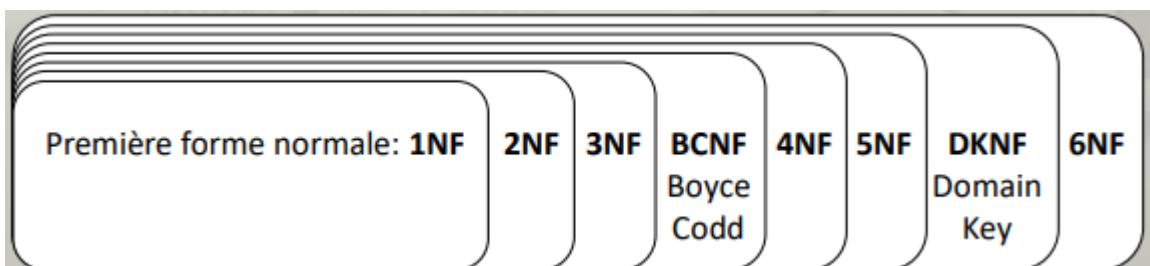


Pas en mettre car surcharge MCD, avec les liens on sait déjà retrouvé qui et qui donc inutile

Normalisation

Quoi :

- Règles de bonne pratique
But :
- Améliorer la modélisation
- Éviter les redondances des données (éviter les problèmes de mise à jour)
Contexte :
- Principalement OLTP
- Niveaux :



1. 1FN

id	nom	email
1	M. Frank Dewit	f.dewit@gmail.com, zedewit@hotmail.com
2	M. Ivan Raoul	rivan@hotmail.fr
3	Mme Leen Breckx	Leen.breckx@gmail.com, brekyleen@live.fr

Les attributs doivent être atomiques :

- Supprimer les attributs composites en promouvant au rang d'attribut les sous-attributs
- Les attributs multivalués donnent naissance à de nouvelles entités

id	titre	prenom	nom	id	email
1	M.	Frank	Dewit	1	f.dewit@gmail.com
2	M.	Ivan	Raoul	2	zedewit@hotmail.com
3	Mme	Leen	Breckx	3	rivan@hotmail.fr

0,N 1,1

id	email
1	f.dewit@gmail.com
2	zedewit@hotmail.com
3	rivan@hotmail.fr
4	Leen.breckx@gmail.com
5	brekyleen@live.fr

Attributs atomiques

2. 2FN

Respecter la 1FN

Les attributs doivent dépendre complètement de la clé et non partiellement

Ce cas se rencontre principalement avec les clés composées.

Exemple avec une entité qui contient les évaluations des films par des utilisateurs :

<u>id_film</u>	<u>id_user</u>	note	genre
1	1	3	Comédie
2	1	5	Action
2	8	4	Action
3	8	1	Aventure

Pas en 2FN !

L'attribut "genre" est propre au film (id_film) et ne dépend pas de l'utilisateur (id_user), cet attribut n'est pas dans la bonne entité.

Attribut dépendre complètement de la clé et non partiellement

3. 3FN

Respecter la 2FN

Pas de dépendance fonctionnelle entre les attributs non clé

Exemple avec une entité qui contient les évaluations des films par des utilisateurs :

<u>id_film</u>	<u>id_user</u>	note	image_note
1	1	3	3etoiles.png
2	1	5	5etoiles.png
2	8	4	4etoiles.png
3	8	1	1etoiles.png

Pas en 3FN !

<u>id_film</u>	<u>id_user</u>	note		note	image_note
1	1	3		3	3etoiles.png
2	1	5		5	5etoiles.png
2	8	4		4	4etoiles.png
3	8	1		1	1etoiles.png

Pas de dépendance fonctionnelle entre les attributs non clé

En gros, si attribut pas important, ne pas mettre de dépendance

Forme normale de Boyce-Codd : BCNF

Respecter la 3FN

Les attributs non clé ne doivent pas être source de dépendance fonctionnelle vers une partie de la clé

Cela peut être interprété comme la contrainte inverse de la 2FN.

Exemple avec une entité contenant un historique des transports :

<u>id_pays</u>	<u>id_user</u>	date	transport	ville
1	1	10/10/2018	Voiture	Bruxelles
1	2	11/10/2018	Bus	Liège
3	3	11/10/2018	Avion	Boston
4	2	11/11/2018	Voiture	Paris

Dépendances fonctionnelles :

<u>id_pays</u> , <u>id_user</u>	-> date	OK
<u>id_pays</u> , <u>id_user</u>	-> transport	OK
ville	-> <u>id_pays</u>	Pas en BCNF

4. 4FN

Respecter la BCNF

Une relation est en 4FN si pour chaque dépendance multivaluée $X \rightarrow\!\!> Y$ non triviale, X est une superclé de la relation

...

Exemple avec l'entité "livres" :

Les "formes normales"
font partie de ces
choses très complexes
à formuler mais qui
sont plutôt simples à
mettre en œuvre

<u>isbn</u>	<u>auteur</u>	<u>mot_clé</u>
1-12312-323-2	Marc	Tortue
1-12312-323-2	Michel	Tortue
1-12312-323-2	Marc	Natation
1-12312-323-2	Michel	Natation
8-12312-323-4	Alice	Hiver

Pas en 4FN !



<u>isbn</u>	<u>auteur</u>
1-12312-323-2	Marc
1-12312-323-2	Michel
8-12312-323-4	Alice

<u>isbn</u>	<u>mot_clé</u>
1-12312-323-2	Tortue
1-12312-323-2	Natation
8-12312-323-4	Hiver



MCD

- Modèle Conceptuel des données

▼ Composants schéma MCD

Entités = un peu comme classe en POO

→ regroupe des infos communes à un individu

schéma = rectangle

Attributs = caractéristiques décrivant les entités

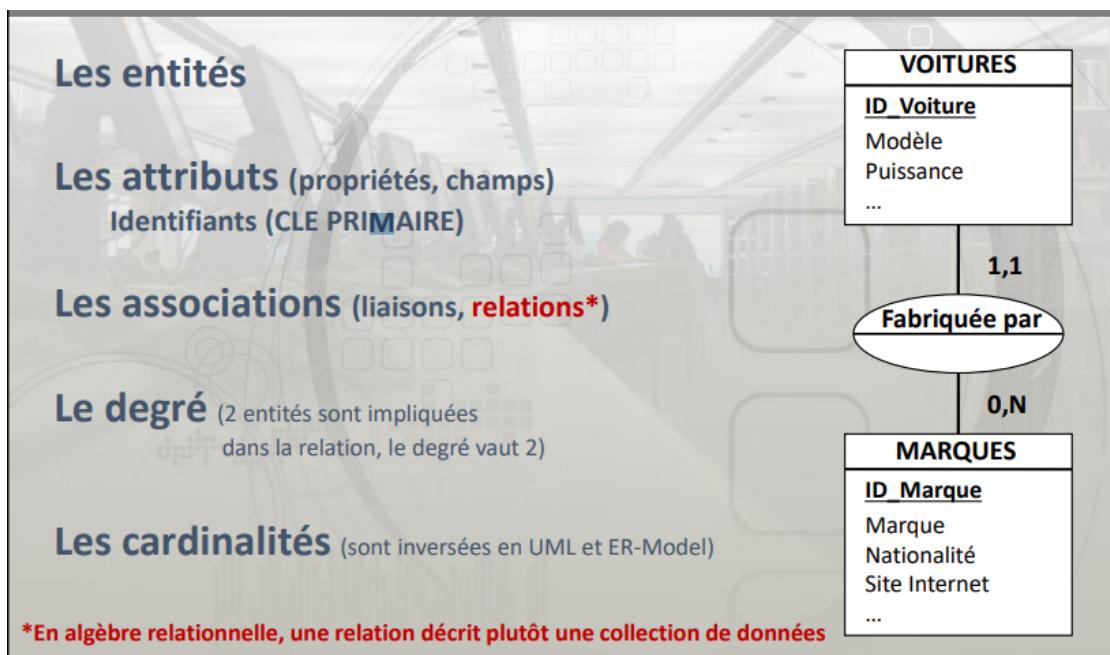
Stocker valeur

Association = liaison logique entre une ou plusieurs entités

ellipse (Merise) reliée aux entités

nombre d'entités reliées = degré

cardinalité mais sens inverse de UML



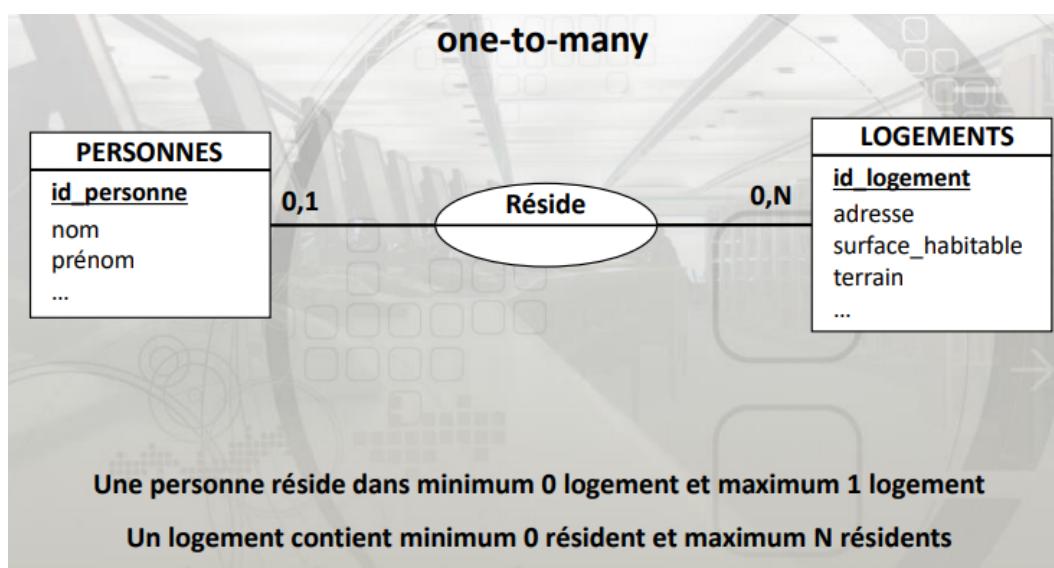
Clé primaire (PK, primary key)



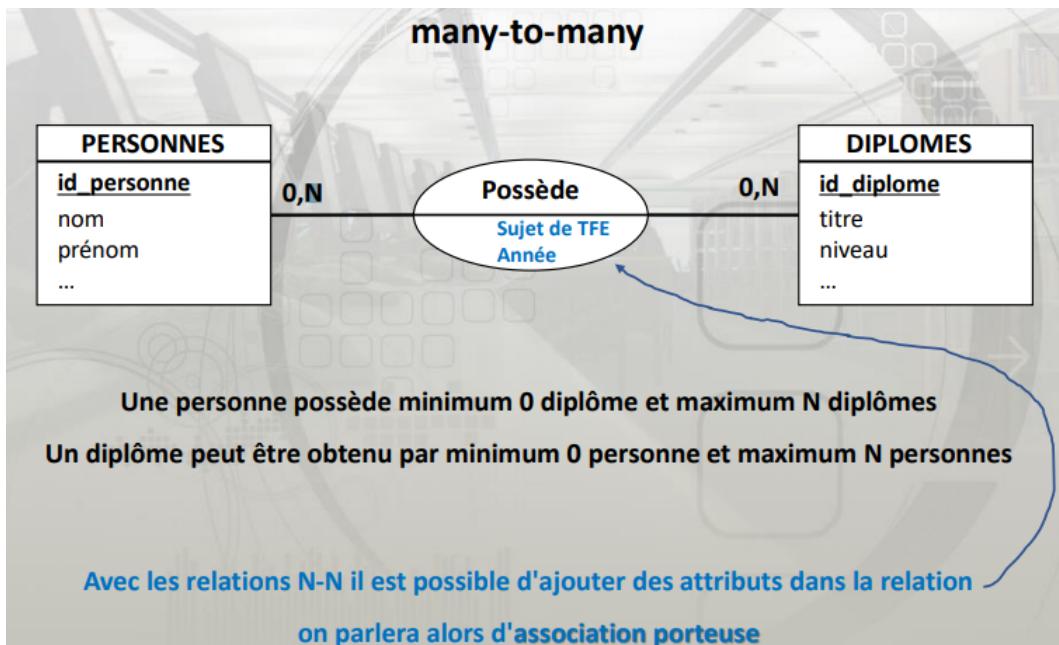
Il me semble qu'il faut toujours souligner l'ID, jsp si c que l'ID ou toutes les PK

- Propriété naturelle, doit être unique car modif ultérieurs = caca
Exemple: personne → email
- Propriété artificielle → meilleur perf
Client → ID_client
- Propriété composée
Entreprise → Enseigne + localité

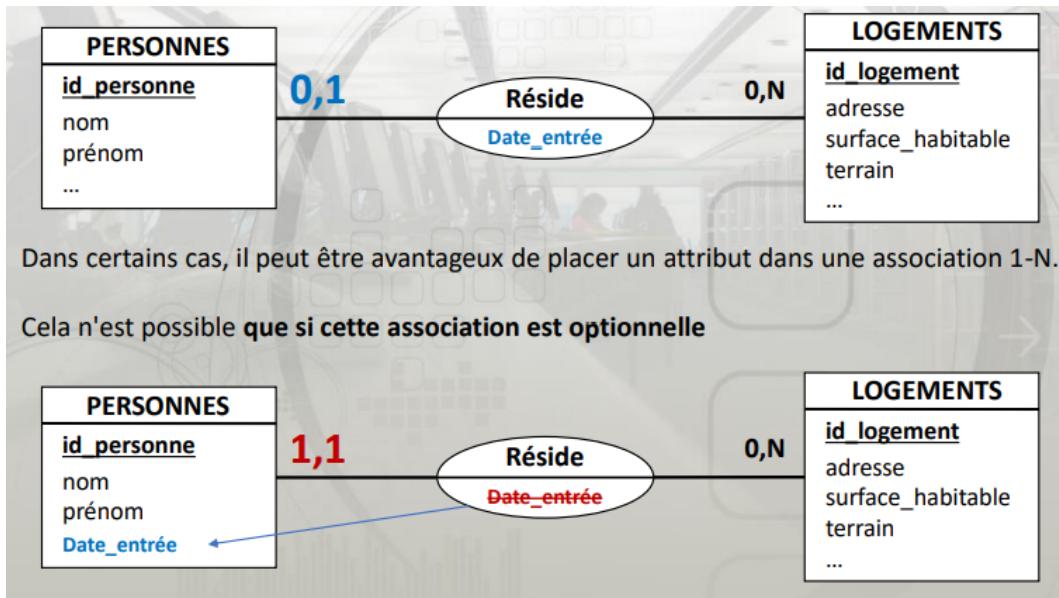
▼ Liaisons schéma



1-N



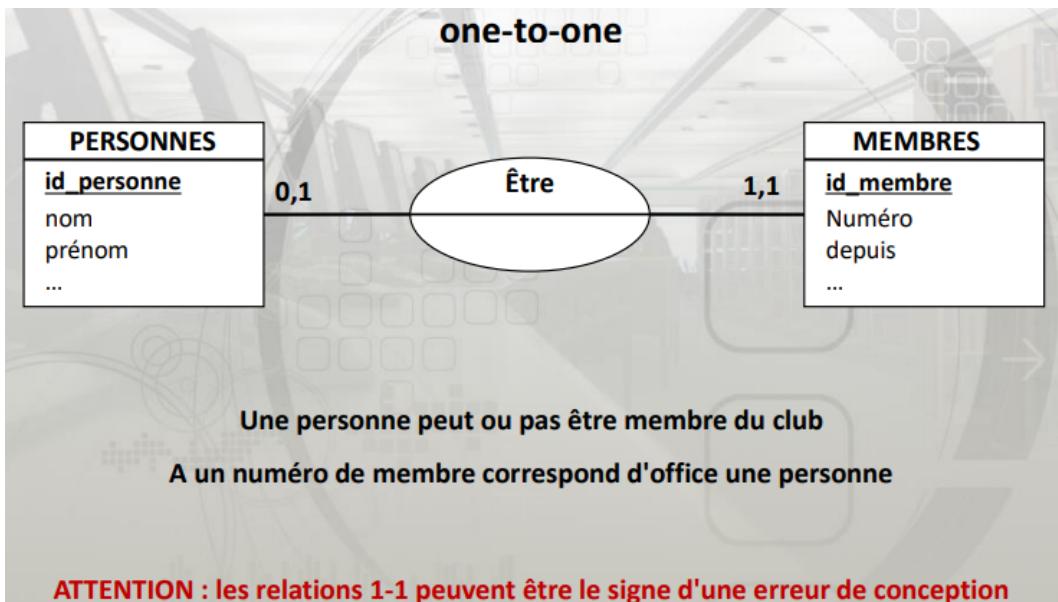
N-N



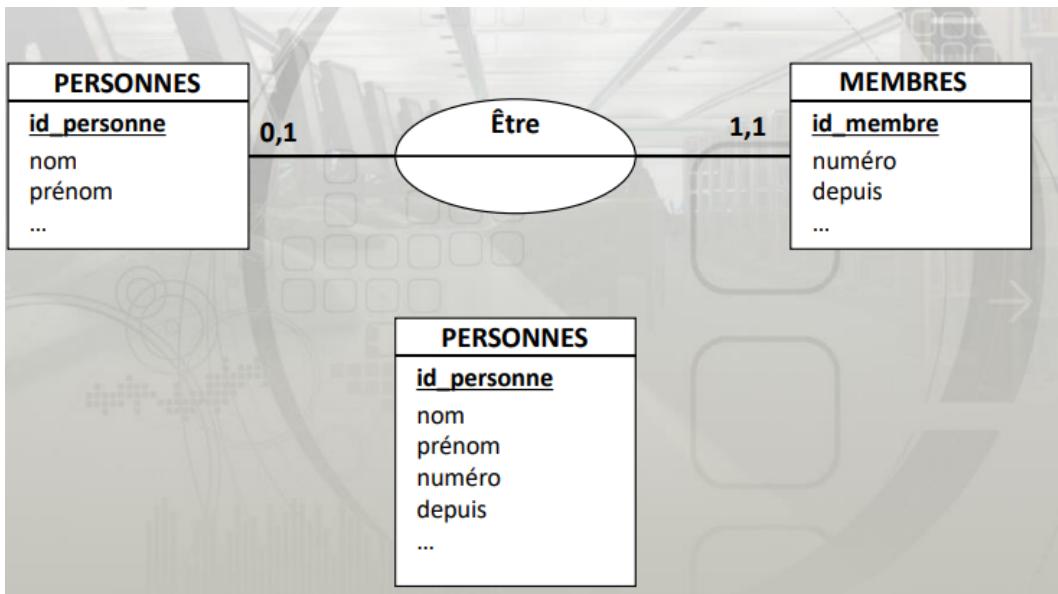
1-N



Que si association optionnelle

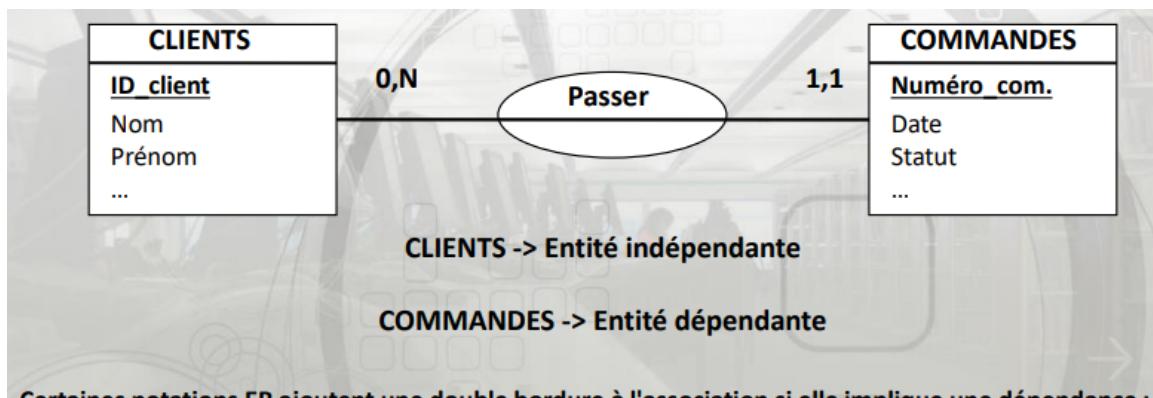


1-1

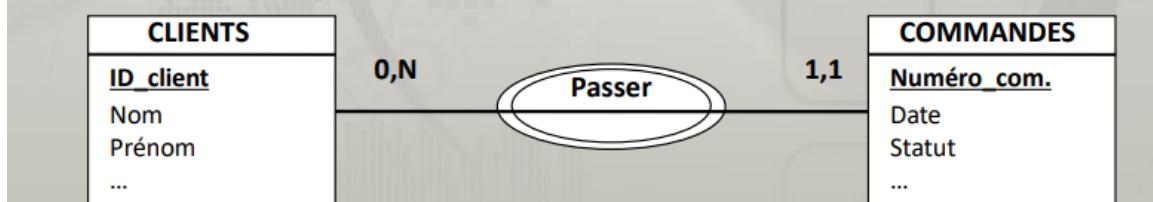


Le problème du 1-1 ici, par exemple, ici une personne et un membre peuvent être regrouper, il faut faire attention de pas faire des entités de merde pour pas compliquer le schéma pour rien

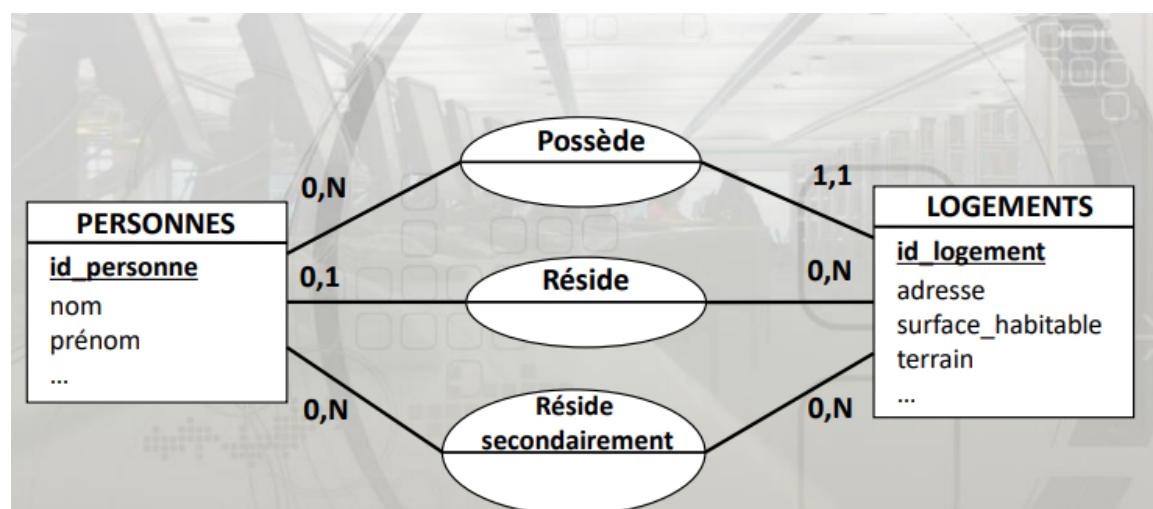
Indépendance/dépendance



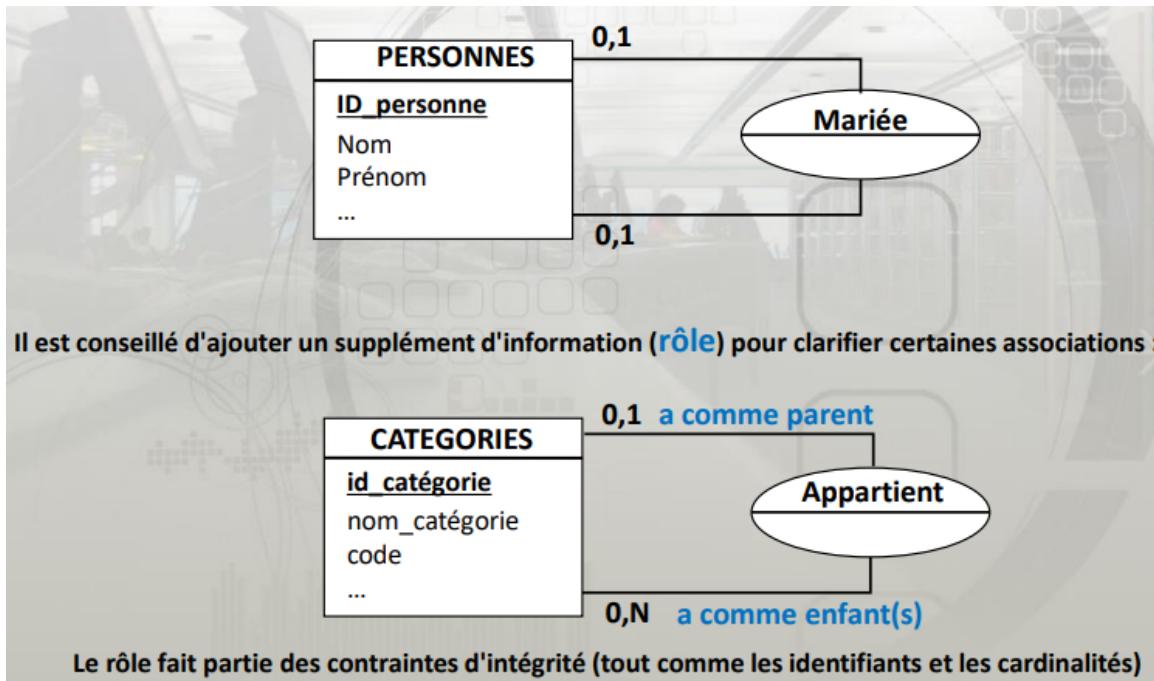
Certaines notations ER ajoutent une double bordure à l'association si elle implique une dépendance :



Association Plurielle



Association réflexive



Relation qu'un **objet (entité)** entretient avec lui-même. Autrement dit, c'est une association dans laquelle une entité est liée à d'autres occurrences de cette même entité.

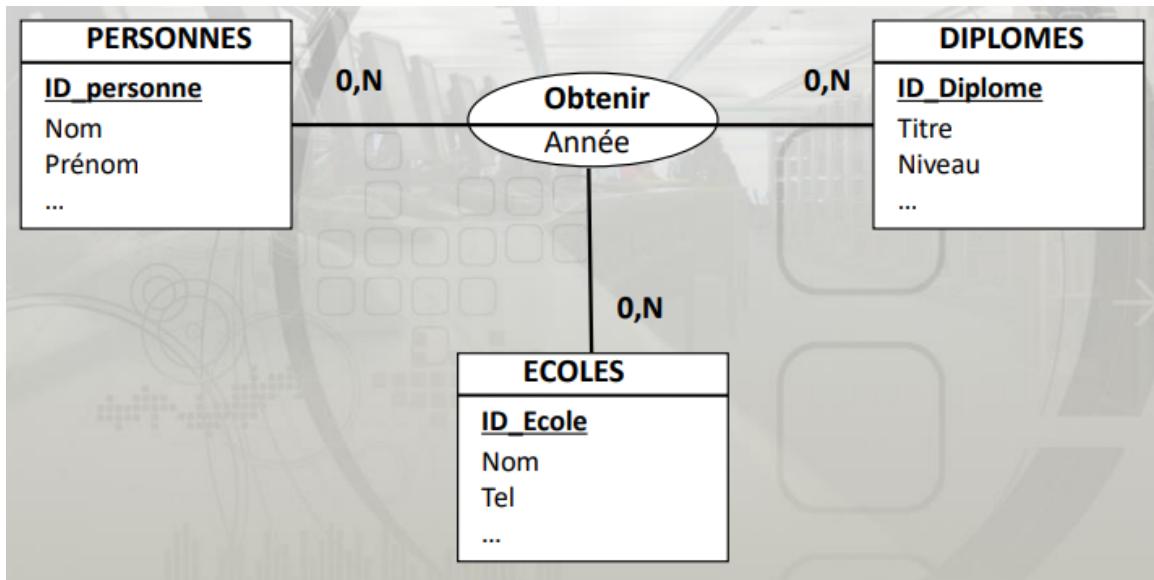
Exemple:

ID	Nom	Superviseur_ID
1	Alice	NULL
2	Bob	1
3	Charlie	1
4	Diane	2

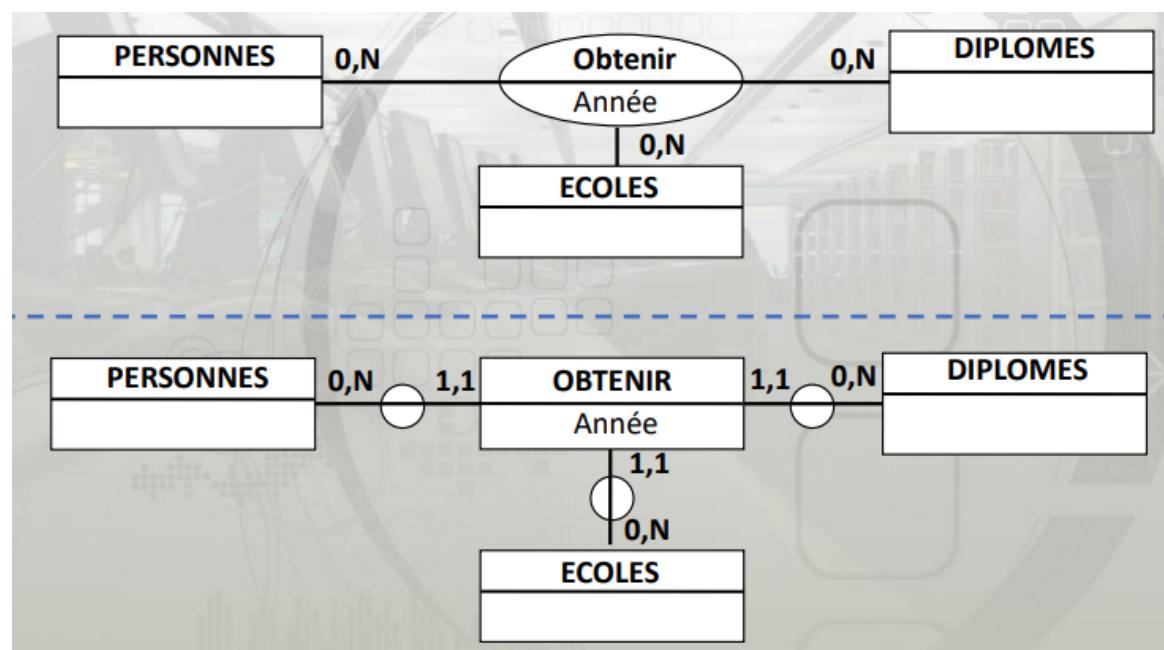
Ici :

- Alice n'a pas de superviseur.
- Bob et Charlie sont supervisés par Alice.
- Diane est supervisée par Bob.

Association ternaire



Conversion: ternaire → 3 binaires

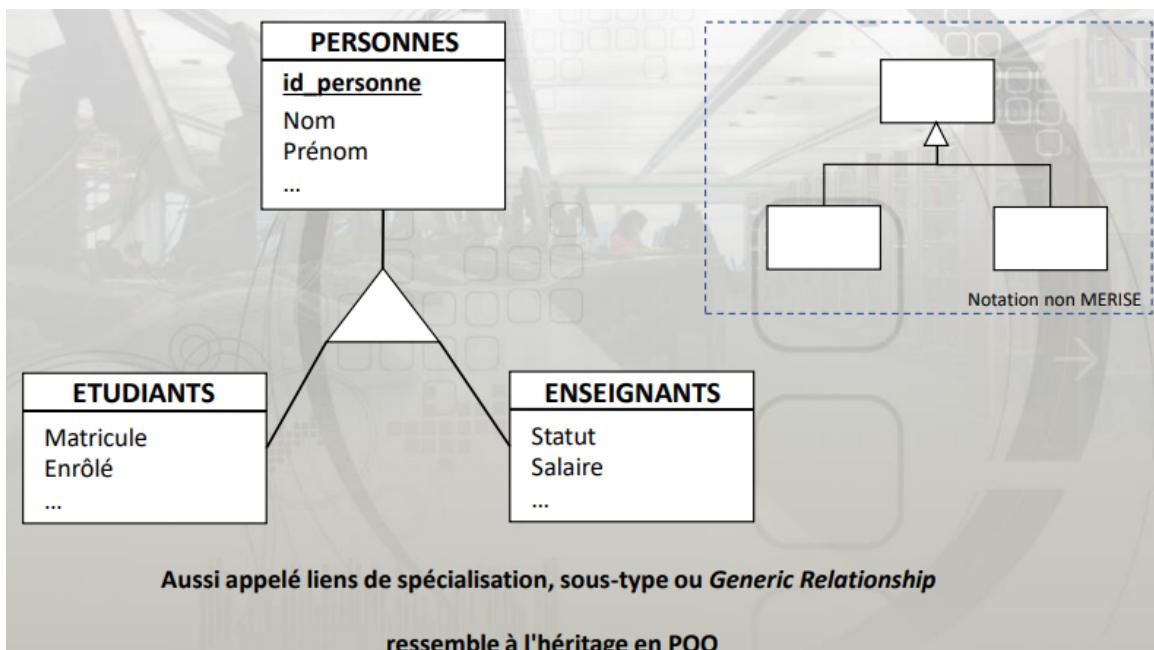


Une association ternaire n'est pas équivalente à 3 associations binaires !!!

OBTENIR					
ETUDIE		OBTENIR		DELIVRE	
Jean	Ingénieur	UMONs	2015	UMONs	Ingénieur
Sarah	Ingénieur	UMONs	2008	HEH	Electronicien
Jean	Electronicien	HEH	2012	HEH	Informaticien
Cindy	Informaticien	HEH	2012	UMONs	Informaticien
Jean	Informaticien	UMONs	2005		

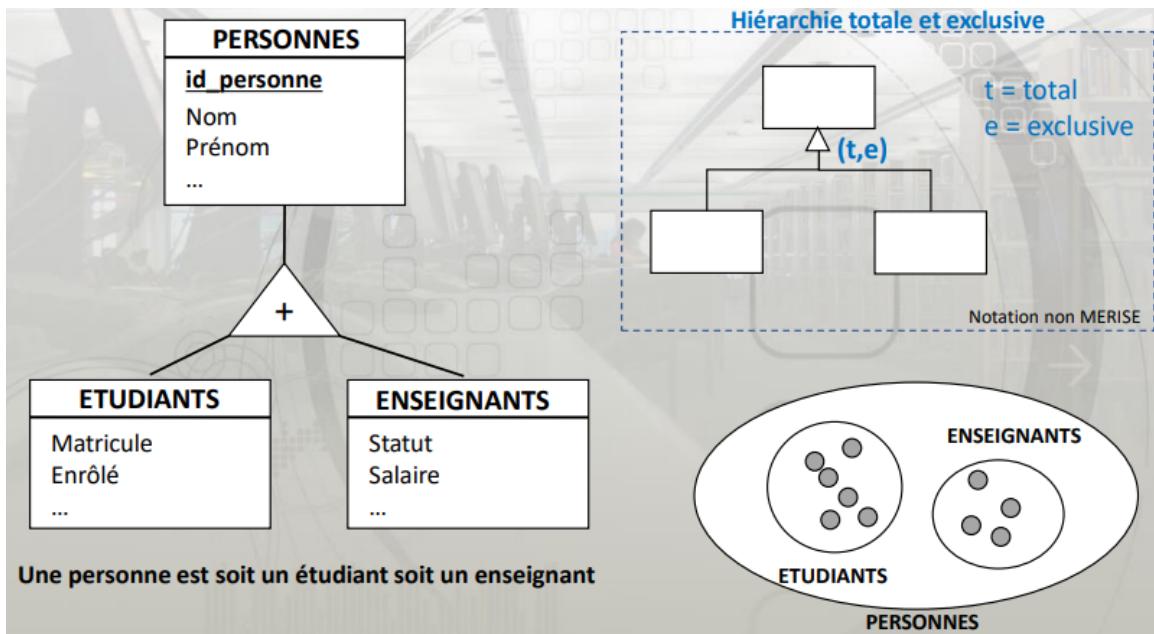
Pas tout capté comment on peut transformer l'un en l'autre mais ça a l'air d'être de la merde

▼ Liens (hiérarchie) de généralisation

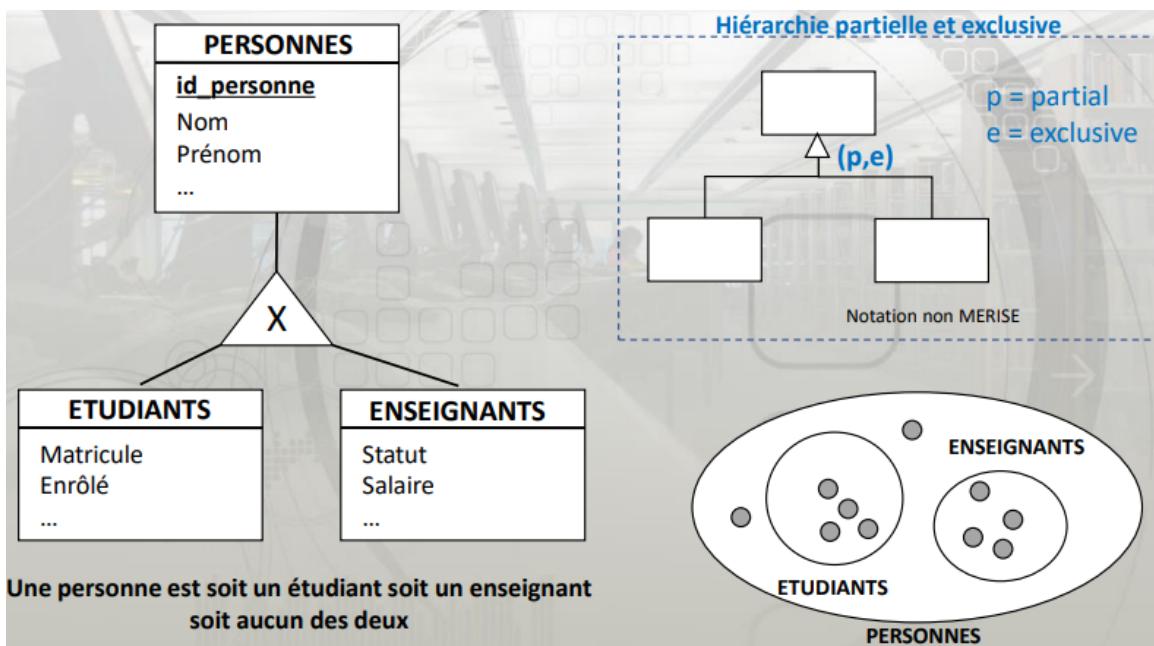


Contraintes sur les sous-types

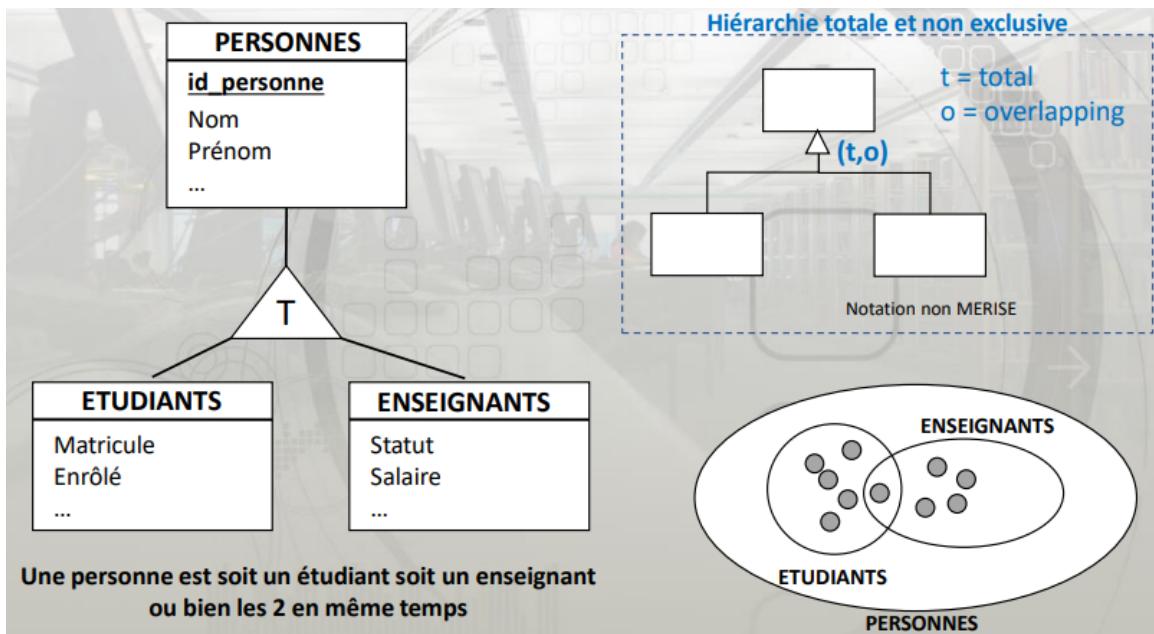
Hiérarchie totale et exclusive (+)



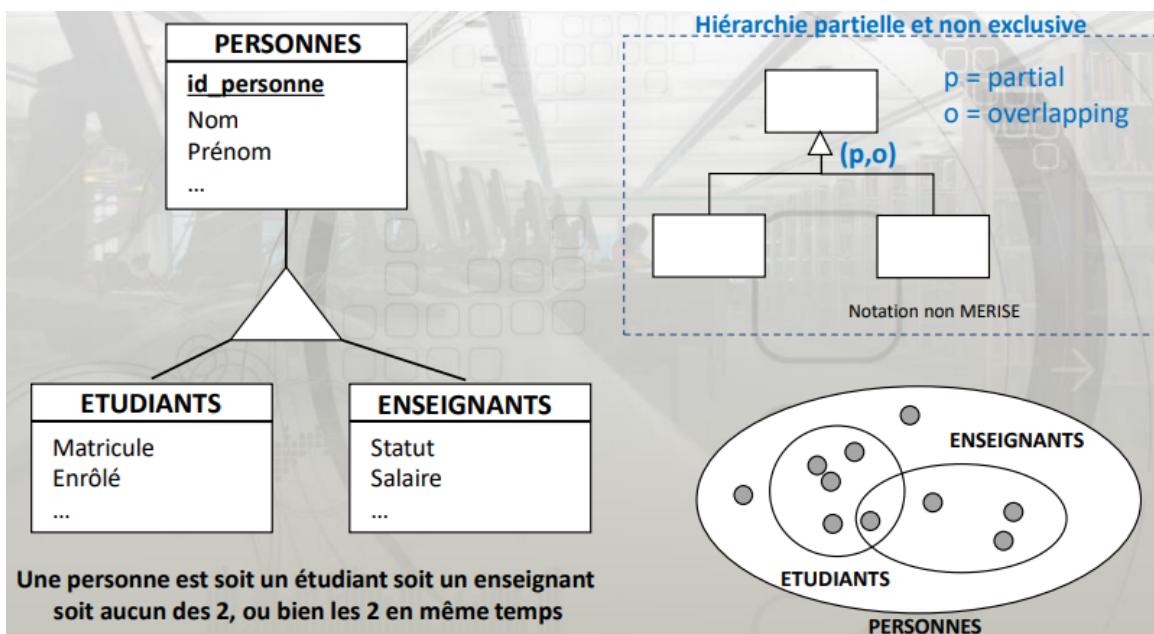
Hiérarchie partielle et exclusive (X)



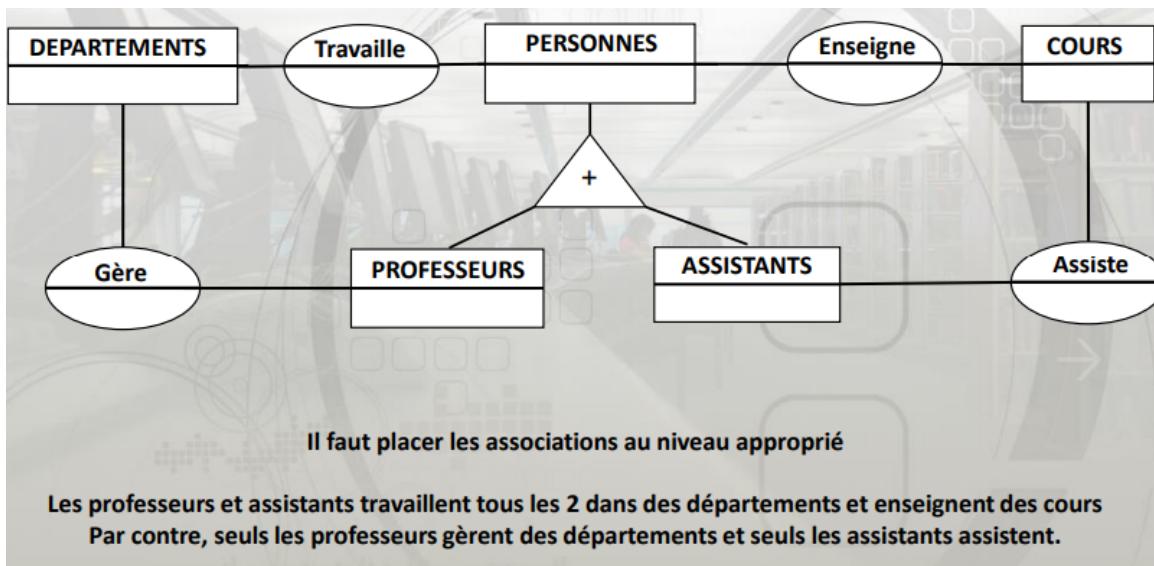
Hiérarchie totale et non exclusive (T)



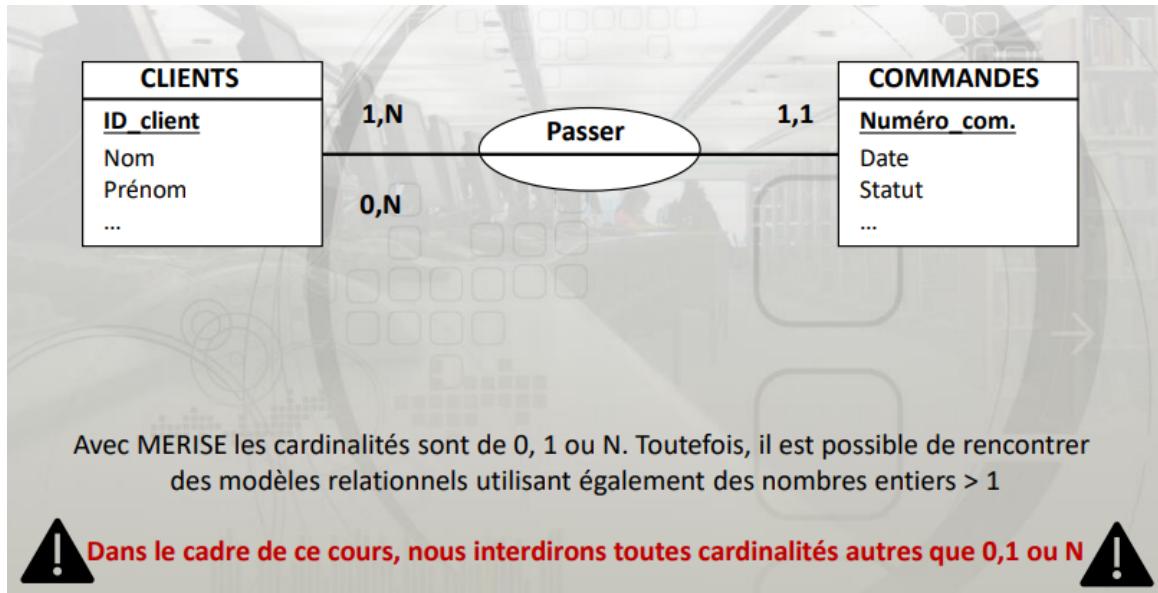
Hiérarchie partielle et non exclusive



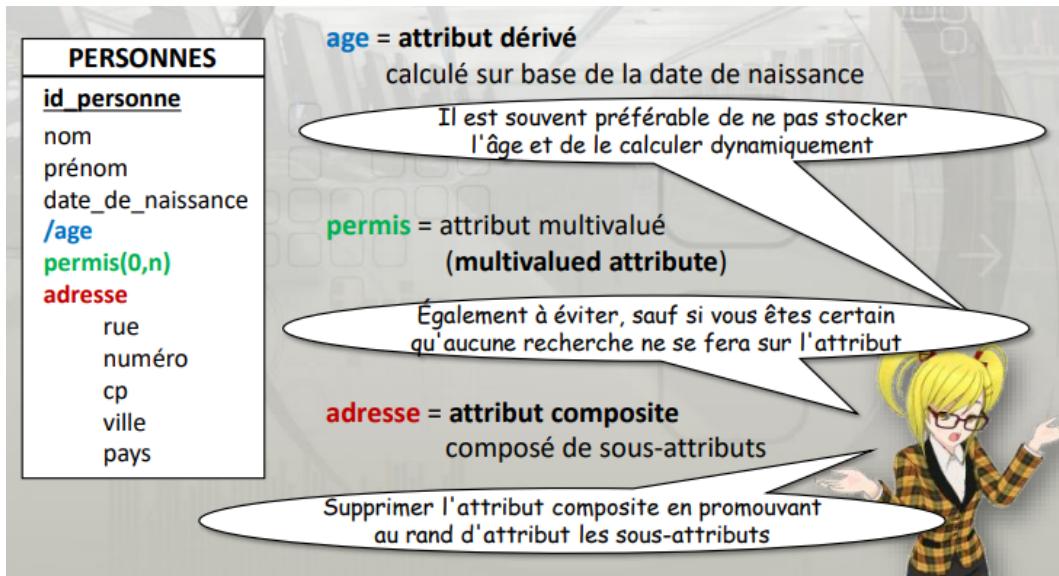
Attention avec les sous-types



▼ Cardinalité



▼ Attributs spéciaux (Modélisation ER, non Merise)



C de la merde, ne pas utiliser

MLD

- Modèle Logique des données

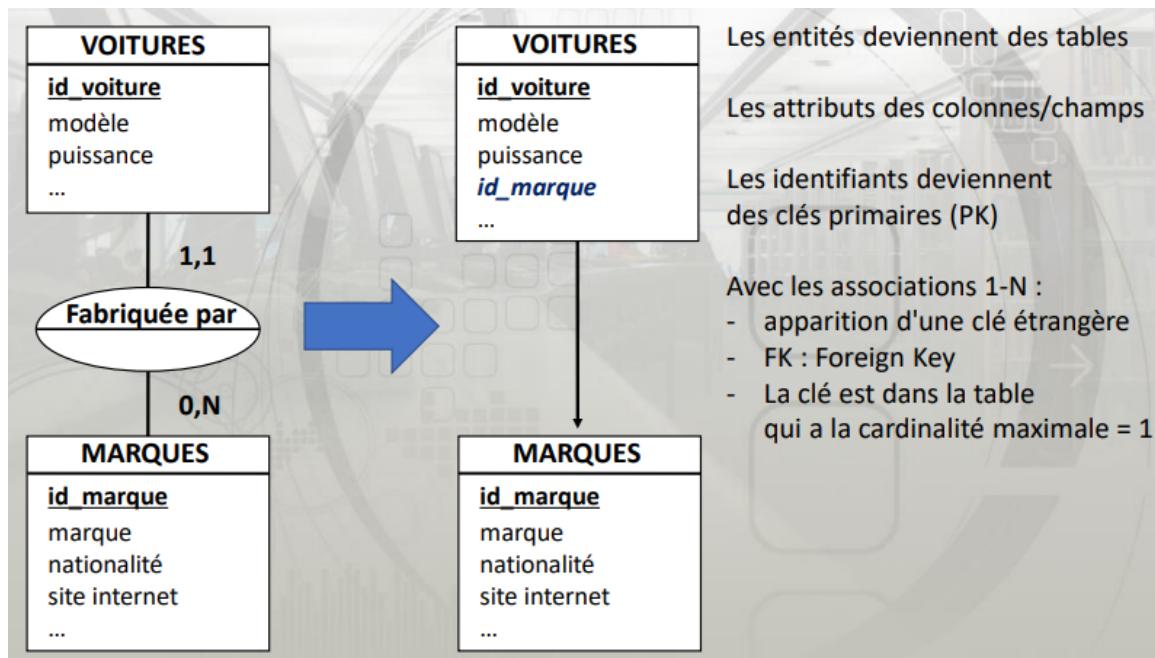


Peut être générée directement grâce à logiciel depuis un MCD, mais pour cour, tout à la main

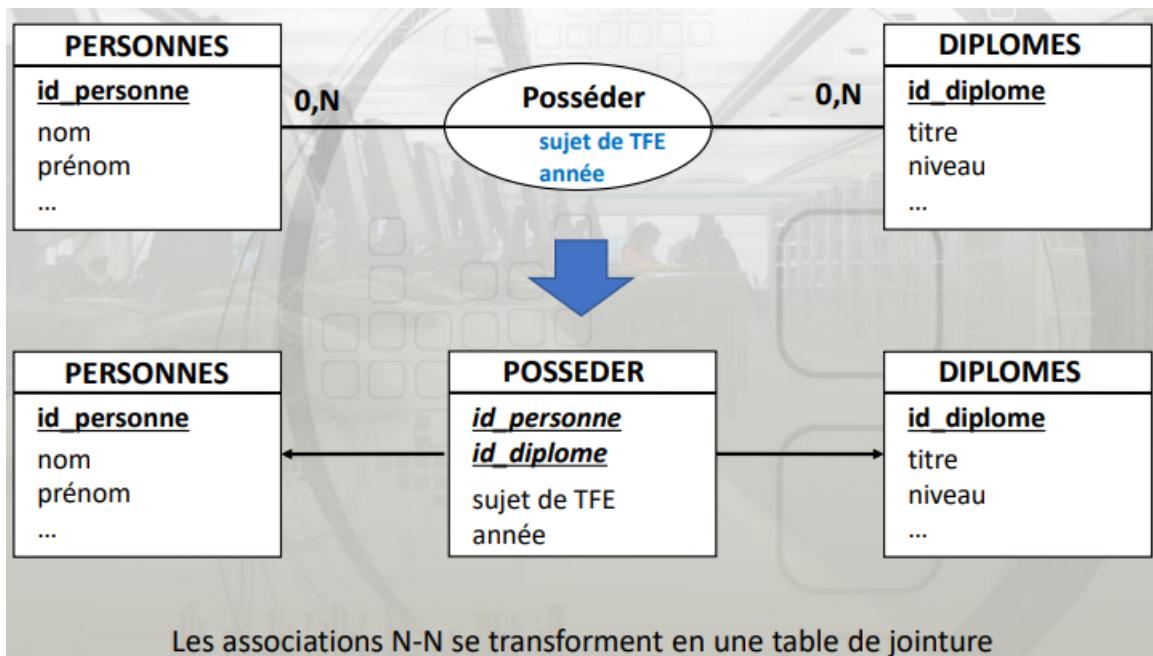
- PK = Primary key
- FK = Foreign key
 - Clé provenant d'une autre table

▼ MCD → MLD (Liens)

1-N



N-N



▼ Conseils (Tables de jointure)

Conversion MCD → MLD Merise

- Noms associations disparaissent
- Cardinalités disparaissent



Garder cardinalités et certains nom pour la clareté

Nommer tables de jointure en utilisant les noms des associations



Utiliser noms des tables jointes pour la tables de jointure

ex:

Personnes ← posséder → diplômes

Personnes ← Personnes_diplômes → diplômes

▼ Tables de jointures et PK

2 manières:

- Une clé primaire auto-incrémentée, et les clés étrangères dans la table sont de simples attributs

PERSONNES_DIPLOMES	
<i>id_personnes_diplomes</i>	
<i>id_personne</i>	
<i>id_diplome</i>	
sujet de TFE	
année	

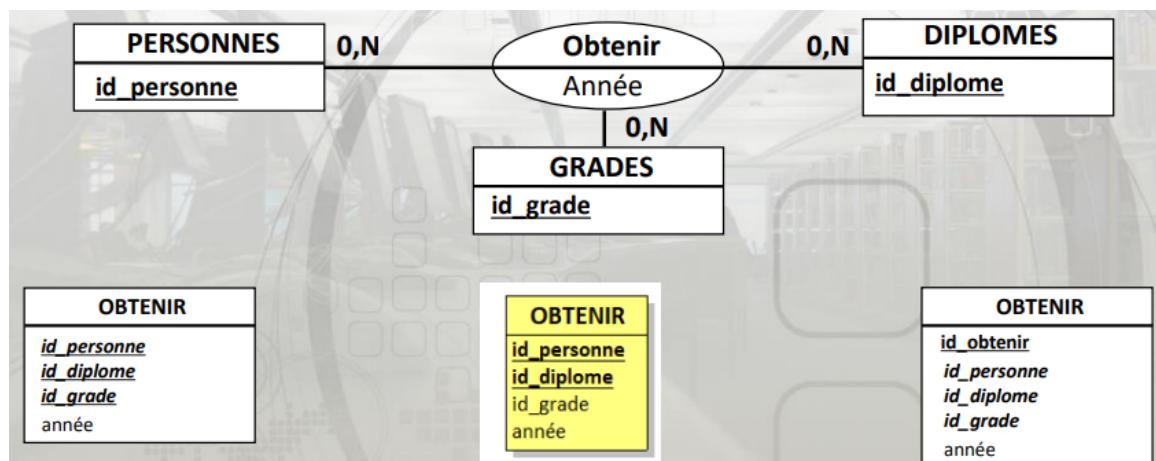
- Une clé primaire composée des clés étrangères (indexation + efficace et table + compacte)

→ Pas toujours possible

PERSONNES_DIPLOMES	
<i>id_personne</i>	
<i>id_diplome</i>	
sujet de TFE	
année	



Attributs d'une PK ne peuvent pas être vides



- Si toujours grades pour tous les diplômes, PK = plusieurs FK
- Si diplômes peut être obtenu sans grade, PK composée pas possible car une PK ne peut pas être NULL → Normal sinon il y a rien qui identifie l'entité

Les tables avec une FK, pointent vers la table qui possède la FK

→ Pareil pour les tables de jointure, pointent vers les tables qu'elles joignent



Éviter les associations 1,1-1,1 car ça provient sûrement d'une erreur de conception du MCD

→ Souvent, on peut rassembler les attributs des deux dans la même table

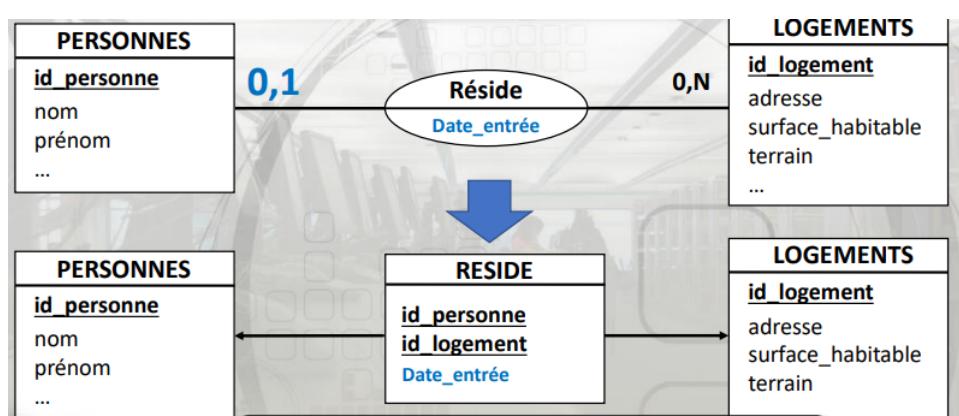


Pour 1-1

Choisir solution qui provoque le moins de NULL

Si association est **fortement** occasionnelle, table de jointure envisagée

1-N

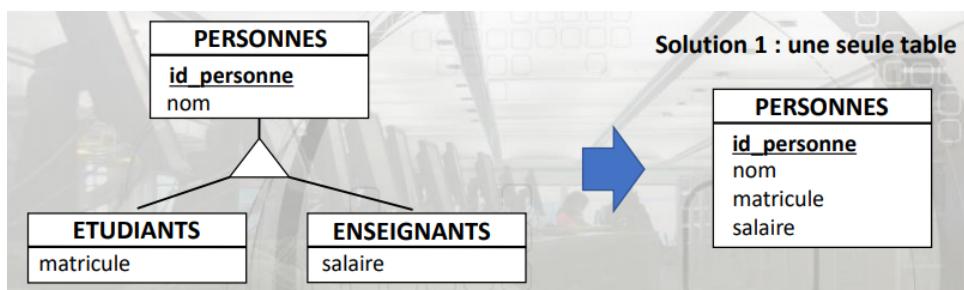


Attribut de liaison 1-N obligent à créer table supplémentaire

- Complexifie schéma
- Évite de stocker des valeurs NULL
- [Qu'avec les Associations fortement optionnelles](#)

▼ Traduction des hiérarchies de généralisation

S1: Une seule table

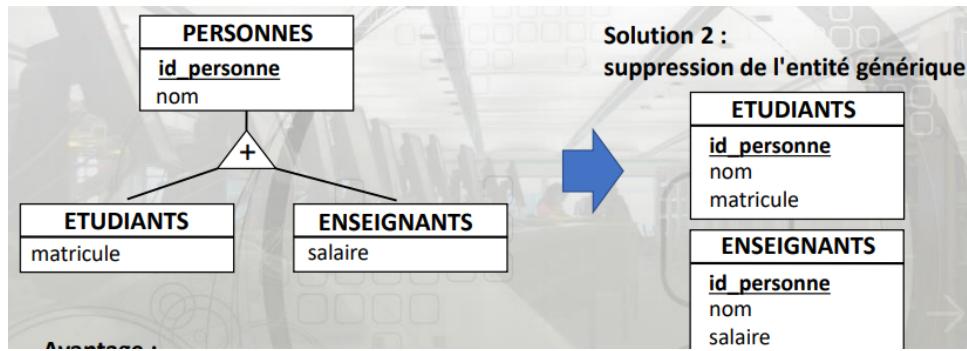


- + Simple → Simplifie schéma
- + Fonctionne pour tout type de hiérarchie
- - Potentiellement beaucoup d'attributs NULL

Privilégier:

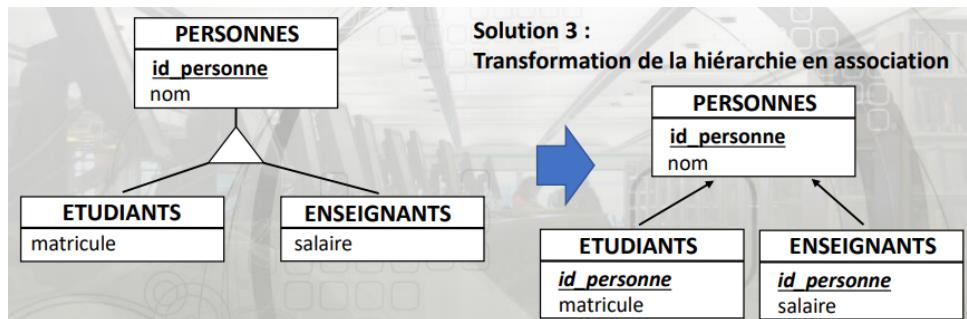
- S'il n'y a que peu d'attributs spécifiques
- Si majorité des traitements d'écriture se font simultanément sur des groupements de tuples et sur l'ensemble des attributs génériques et spécifiques

S2: Suppression de l'entité générique



- Arracher au sol, pas d'avantages
 - Ne fonctionne qu'avec une généralisation totale et exclusive (+)
 - Cela peut compliquer le système (if else if else...)
 - Perte de sémantique (la notion de PERSONNES)
- Préférer uniquement si majorité des traitements accèdent simultanément aux attributs génériques et spécifiques

S3: Transformation de la hiérarchie en association



- + Fonctionne pour tout type de hiérarchie
 - + Pas de perte de sémantique
 - - Le schéma résultant est plus complexe qu'avec l'approche 1
- Approche à privilégier sauf pour certains cas bien précis, ceux du dessus j'imagine

Normalisation

Dépendance fonctionnelle

ARTICLES	
<u>id_article</u>	
nom	
référence	
prix	
numéro_catégorie	
libellé_catégorie	
Emmental	
4578912	
1,25 €	
4	
Fromage	
Brie	
1256912	
0,98 €	
4	
Fromage	
Pink Lady	
4568941	
0,85 €	
7	
Fruits	

On dit qu'il existe une dépendance fonctionnelle entre un attribut A1 et un attribut A2, on note A1 → A2 si, connaissant une valeur de A1, on ne peut lui associer qu'une seule valeur de A2.

On dit aussi que A1 détermine A2.

A1 est la source de la dépendance fonctionnelle et A2 le but.

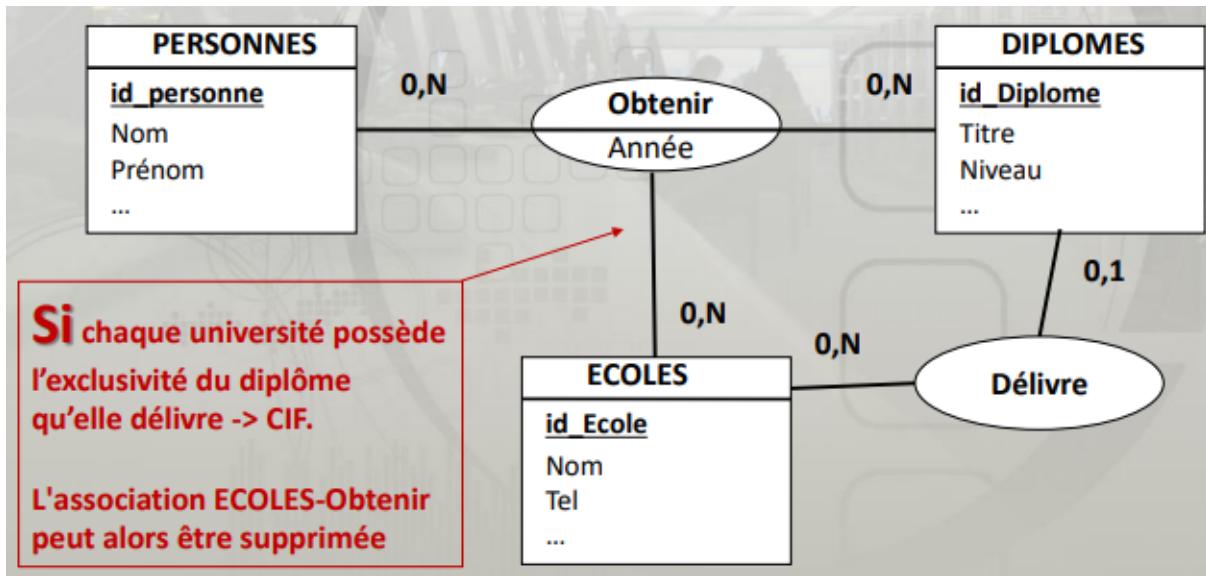
A1 = source de dépendance

A2 = but

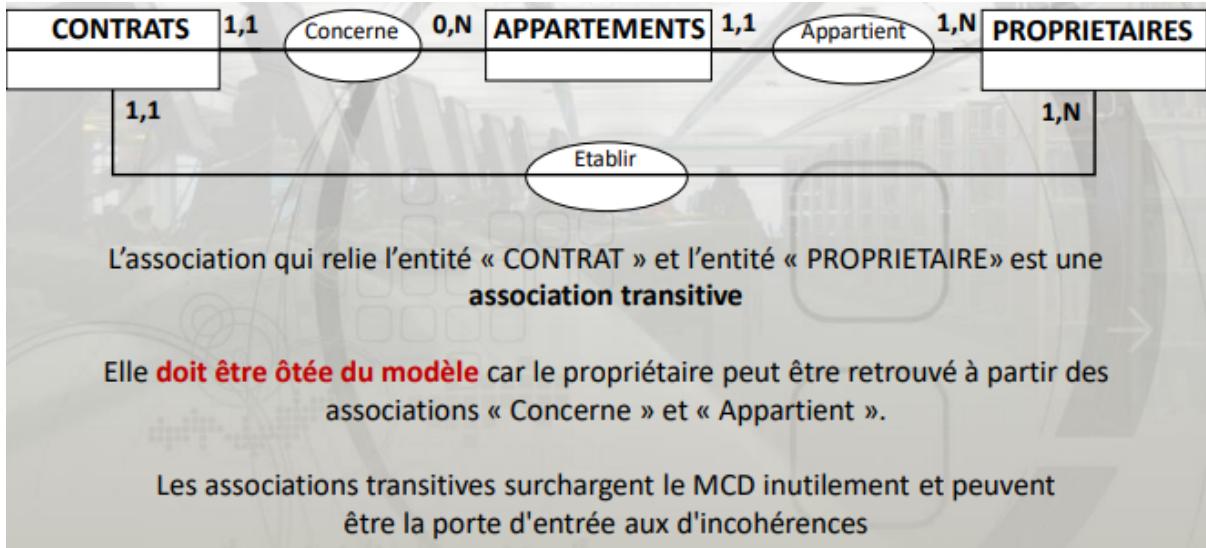
Avec A1 on connaît la valeur de A2

Contrainte d'intégrité fonctionnelle

Définit par le fait que l'une des entités participant à l'association est complètement déterminée par la connaissance d'une ou de plusieurs autres entités participant dans cette même association



Association transitive

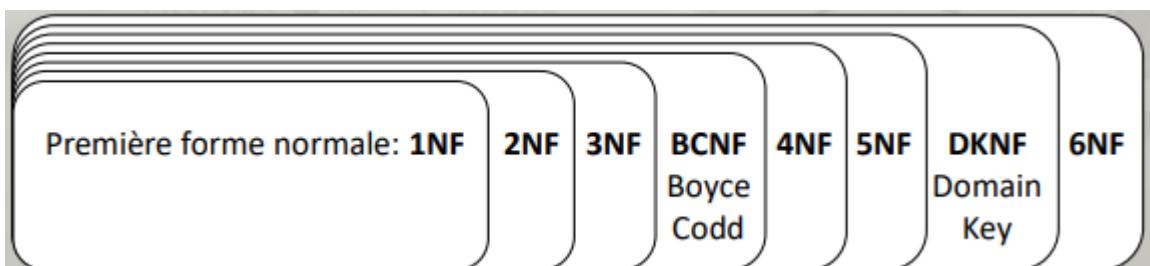


Pas en mettre car surcharge MCD, avec les liens on sait déjà retrouvé qui et qui donc inutile

Normalisation

Quoi :

- Règles de bonne pratique
But :
- Améliorer la modélisation
- Éviter les redondances des données (éviter les problèmes de mise à jour)
Contexte :
- Principalement OLTP
- Niveaux :



1. 1FN

id	nom	email
1	M. Frank Dewit	f.dewit@gmail.com, zedewit@hotmail.com
2	M. Ivan Raoul	rivan@hotmail.fr
3	Mme Leen Breckx	Leen.breckx@gmail.com, brekyleen@live.fr

Les attributs doivent être atomiques :

- Supprimer les attributs composites en promouvant au rang d'attribut les sous-attributs
- Les attributs multivalués donnent naissance à de nouvelles entités

id	titre	prenom	nom	id	email
1	M.	Frank	Dewit	1	f.dewit@gmail.com
2	M.	Ivan	Raoul	2	zedewit@hotmail.com
3	Mme	Leen	Breckx	3	rivan@hotmail.fr

0,N 1,1

id	email
1	f.dewit@gmail.com
2	zedewit@hotmail.com
3	rivan@hotmail.fr
4	Leen.breckx@gmail.com
5	brekyleen@live.fr

Attributs atomiques

2. 2FN

Respecter la 1FN

Les attributs doivent dépendre complètement de la clé et non partiellement

Ce cas se rencontre principalement avec les clés composées.

Exemple avec une entité qui contient les évaluations des films par des utilisateurs :

<u>id_film</u>	<u>id_user</u>	note	genre
1	1	3	Comédie
2	1	5	Action
2	8	4	Action
3	8	1	Aventure

Pas en 2FN !

L'attribut "genre" est propre au film (id_film) et ne dépend pas de l'utilisateur (id_user), cet attribut n'est pas dans la bonne entité.

Attribut dépendre complètement de la clé et non partiellement

3. 3FN

Respecter la 2FN

Pas de dépendance fonctionnelle entre les attributs non clé

Exemple avec une entité qui contient les évaluations des films par des utilisateurs :

<u>id_film</u>	<u>id_user</u>	note	image_note
1	1	3	3etoiles.png
2	1	5	5etoiles.png
2	8	4	4etoiles.png
3	8	1	1etoiles.png

Pas en 3FN !

<u>id_film</u>	<u>id_user</u>	note		note	image_note
1	1	3		3	3etoiles.png
2	1	5		5	5etoiles.png
2	8	4		4	4etoiles.png
3	8	1		1	1etoiles.png

Pas de dépendance fonctionnelle entre les attributs non clé

En gros, si attribut pas important, ne pas mettre de dépendance

Forme normale de Boyce-Codd : BCNF

Respecter la 3FN

Les attributs non clé ne doivent pas être source de dépendance fonctionnelle vers une partie de la clé

Cela peut être interprété comme la contrainte inverse de la 2FN.

Exemple avec une entité contenant un historique des transports :

<u>id_pays</u>	<u>id_user</u>	date	transport	ville
1	1	10/10/2018	Voiture	Bruxelles
1	2	11/10/2018	Bus	Liège
3	3	11/10/2018	Avion	Boston
4	2	11/11/2018	Voiture	Paris

Dépendances fonctionnelles :

<u>id_pays</u> , <u>id_user</u>	-> date	OK
<u>id_pays</u> , <u>id_user</u>	-> transport	OK
ville	-> <u>id_pays</u>	Pas en BCNF

4. 4FN

Respecter la BCNF

Une relation est en 4FN si pour chaque dépendance multivaluée $X \rightarrow\!\!> Y$ non triviale, X est une superclé de la relation

...

Exemple avec l'entité "livres" :

Les "formes normales"
font partie de ces
choses très complexes
à formuler mais qui
sont plutôt simples à
mettre en œuvre

<u>isbn</u>	<u>auteur</u>	<u>mot_clé</u>
1-12312-323-2	Marc	Tortue
1-12312-323-2	Michel	Tortue
1-12312-323-2	Marc	Natation
1-12312-323-2	Michel	Natation
8-12312-323-4	Alice	Hiver

Pas en 4FN !



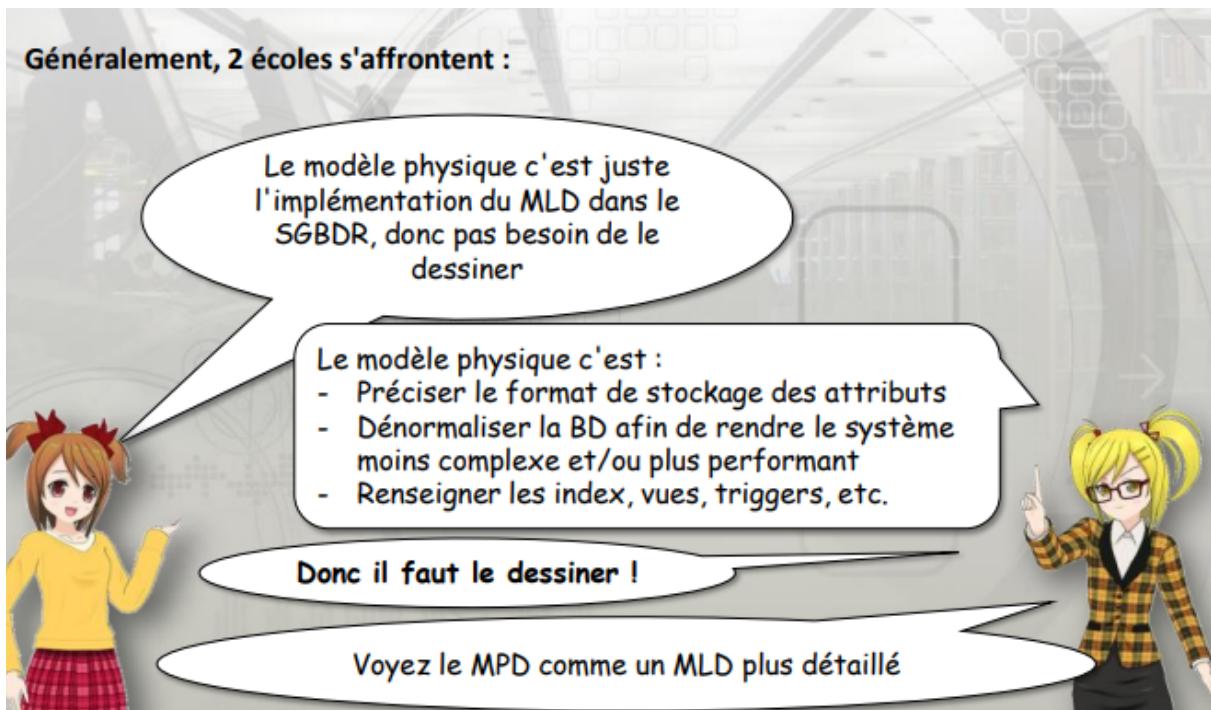
<u>isbn</u>	<u>auteur</u>
1-12312-323-2	Marc
1-12312-323-2	Michel
8-12312-323-4	Alice

<u>isbn</u>	<u>mot_clé</u>
1-12312-323-2	Tortue
1-12312-323-2	Natation
8-12312-323-4	Hiver



MPD

Modèle physique de données



Fonctionnalités avancées - PN maison

Indexation

Recherche sans index

The diagram illustrates a search operation without an index. On the left, there is a table of client data:

id	nom	prénom	tel
1	Green	Roby	0471 123 123
2	Blue	Will	0486 123 123
3	Pink	Jen	0470 123 123
4	Red	Sarah	0477 123 123
5	Pink	Elise	0473 123 123
6	White	John	0499 123 123
7	Black	Jack	0494 123 123
...	...		

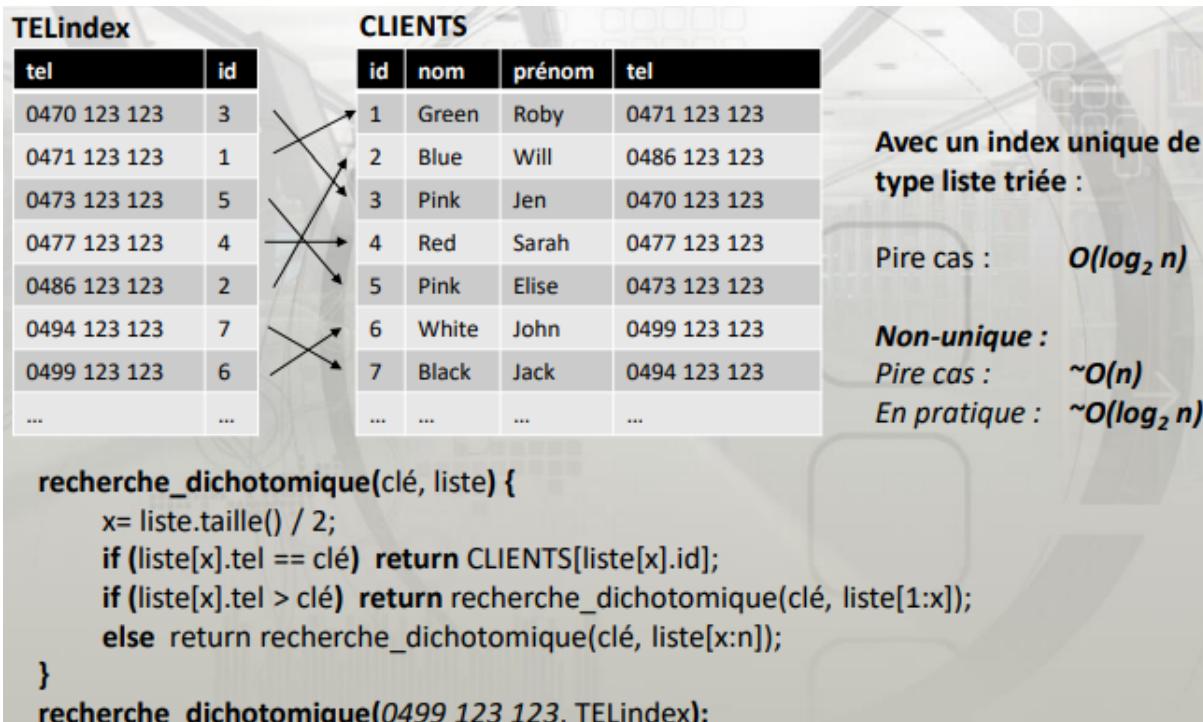
A speech bubble contains the question: "Qui à le numéro 0499 123 123".

On the right, there is a code snippet in pseudocode:

```
rep array()
foreach (CLIENTS as CLIENT) {
    if (CLIENT[tel]==0494 123 123)
        rep.add (CLIENT[id])
}
```

The complexity is labeled as $O(n)$.

Avec index

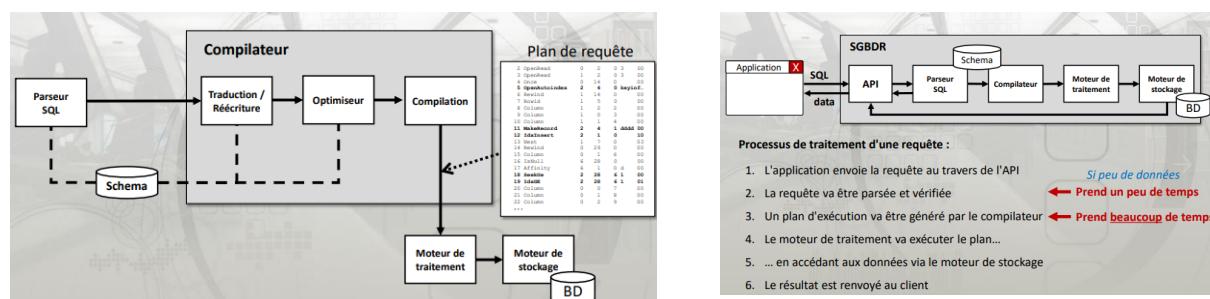


Avantages des index :

- Accélèrent (considérablement) les recherches, jointures et agrégations
- Inconvénients :
- Occupent de l'espace
- Ralentissent les mises à jour (INSERT, UPDATE et DELETE)
- Différentes structures d'indexation (B-Arbre, Hash, bitmap)

Requêtes préparées

Architecture d'une SGBDR



L'idée première des requêtes préparées :

- Court-circuiter la phase d'analyse du parseur ainsi que toutes les étapes du compilateur

Processus de traitement d'une requête préparée :

Phase 1 :

1. L'application envoie la requête au travers de l'API
2. La requête va être parsée et vérifiée
3. Un plan d'exécution va être généré par le compilateur
4. Sauvegarde du plan et envoi d'un identifiant de requête au client

Phase 2 :

5. Le client envoie l'identifiant de plan accompagné d'éventuels paramètres
6. Le moteur de traitement va exécuter le plan...
7. ... en accédant aux données via le moteur de stockage
8. Le résultat est renvoyé au client

Préparation d'une requête :

`PREPARE test FROM 'SELECT * FROM matable WHERE ID=? AND truc>?';`

Les '?' présents dans la requête représentent des paramètres.

Déclaration des variables :

`SET @x=104 ;
SET @y=18 ;`

Exécution de la requête :

`EXECUTE test USING @x, @y ;`

Gros avantage des requêtes préparées : **seul le code SQL préparé est compilé.**

Si les paramètres envoyés après compilation contiennent du code SQL malicieux, il ne sera pas exécuté !

Bien utilisées, elles permettent donc d'éviter les injections SQL



Exemple PDO:

```

$connexion = new PDO("...", "...", "...");

$query = "INSERT INTO clients (nom, prenom, blabla, quand)
VALUES (:nom, :prenom, :blabla, NOW() )";

$result = $connexion->prepare($query);

$result->bindValue("nom", $x);
$result->bindValue("prenom", $y);
$result->bindValue("blabla", $z);

$result->execute();

```

ATTENTION : une requête préparée est stockée en session utilisateur

```

# Pour supprimer une requête préparée :
DROP PREPARE test ;
# OU
DEALLOCATE PREPARE test ;

```

Avantages et inconvénients

 Avantages des requêtes préparées <ul style="list-style-type: none"> 1. Protection contre les injections SQL 2. Accélèrent les scripts qui exécutent une même requête en boucle 3. Peuvent réduire sensiblement la charge réseau pour les scripts du point ci-dessus <p>Ce genre de script est peu courant dans un système bien conçu.</p>  Inconvénients des requêtes préparées <ul style="list-style-type: none"> 1. Légère baisse de réactivité 2. Légère augmentation de la charge réseau 	<div style="border-left: 2px solid #ccc; padding-left: 10px; margin-bottom: 10px;"> Virtuellement 2 interactions client/serveur pour une seule requête </div>
--	--

Vues

VENEDEURS		RECETTES	
<u>Id_vendeur</u>		<u>id_vendeur (FK)</u>	
nom		<u>date_recette</u>	
prenom		montant	

Nous voulons **créer une VUE** pour récupérer le total des recettes par vendeur et par mois.
Nous voulons obtenir les colonnes 'année', 'mois', 'nom', 'prénom' et 'recettes' :

CREATE VIEW recettes_mensuelles_vendeurs **AS**

```
SELECT YEAR( r.date_recette ) AS 'annee', MONTH( r.date_recette ) AS 'mois',
       v.id_vendeur, v.nom, v.prenom, SUM( r.montant ) AS 'recettes'
  FROM (recettes AS r ) JOIN (vendeurs AS v) ON (v.id_vendeur=r.id_vendeur)
 GROUP BY annee, mois, r.id_vendeur
```

La vue s'utilise ensuite comme une table :

```
SELECT * FROM recettes_mensuelles_vendeur WHERE annee=2019
```

S'utilise comme une table

Utilité vue

Utilité principale :

- Méthode d'abstraction qui permet de rendre l'application agnostique du schéma

Utilité secondaire :

- Offre un mécanisme de confidentialité supplémentaire en permettant de masquer des colonnes à un compte qui n'aurait accès qu'aux vues.

Fonctionnalités souvent méconnue :

- Il est possible de mettre à jour les tables si la requête originelle n'est pas ambiguë

- S'il est possible d'effectuer des mises à jour et que la requête originelle n'est "vraiment" pas ambiguë, il peut également être possible d'effectuer des insertions. (exemple : toutes les colonnes non-nulles et sans valeur par défaut doivent être présentes dans la requête)

Vues peuvent être parfois plus lentes

Pour l'exemple précédent, MySQL et MariaDB traitent la vue en 2 temps :

1. la requête source de la vue est exécutée;
2. le filtre est appliqué sur le résultat.

PostgreSQL, quant à lui, va procéder autrement :

1. le filtre va être ajouté à la requête initiale;
2. cette "nouvelle" requête va être exécutée.

MySQL et MariaDB, pas d'index qui peut être utilisé par le filtre, PostgreSQL, génère nouveau plan de requête optimisé

→ ça tue les vues indexées, mais possible dans rien sauf oracleDB, et SQL server

Vue matérialisée

Une vue matérialisée : remplacer la vue par une vraie table

Trop de redondance, respecte pas FN, mais au moins c'est réactif

Utilisation de triggers pour faire MAJ de donnée et MAJ des vues matérialisées

Triggers

Un déclencheur (ou Trigger) est un code que le SGBD doit exécuter dès qu'un événement particulier se produit.

Les déclencheurs peuvent être déclenchés (...) à partir des commandes DML suivantes :

- INSERT
- UPDATE
- DELETE



Pas de déclenchement à la suite des DDL (DROP ou TRUNCATE)

Trigger se fait:

- Soit avant ou après un INSERT, UPDATE ou DELETE
- Soit une fois par ligne modifiée, soit une fois par requête SQL

Nous désirons une vue matérialisée pour cette requête :

```
SELECT YEAR( r.date_recette ) AS 'annee', MONTH( r.date_recette ) AS 'mois',
       v.id_vendeur, SUM( r.montant ) AS 'recettes'
  FROM (recettes AS r ) JOIN (vendeurs AS v) ON (v.id_vendeur=r.id_vendeur)
 GROUP BY annee, mois, r.id_vendeur
```

1. On crée la table :

```
CREATE TABLE vuemat_recettes (
    annee INTEGER NOT NULL,
    mois INTEGER NOT NULL,
    id_vendeur INTEGER NOT NULL,
    recettes DECIMAL(10,2) NOT NULL,
    PRIMARY KEY(annee, mois, id_vendeur),
    INDEX(id_vendeur)
);
```

2. On crée le trigger qui devra se déclencher pour chaque insertion dans la table recettes :

```
DELIMITER |
CREATE TRIGGER trigger_insert_recettes
BEFORE INSERT ON recettes
FOR EACH ROW
BEGIN
    INSERT INTO vuemat_recettes (annee, mois, id_vendeur, recettes)
    VALUES (YEAR(NEW.date_recette), MONTH(NEW.date_recette),
            NEW.id_vendeur, NEW.montant)
    ON DUPLICATE KEY
        UPDATE recettes= recettes + NEW.montant;
END |
```

RECETTES
id_vendeur (FK)
date_recette
montant

VUEMAT_RECETTES
annee
mois
id_vendeur (FK)
recettes

3. On crée le trigger qui devra se déclencher pour chaque UPDATE pour la table recettes :

```
DELIMITER |
CREATE TRIGGER trigger_update_recettes
BEFORE UPDATE ON recettes
FOR EACH ROW
BEGIN
    UPDATE vuemat_recettes
        SET recettes= recettes + (NEW.montant - OLD.montant)
        WHERE annee=YEAR(NEW.date_recette)
              AND mois=MONTH(NEW.date_recette)
              AND id_vendeur=NEW.id_vendeur;
END |
DELIMITER ;
```

RECETTES
id_vendeur (FK)
date_recette
montant

VUEMAT_RECETTES
annee
mois
id_vendeur (FK)
recettes

4. On crée le trigger qui devra se déclencher pour chaque DELETE pour la table recettes :

```
DELIMITER |
CREATE TRIGGER trigger_delete_recettes
BEFORE DELETE ON recettes
FOR EACH ROW
BEGIN
    UPDATE vuemat_recettes
        SET recettes= recettes - OLD.montant
        WHERE annee=YEAR(OLD.date_recette)
              AND mois=MONTH(OLD.date_recette)
              AND id_vendeur= OLD.id_vendeur;
END |
DELIMITER ;
```

RECETTES
id_vendeur (FK)
date_recette
montant

VUEMAT_RECETTES
annee
mois
id_vendeur (FK)
recettes

Avantages et inconvénients trigger et vues matérialisées

Les vues matérialisées sont des vraies tables et permettent d'accélérer les recherches :

- possibilité de créer des index ;
- données directement accessibles (pas besoin de les calculer dynamiquement) ;
- contiennent souvent moins de données que les tables sources (pas toujours nécessaire d'indexer).

Les déclencheurs :

- offrent un mécanisme transparent pour les "codeurs" ;
- permettent de maintenir la BD dans un état cohérent.

Les vues matérialisées :

- occupent de la place;
- introduisent de la redondance.

L'utilisation des déclencheurs à un coût :

- Toute opération d'écriture dans une table pourvue de déclencheurs provoquera un overhead lié aux écritures additionnelles et aux éventuels maintiens des index.

→ Adapté quand + de lecture que d'écriture



Bonne perf mais stockage monte mais contraintes

Contraintes

ATTENTION : avec MySQL/MariaDB, les déclencheurs sont liés aux commandes SQL,
pas aux opérations internes du SGBD...

Cascade de triggers

Si MySQL et MariaDB implémentent les déclencheurs de cette façon, c'est pour éviter la problématique de la **CASCADE de trigger** (ce qui permet de ne l'éviter que partiellement...).

Une cascade de trigger c'est quand un trigger déclenche un autre trigger qui en déclenche un autre etc. jusqu'à redéclencher le trigger initial ce qui provoque **une récursion infinie...**

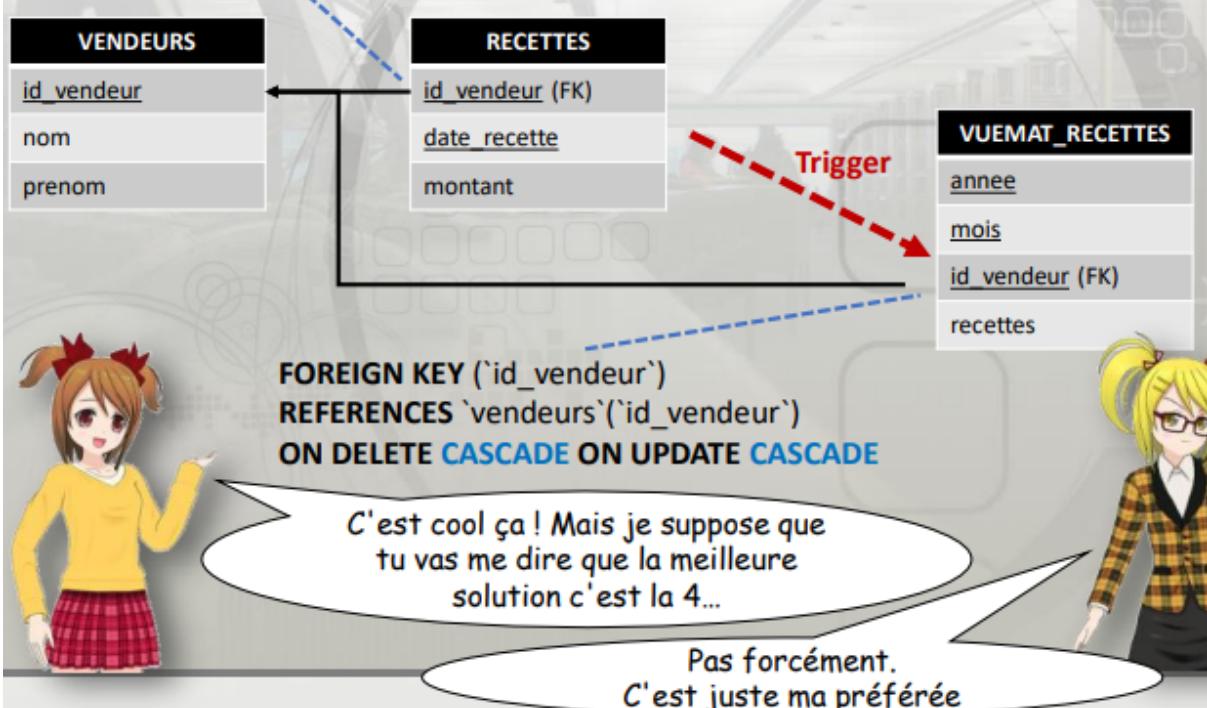
C'est une des raisons qui pousse les hébergeurs à désactiver les triggers et procédures stockées pour les **hébergements mutualisés**.

Pour prévenir ce genre de situation des SGBDR comme PostgreSQL permettent de checker le nombre d'appels avec la fonction **`pg_trigger_depth()`** (à condition de s'en servir).

Solutions pour mysql/mariadb

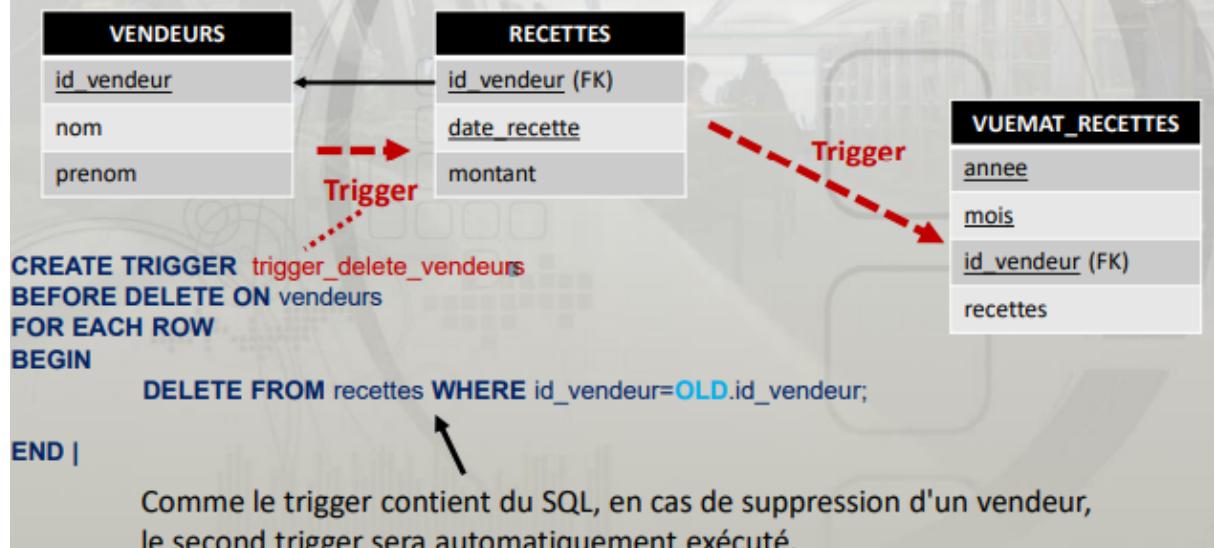
- Solution 1: tu le sais et tu rajoutes des requêtes dans ton code pour ces cas de figure
- Solution 2: tu rajoutes des contraintes de clé étrangère de type CASCADE dans ta vue matérialise

**FOREIGN KEY ('id_vendeur') REFERENCES 'vendeurs'('id_vendeur')
ON DELETE CASCADE ON UPDATE CASCADE**



- Solution 3: Avec MySQL et MariaDB, on se passe des effets de CASCADE pour les clés étrangères et on utilise des triggers un peu partout :

Solution 4 : Avec MySQL et MariaDB, on se passe des effets de CASCADE pour les clés étrangères et on utilise des triggers un peu partout :



Fonctions et procédures stockées

Fonctions

Il est possible de créer ses propres fonctions dans le SGBD :

```
CREATE FUNCTION cm2inch(cm FLOAT)
RETURNS FLOAT
BEGIN
    RETURN (cm / 2.54);
END
```

```
mysql> select cm2inch(25.4);
+-----+
| cm2inch(25.4) |
+-----+
|          10   |
+-----+
1 row in set (0.00 sec)
```

Contrairement à une procédure, une fonction doit fournir une sortie (une valeur ou une relation)

Une fonction en peut pas :

- Gérer des transactions
- Faire des mise à jour (INSERT, UPDATE, DELETE)
- Pas de DDL (CREATE, ALTER, TRUNCATE, etc.)
- Pas d'EXEC/CALL (appel à des procédures stockées)

Procédure stockée

C'est comme une fonction, mais sans les limitations inerrantes.

```
CREATE PROCEDURE nom_de_la_procedure([paramètres])
BEGIN
...
END
```

Avec les procédures stockées, les paramètres peuvent être :

- IN
- OUT
- INOUT

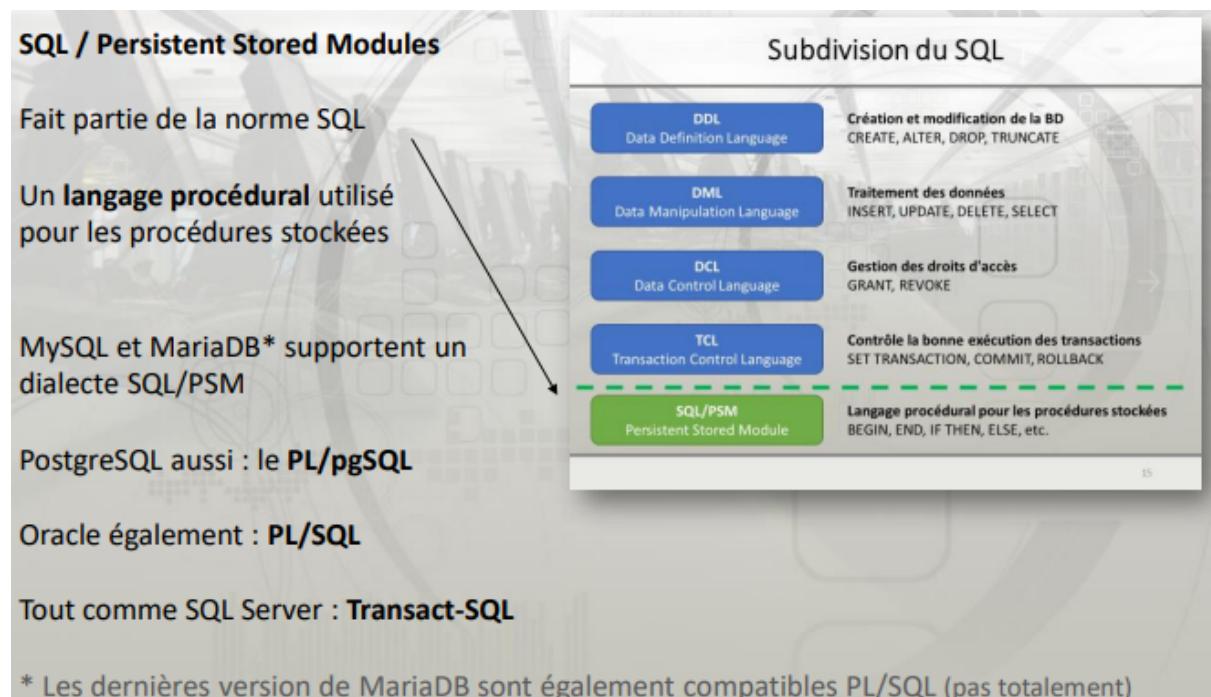
Une procédure est appelée avec la commande : **CALL maprocedure('machin');**

Contrairement aux requêtes préparées, les fonctions et procédures stockées ne sont pas stockées en session utilisateur. Elles sont sauvegardées durablement sur le serveur.

Elles peuvent même être déclenchées par un trigger

En gros, stocké côté serveur, peuvent être déclenchées par trigger

SQL/PSM



Langage procédural pour procédures stockées et norme SQL/PSM

Exemple:

```
DELIMITER |
```

```
CREATE PROCEDURE volume_cube(IN taille INT, OUT volume INT)
BEGIN
    SELECT POW(taille,3) INTO volume;
END|
```

DELIMITER ;

Appel de la procédure :

```
mysql> CALL volume_cube(10, @x);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @x;
+-----+
| @x   |
+-----+
| 1000 |
+-----+
```

Avantages et inconvénients

- Fournissent un mécanisme d'abstraction du schéma pour les utilisateurs :
 - Code de l'application plus simple
 - Pas d'accès direct aux tables (meilleure sécurité)
 - Meilleure réactivité car les procédures stockées sont précompilées
 - Gains substantiels au niveau de la charge réseau :
 - Requêtes plus courtes
 - Moins d'interactions
 - Possible d'appeler des scripts externes
-
- Syntaxe plutôt "has been" et fonctionnalités limitées du langage
 - Elles sont précompilées, ce qui ne protège pas totalement contre les injections SQL si l'implémentation est maladroite.
 - Nécessite un peu de "skill"

Base de données épaisses

La BD est accédée uniquement par des :

- **Vues (matérialisées)** : pour les lectures
- **Procédures Stockées** : pour les écritures

Rend le code de l'application agnostique du schéma de la BD
(bien plus qu'avec un ORM qui lui... ne l'est pas)

Contrainte et assertion

Les contraintes permettent de se prémunir contre l'enregistrement de données incohérentes

Types de contraintes en SQL :

1. Les contraintes de domaine : pour la validation d'une valeur (pas supporter en mysql/mariadb)
 1. Le DBA crée un nouveau type de données décrivant une plage de valeurs possibles
 2. Les colonnes qui sont de ce nouveau type devront obligatoirement stocker des données conformes à la plage de valeurs définie.

```
CREATE DOMAIN type_age int CONSTRAINT contrainte_sur_age CHECK( VALUE>17 AND VALUE<130);
```

```
CREATE TABLE clients (
    id_client INT PRIMARY KEY,
    nom VARCHAR(255) NOT NULL,
    prenom VARCHAR(255) NOT NULL,
    age type_age NOT NULL
);
```

```
CREATE DOMAIN nom_jour VARCHAR(10) CONSTRAINT contrainte_sur_jour
CHECK(VALUE IN ('lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche'));
```

2. Les contraintes de table : pour la validation d'une ligne

Pour valider un enregistrement dans une table.

Les contraintes de table :

- **NOT NULL** : oblige à fournir une valeur non nulle
- **UNIQUE** : il ne peut y avoir plusieurs fois la même valeur (sera indexé) mais peut éventuellement contenir des valeurs nulles
- **PRIMARY KEY** : comme UNIQUE mais il ne peut pas y avoir de valeurs nulles (sera indexé)
- **FOREIGN KEY** : la valeur devra correspondre à une PK d'une autre table et il peut éventuellement contenir des valeurs nulles
- **DEFAULT** : permet de fournir une valeur par défaut (pas réellement une contrainte)
- **CHECK(condition)** : permet définir les conditions à respecter (plage de valeur)

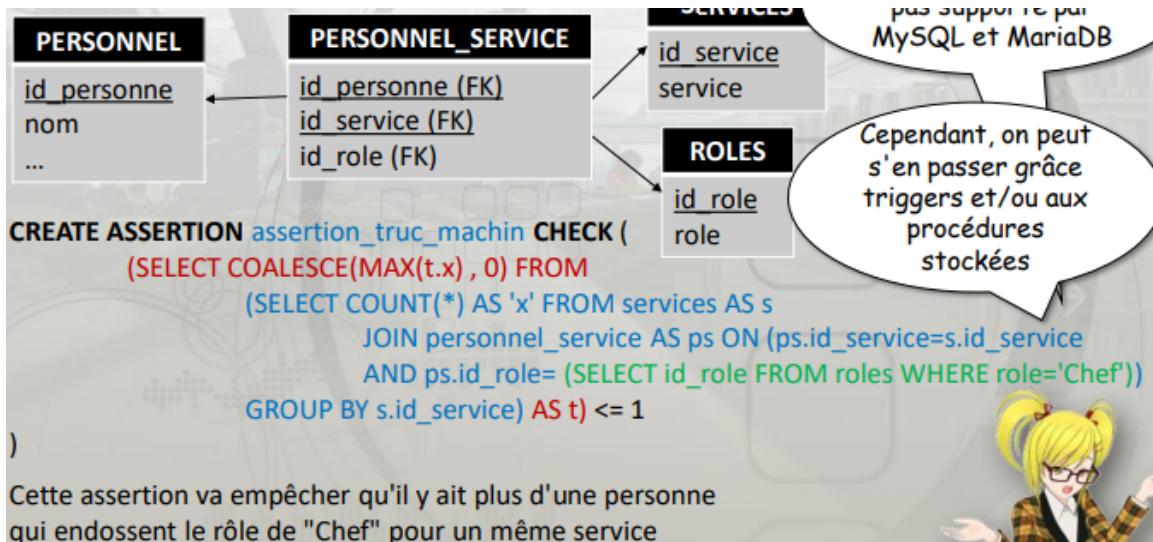
```
CREATE TABLE articles (
    id_article INT NOT NULL,
    reference VARCHAR(255) NOT NULL,
    nom_article VARCHAR(255) NOT NULL,
    stock INT DEFAULT 0,
    prix DECIMAL(10,2) NOT NULL,
    id_categorie INT,
    PRIMARY KEY(id_article),
    UNIQUE(nom_article) ,
    CHECK(prix>0),
    FOREIGN KEY(id_categorie) REFERENCES categories(id_categorie),
    INDEX index_fk_categorie (id_categorie)
);
```



Quand vous définissez une clé étrangère, le SGBD ne va pas forcément automatiquement y associer un index. Pensez donc à bien vérifier vos index.

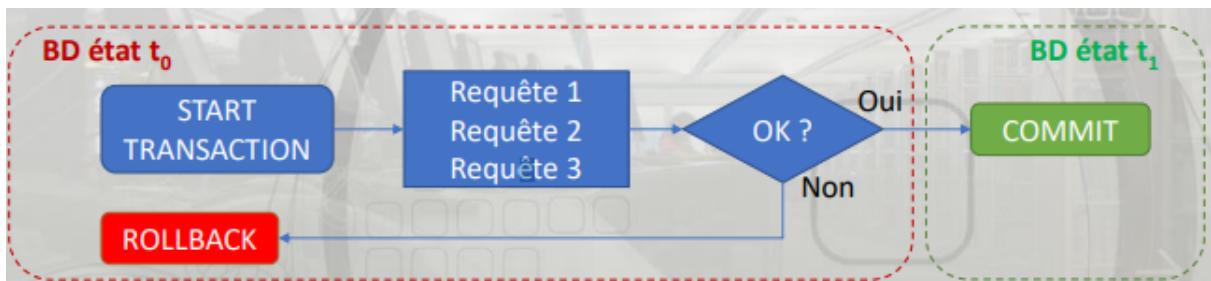
3. Les assertions : pour valider des ensembles de lignes

Permettent de valider des ensembles de lignes (pas supporter mysql/mariadb, remplacer par trigger)



Transactions

Les transactions permettent de **s'assurer** qu'un **ensemble de requêtes** puisse être correctement traité avant de l'**exécuter**.



Mécanisme de protection qui :

- permet de maintenir la DB dans un **état cohérent** ;
- permet de garantir l'**ACIDité** des transactions ;
- est **indispensable** aux utilisations de type **OLTP**

ACIDité d'une transaction

Propriété ACID

Ensemble de propriétés qui garantissent qu'une **transaction** est exécutée de façon **fiable**

- Atomicité : une **transaction** se fait au **complet** ou **pas** du tout

- Cohérence : assure que chaque **transaction** amènera le **système** d'un **état valide** à un autre **état valide**
- Isolation : Toute transaction doit s'exécuter comme si elle était la **seule** sur le **système** (pas de **dépendance**)
- Durabilité : lorsqu'une **transaction** a été **confirmée**, elle demeure **enregistrée**
même à la suite d'une **panne**

Charge OLTP

OLTP (Online Transaction Processing) (traitement transactionnel en ligne) :
Contexte d'utilisation typique des activités opérationnelles (e-commerce, production, système de réservation, etc.)

- Pouvoir **répondre** à un **nombre élevé** de **transactions** (ensemble solidaire de requêtes) par minute.
- Doit être **performant** tant en **lecture** qu'en **écriture**.
- Garantir l'**ACIDité** des transactions

Gestion des accès concurrent :

- Verrouillage pessimiste : La ressource est accessible à une seule transaction et les autres sont en attente
- Verrouillage optimiste : Les accès peuvent se faire concurremment, le SGBD conserve différentes versions d'une ressource associées à éventuellement un Timestamp

MVCC (Multiversion concurrency control) en optimiste : Chaque version d'une ressource dispose d'un tag (numéro de transaction ou timestamp) /!\ peut être pessimiste aussi

Utilisation de LOG

Concomitamment aux données, les SGBD(R) sauvegardent toutes les modifications de la BD (DDL et DML) dans des fichiers logs.

Utilité :

- Revenir en arrière en cas de problème
- Souvent utilisé pour la réPLICATION
- Il peut être avisé d'utiliser des supports de stockage différents pour logs et BD.
- Permet d'écrire simultanément sur les 2 supports (perfs)
- En cas de perte d'un des 2 supports, on conserve une version (soit la BD, soit les logs)
- Permet de gérer plus facilement l'espace libre (les 2 systèmes ne s'impactent pas)

Partitionnement des données

Sharding : Diviser une table (ou BD) pour la répartir (le plus souvent) entre ≠ serveur

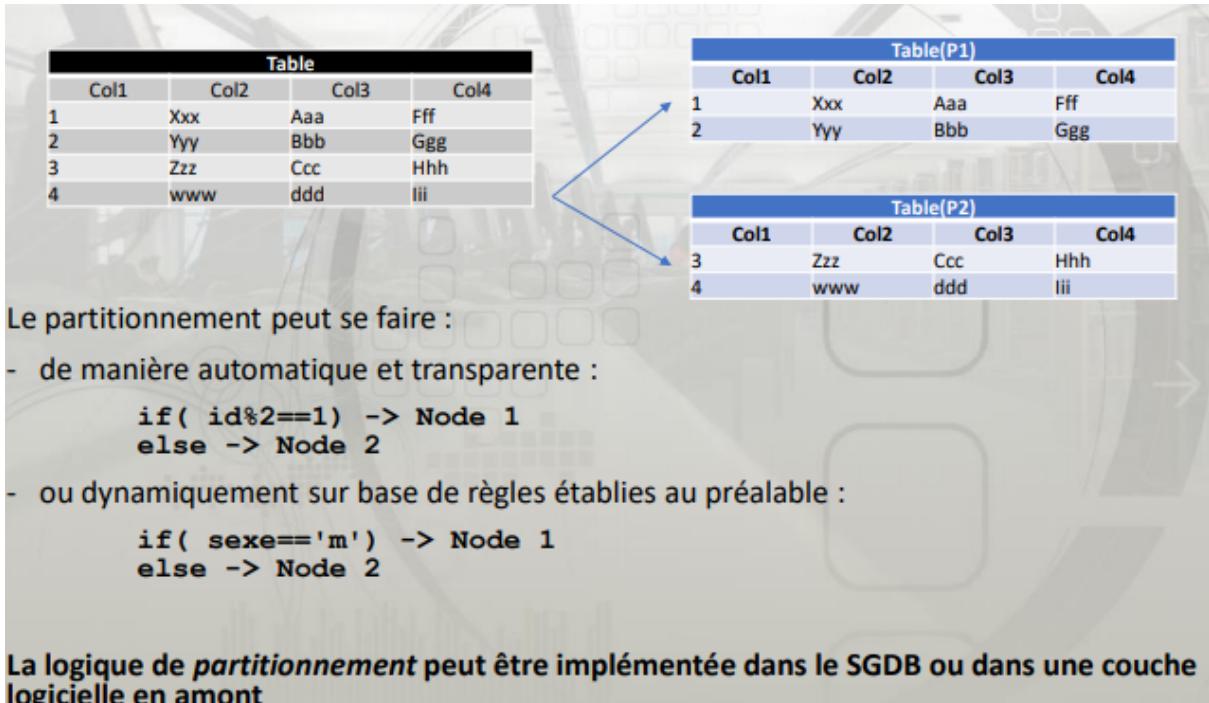
Peut aussi se faire sur une seule et même machine

Une machine n'est plus capable de traiter les informations efficacement ou/et pour fournir plus de scalabilité → **Partitionnement**

Les règles de partitionnement peuvent être définies manuellement ou être automatique

Possible de partitionner la BD en différentes sous-BD isolées les unes des autres ou déployer une seule instance répartie entre différents serveurs

Horizontal



Sharding horizontal

Avantage Horizontal

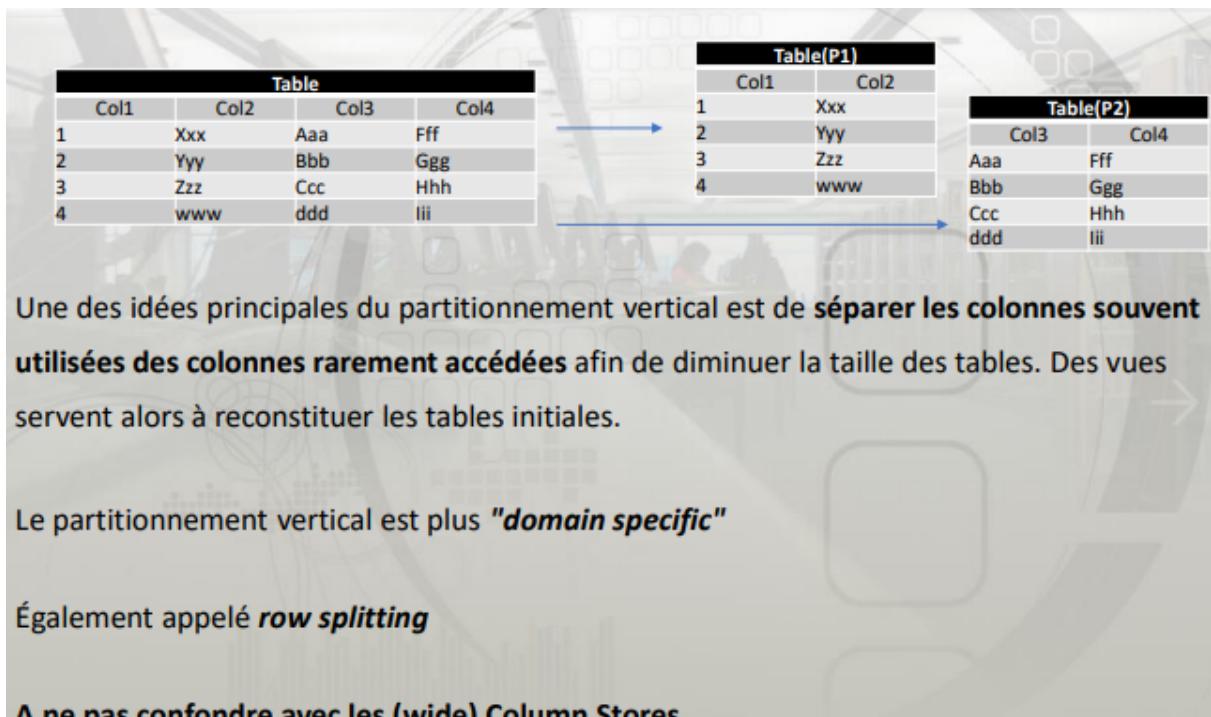
- Le partitionnement horizontal offre la possibilité au système d'interroger les fragments de table simultanément par les différents serveurs :
 - gain en rapidité
 - les index sont également fragmentés (gain en rapidité d'écriture)
- Load balancing pour les écritures

Désavantage Horizontal

- Infrastructure plus complexe
- Augmentation de la latence

- Les recherches croisées (jointures) peuvent saturer le canal de communication utilisé entre les serveurs (bottleneck).

Partitionnement Vertical



RéPLICATION et proxy

Répliquer les données sur plusieurs serveurs

2 serveurs minimum: relation master-slave

- Utilité première : avoir une copie de la BD, si crash, slave peut prendre le relais
- Autre utilité : soutenir de fortes charges, requête d'interrogation aussi sur le slave

Different types of replication: asynchrone, semi-synchrone, synchrone

Asynchrone

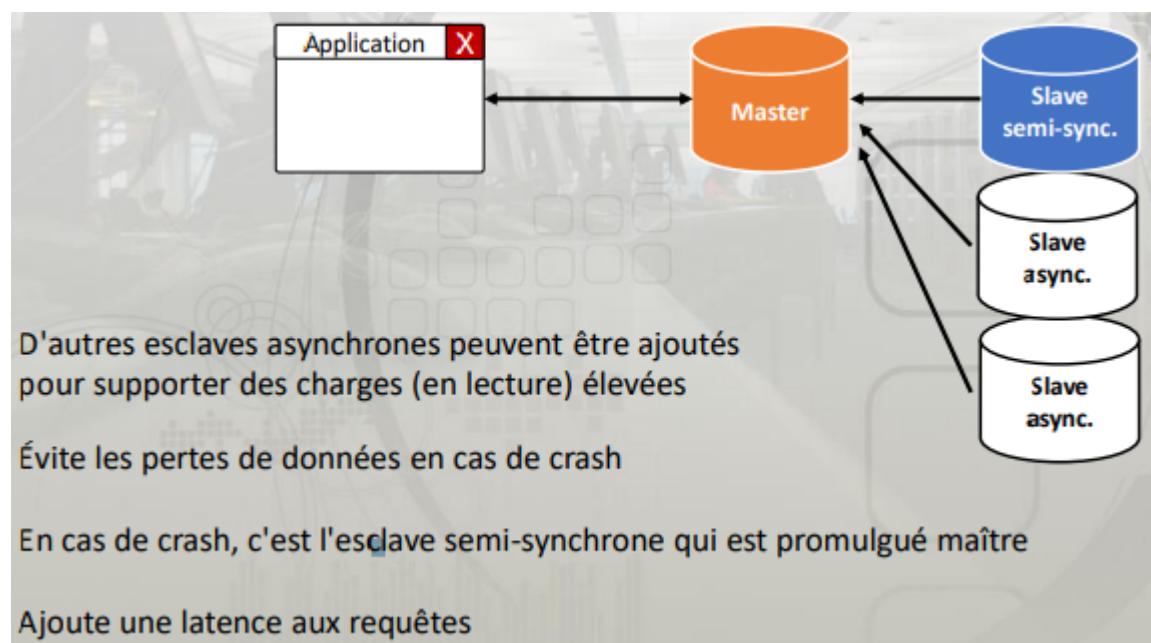
Le master n'attend pas le slave

Evènement dans journal de log qui est consulté tout les X temps (lag) par le slave, peut avoir perte d'info si maître crash, dû au lag, les réponses du Slave peuvent être temporellement fausses à cause du lag aussi

Slave en lecture seul, et peut être sollicité pour les lectures

Semi-synchrone

Le maître ne valide pas la transaction tant qu'au moins un esclave ne l'a pas



Synchrone



RéPLICATION logique (transactionnelle)

Avec la réPLICATION logique (PostgreSQL) ou transactionnelle (SQL Server) ce n'est pas la BD qui est répliquée mais la (ou les) table(s).

Proxy

Un proxy est un logiciel qui gère la connectivité entre 2 partis

Un proxy est un logiciel qui gère la connectivité entre 2 partis L'application cliente est connectée au proxy en pensant que c'est le SGBDR

Permet:

- Routage du traffic SQL (pour les infrastructures multi-serveurs) et load balancing
- Firewall SQL
- Réécriture des requêtes
- Switching Master/Slave automatique
- Meilleure sécurité car il n'y a que le proxy qui peut interroger les SGBDR

Définition

Technologie sémantique :

- Permet d'effectuer des recherches sur base du sens des valeurs et non sur la valeur des valeurs ...
- Ajoute des nouveaux index et fonctionne en complément de l'indexation FULL-Text

Fuzzy Search (recherche approximative) :

- Trouver des chaînes de caractères qui correspondent à un motif approximatif
- Exemple : entrer "algorytme" et le système trouve "algorithme"

Multi-tenant / multi-base :

- Permet d'exécuter des requêtes qui impliquent plusieurs bases de données
- Natif dans SQL Server et avec des performances optimisées
- Non natif mais possible avec Oracle(option \$\$\$) et PostgreSQL
- Natif avec MySQL/MariaDB

Les SGBD découpent les données en fragments : pages Ce sont les pages qui sont la plus petite unité de la BD qui est lue/écrite sur le disque

Big Data, NoSQL et NewSQL

Règle des 3V

- **Volume**

Le critère principal = quantité de données à traiter. Actuellement, de nombreux "experts" estiment que les volumes "type" Big Data sont $\geq 1 \text{ To}$

- **Vélocité**

Ici 2 axes :

- le **rythme élevé** auquel les données sont générées
- le **besoin d'analyser** mes données en "temps réel"

- **Variété**

Les sources sont **diverses** et **variées** et les données brutes sont souvent déstructurées

Définitions

Le Big Data, c'est quand les SGBDR ne sont plus capable d'effectuer les traitements

Data Mining (exploration des données) : extraction de connaissance à partir de grande quantité de **données structurées**

Big Data :

- Tenter de résoudre les **problèmes de stockage des données** tout est fournissant un **accès rapide**
- Offre la **possibilité** aux "développeurs" de tenter de **structurer les données extraites** pour éventuellement faire du **Data Mining**

Scalabilité:

Pouvoir adapter un système à la montée en charge en lui ajoutant des ressources

Scale UP: Scalabilité verticale: SGBDR

Scale OUT: scalabilité horizontale: NOSQL, Big DATA

NoSQL

Not Only SQL

Bases de données géantes

Inverse de SGBDR

- Pas de schéma et données non structurées
- Pas d'ACIDité complète

Bonnes perfs avec scalabilité

Diff paradigmes dont key-values stores

Search Engine

lassé comme SGBD NoSQL

Sont souvent utilisés en renfort pour se charger des requêtes de recherche complexe afin de décharger les serveurs plus critiques

key-values stores

Peut être vu comme un **simple tableau associatif clé-valeur** (voir plusieurs tableaux)

Clé unique, clé indexée grâce à table de hachage

Utiliser dans NoSQL, systèmes embarqués, serveur de cache

Document Store

Évolution des bases de données XML, mais utilise souvent json car xml pue et json moins lourd à charge égale

Avantages:

- Scalable
- POO adapté
- Charger un seul fichier au lieu d'exécuter des jointures

Inconvénients:

- "Putain" de redondance de données
- Performances peuvent être catastrophiques, si il faut croiser une multitude de document

Wide column store

Aussi appelé Extensible Record Store

Système Google Big Table à l'origine

Cassandra le + connu

Graph DBMS

Base de données orientée graphe

Base de données orientée objet dans laquelle les objets sont interconnectés entre eux pour former un graphe

Hadoop

Framework constitué de:

- Stockage HDFS
- Partie traitement implémentale

Souvent utilisé conjointement à d'autres systèmes en tant que solution de stockage et d'accès aux données :

- Abstraction du stockage physique
- Scalabilité horizontale : il suffit d'ajouter des nœuds pour accroître l'espace de stockage

NewSQL



- Modèle relationnel
- Scalabilité horizontale
- Architecture distribuée de type *shared-nothing*
- Utilisation OLTP
- ACIDité des transactions
- Un minimum de verrous (voir aucun)
- Moteur de stockage propre (pas de HDFS)

Un système NewSQL doit intégrer son propre moteur de stockage

- Permet certaines optimisations (car non généraliste)
- Offre des stratégies de partitionnement, de réPLICATION et de rééquilibrage de charge plus sophistiquées.

SQL - PN maison

Historique et description SQL

Prononcé SEQUEL

Langage de prog. déclaratif (Depuis 1999), inspiré d'algèbre relationnelle, qui est inspiré de la théorie des ensembles

SQL = langage procédural pour les procédures stockées

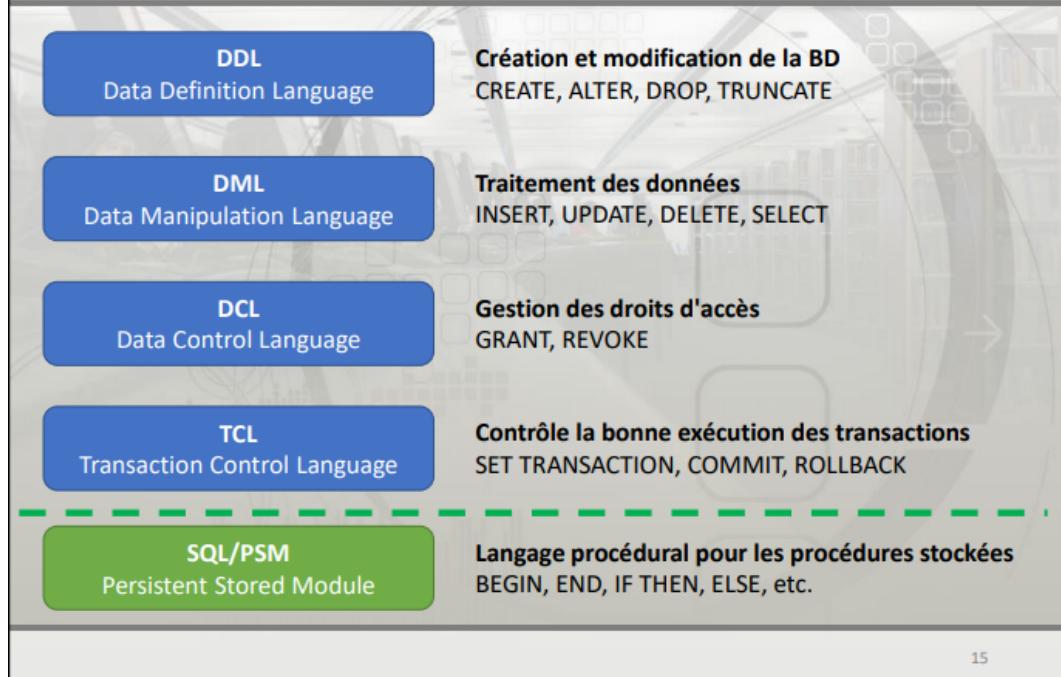
4 manière de connecter SQL à DB:

1. Connecteur de DB avec écriture des requêtes sous forme de chaînes de caractères. Soit driver dédié à SGBD particulier, soit middleware qui se connecte à driver dédié
2. Embedded SQL: Intégration du langage SQL directement dans le code source de l'application. SQL est analysé par le compilateur/interpréteur de l'application
3. ORM (Object Relational Mapping): Permet de générer automatiquement le SQL via une couche d'abstraction Objet, requêtes générées automatiquement, pas optimiser et plus lent et rajoute une couche logicielle supplémentaire et gère pas le multithreading (ou mal).
4. Principe des bases épaisse: SQL pas dans l'application mais dans le SGBDR (Utilisation de procédures stockées)
 - Vues : une table virtuelle définie par une requête SQL stockée côté serveur
 - Procédures Stockées : fonctions précompilées et stockées côté serveur. Elles peuvent contenir du SQL ainsi que du code impératif (procédural)

Rend le code de l'application agnostique du schéma de la DB mais + chaud

Fonctionnement SQL

Subdivision du SQL



15

DDL

Ce sont les instructions SQL qui permettent :

- La création de la BD: `CREATE DATABASE ma_bd;`
- La création des tables : `CREATE TABLE ma_table (...);`
- Modification des tables : `ALTER ma_table ...`
- Effacer une base de données : `DROP DATABASE ma_vieille_bd;`
- Effacer une table : `DROP TABLE ma_vieille_table;`
- Effacer le contenu d'une table : `TRUNCATE TABLE ma_table_pleine;`

Il n'y a pas que les tables mais également les **index** (voir chapitre sur l'indexation), les **assertions, vues, fonctions, trigger et procédures** (voir chapitre sur les fonctionnalités avancées).

```
CREATE TABLE ma_table (
    id_table  INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    nom VARCHAR(100) NOT NULL,
    prenom  VARCHAR(100) NOT NULL,
    date_de_naissance DATE
);
```

Faire un ALTER TABLE si on a oublié un truc

Types de données textuelles

CHAR(taille), fixe et max 255

VARCHAR(taille), variable, max 65535

→ TINYTEXT et TEXT = pareil

Interclassement (Collation)

Table de correspondance spécifique à un jeu de caractères

Permet l'ordonnancement correct d'une liste de caractères

- Le "é" vient après le "e", ou que les majuscules viennent avant les minuscules par ex

Permet de savoir quand un caractère est "équivalent" à un autre

- Le caractère "e" est équivalent à "E", mais aussi à "é", "è" ou "ê"

→ MySQL et MariaDB: **utf8mb4_unicode_520_ci**

Types numériques

INT: entier de 32bits

FLOAT: réel simple précision 32bits (DOUBLE:64)

NUMERIC(taille) Entier de taille chiffres

DECIMAL(taille,n) Réel de taille chiffres et de n chiffres après la virgule (inclus dans taille)

Types temporelles

DATE: Date au format 'aaaa-mm-jj'

DATETIME: Date avec heure au format 'aaaa-mm-jj hh:mm:ss'

TIMESTAMP: Pareil que DATETIME mais temps écoulé depuis 1er janvier 1970 au niveau stockage

TIME: Durée au format 'HH:MM:SS' ou 'HH:MM:SS.uduuuu', peut être négatif

YEAR: Année au format 'aaaa' , bornée entre 1901 et 2155

DML



```
INSERT INTO ma_table VALUES ( 10 , 'Marc' , 'Dufour' , NULL , '1988-10-19' );
INSERT INTO ma_table (id, nom, prenom, email, date_naissance)
VALUES ( 10 , 'Marc' , 'Dufour' , NULL , '1988-10-19' );
```

Si on rajoute des attributs, la première n'est plus fonctionnelle, à part si les attributs null sont autorisés alors que la deuxième fonctionne tjrs car on précise les colonnes

On peut faire plusieurs tuples d'un coup, c'est plus rapide que plusieurs requêtes mono-tuple

Insertion et clé dupliquée



num_chassis	voiture	couleur
abcdabcde13	Polo	grise
yzzzABC4442	Golf	rouge
BKxyOMzz13	Clio	bleue
xxxYbbzz14	Clio	jaune
zaeaFJW123	Punto	rouge

La requête suivante va provoquer une erreur (PK dupliquée) :

```
INSERT INTO ma_table VALUES ( 'xxxYbbzz14','Clio','noire');
```

Il est possible d'ignorer l'erreur et rien ne sera ajouté dans la table

```
INSERT IGNORE INTO ma_table VALUES ( 'xxxYbbzz14','Clio','noire');
```

Il est possible de transformer l'insertion en une modification en cas de PK dupliquée :

```
INSERT INTO ma_table VALUES ( 'xxxYbbzz14','Clio','noire')
```

```
ON DUPLICATE KEY UPDATE voiture='Clio', couleur='noire';
```

Supprimer données

```
DELETE FROM ma_table WHERE id < 100;
```

Pas oublier le WHERE sinon on supp tout

DELETE et TRUNCATE

DELETE = DML

TRUNCATE = DDL

Contrairement à DELETE, TRUNCATE

- Ne déclenche pas les triggers (déclencheurs)
- Ne peut pas être exécuté sur les tables munies de contraintes d'intégrité référentielle (gestion des clés étrangères liées à la table)
- Réinitialise les compteurs d'auto-incrémentations

Mises à jour

```
UPDATE ma_table
    SET nom='Roger',
        prenom='Tartuffo'
WHERE id=17;
/* Pas oublier le where sinon tout est modifié */
--commentaire
#commentaire

# Opération pour modifier à partir de l'ancienne valeur
UPDATE articles SET prix = prix*0.8 WHERE id_article=42;
```

Rechercher des données (Toujours DML)

SELECT

```
# Cette requête récupère tout le contenu de ma_table  
SELECT * FROM ma_table;
```

SELECT permet une augmentation des perfs et d'économiser beaucoup de ligne de code

```
SELECT liste de colonnes ou * pour toutes les colonnes  
FROM ma_table  
WHERE prédicts  
GROUP BY ...  
HAVING ...  
ORDER BY liste de colonnes;
```

- WHERE = filtre de recherche
- GROUP BY = regroupement des résultats
- HAVING = filtre portant sur les regroupements des résultats
- ORDER BY = tri

Opérateur DISTINCT

```
/*  
Permet de sélectionner les valeurs différentes, on n'aura pas  
Permet aussi un gain de perf  
*/  
SELECT DISTINCT type_plat FROM plats;
```

Concaténation CONCAT

Table *plats*

id	type_plat	nom	prix
1	Sandwich	Poulet-curry	2.50
2	Burger	Royal-Cheese	5.10
3	Sandwich	Martino	3.00
4	Burger	King-Bacon	5.40
5	Pizza	4 fromages	8.50

```
SELECT id, CONCAT(type_plat, ' : ', nom)
FROM plats;
```

id	CONCAT(type_plat, ' : ', nom)
1	Sandwich : Poulet-curry
2	Burger : Royal-Cheese
3	Sandwich : Martino
4	Burger : King-Bacon
5	Pizza : 4 fromages

SELECT

```
id, CONCAT(type_plat, ' : ', nom) AS 'plat'
FROM plats;
```

id	plat
1	Sandwich : Poulet-curry
2	Burger : Royal-Cheese
3	Sandwich : Martino
4	Burger : King-Bacon
5	Pizza : 4 fromages

Table *plats*

id	type_plat	nom	prix
1	Sandwich	Poulet-curry	2.50
2	Burger	Royal-Cheese	5.10
3	Sandwich	Martino	3.00
4	Burger	King-Bacon	5.40
5	Pizza	4 fromages	8.50

SELECT

```
id, CONCAT_WS(' : ', type_plat, nom) AS plat
FROM plats;
```

id	plat
1	Sandwich : Poulet-curry
2	Burger : Royal-Cheese
3	Sandwich : Martino
4	Burger : King-Bacon
5	Pizza : 4 fromages

WS = With Separator

CONCAT_WS(separator,value1, value2, ... , valueN)

ORDER BY (Tri)

Table *plats*

id	type_plat	nom	prix
1	Sandwich	Poulet-curry	2.50
2	Burger	Royal-Cheese	5.10
3	Sandwich	Martino	3.00
4	Burger	King-Bacon	5.40
5	Pizza	4 fromages	8.50

SELECT * FROM plats ORDER BY type_plat, nom

id	type_plat	nom	prix
4	Burger	King-Bacon	5.40
2	Burger	Royal-Cheese	5.10
5	Pizza	4 fromages	8.50
3	Sandwich	Martino	3.00
1	Sandwich	Poulet-curry	2.50

ATTENTION
l'utilisation de l'opérateur
ORDER BY peut avoir un
impact non négligeable sur
les performances !



Il est conseillé de trier les données uniquement quand c'est utile et de façon réfléchie (pas sur toutes les colonnes si ce n'est pas nécessaire)

Trier par type_plat, puis par nom si des type_plat sont identiques

```
SELECT * FROM plats ORDER BY type_plat, nom
```

Ascendant et Descendant (Par défaut ASC)

```
SELECT * FROM plats ORDER BY ASC type_plat, DESC nom
```

Tri forcé avec FIELD

```
SELECT * FROM plats ORDER BY FIELD(type_plat, 'Sandwich', 'Burger')
```

WHERE

```
SELECT * FROM ma_table WHERE prédicts
```

Prédicats = expressions logiques renvoyant des valeurs booléennes

Expressions logiques: Opérateur:

- Comparaison (= < > <= >= <>), peuvent être combinées à OR, NOT, AND
 - IS: *SELECT * FROM plats WHERE prix IS NULL*

- `SELECT id FROM plats WHERE stock; # Peut ne pas fonctionner`
- `SELECT id FROM plats WHERE stock=1; # Fonctionne`
- LIKE: `SELECT * FROM plats WHERE nom LIKE 'King%' OR nom LIKE '_fromages';`
 - % remplace un nombre inconnu de caractères
 - _ remplace un seul caractère inconnu
- BETWEEN: `SELECT * FROM plats WHERE (prix BETWEEN 3 AND 6);`
 - Requête équivalente: `SELECT * FROM plats WHERE prix>=3 AND prix<=6`
- IN: `SELECT * FROM plats WHERE id IN (1,4,2)`
 - Requête équivalente: `SELECT * FROM plats WHERE id=1 OR id=4 OR id=2;`
 - Opposée: `SELECT * FROM plats WHERE id NOT IN (1,4,2) ou SELECT * FROM plats WHERE id<>1 AND id<>4 AND id<>2;`
- Requête imbriquée avec IN

Le contenu des parenthèses du IN peut être remplacé par une requête SELECT ne retournant qu'une seule colonne.

Table *types_plat*

id_type	nom
1	Sandwich
2	Burger
3	Pizza

Table *plats*

id	id_type	nom	prix
1	1	Poulet-curry	2.50
2	2	Royal-Cheese	5.10
3	1	Martino	3.00
4	2	King-Bacon	5.40
5	3	4 fromages	8.50

```
SELECT DISTINCT nom FROM types_plat
WHERE id_type IN (
    SELECT id_type FROM plats
    WHERE (prix BETWEEN 3 AND 6)
);
```

nom
Burger
Sandwich

- NOT, AND, OR
- Mathématiques (+,-,*,/, %)

Gestion de NULL

La fonction COALESCE renvoi la première valeur non **NULL** :

id	type_plat	nom	prix
1	Sandwich	Poulet-curry	2.50
2	Burger	Royal-Cheese	5.10
3	Sandwich	Martino	<i>NULL</i>
4	Burger	King-Bacon	5.40
5	Pizza	4 fromages	8.50

id	nom	prix
1	Poulet-curry	2.50
2	Royal-Cheese	5.10
3	Martino	0
4	King-Bacon	5.40
5	4 fromages	8.50

SELECT id, nom, COALESCE(prix,0) AS 'prix' FROM plats

1. **Gestion des valeurs NULL** : Lorsqu'une colonne peut contenir des valeurs **NULL**, **COALESCE** permet de fournir une valeur par défaut.

Exemple :

```
sql Copier le code
SELECT COALESCE(prenom, 'Inconnu') AS prenom_affiche FROM utilisateur
```

• Si **prenom** contient une valeur, elle est affichée.
 • Sinon, "Inconnu" est retourné.

Structure de contrôle

```

SELECT
    nom,
    IF(prix<3.5, 'pas cher', 'trop cher') AS 'tarif',
    CASE WHEN stock=0 THEN 'rupture'
        WHEN stock<10 THEN 'faible'
        ELSE 'ok'
    END AS stock
FROM plats;

```

id	type_plat	nom	prix	stock
1	Sandwich	Poulet-curry	2.50	35
2	Burger	Royal-Cheese	5.10	4
3	Sandwich	Martino	3.00	87
4	Burger	King-Bacon	5.40	0
5	Pizza	4 fromages	8.50	3

nom	tarif	stock
Poulet-curry	pas cher	ok
Royal-Cheese	trop cher	faible
Martino	pas cher	ok
King-Bacon	trop cher	rupture
4 fromages	trop cher	faible

Traitement donnée temporelle

```

SELECT YEAR('2019-02-01 10:32:54');
SELECT MONTH('2019-02-01 10:32:54');
SELECT DAY('2019-02-01 10:32:54');
SELECT HOUR('2019-02-01 10:32:54');
SELECT MINUTE('2019-02-01 10:32:54');
SELECT SECOND('2019-02-01 10:32:54');
SELECT MICROSECOND('2019-02-01 10:32:54.456456');

```

-> 2019
-> 2
-> 1
-> 10
-> 32
-> 54
-> 456456

Connaître la date et/ou l'heure actuelle : CURRENT_TIMESTAMP, NOW(), CURRENT_DATE, CURRENT_TIME

```
SELECT DATE_FORMAT('2018-11-03', '%a %d/%m/%Y');
```

-> Sat 03/11/2018

Il existe d'autres fonctions dédiées aux traitements des données temporelles :
DATE_ADD, DATEDIFF, DAYOFWEEK, DAYOFYEAR, TIMEDIFF, UTC_TIME, WEEKOFYEAR,
SEC_TO_TIME, MONTHNAME, etc.

Fonctions

Les classiques :

SELECT TRIM(' boulette ');	-> boulette
SELECT POSITION('ca', 'Abracadabra');	-> 5 (0 s'il ne trouve pas)
SELECT SUBSTRING('Abracadabra',5,2);	->ca
SELECT UPPER('Sarah');	->SARAH
SELECT LOWER('Sarah');	-> sarah
SELECT REPLACE('Abracadabra', 'ra', 'R4');	-> AbR4cadabR4
SELECT ROUND(1.58);	-> 2
SELECT FLOOR(1.58);	-> 1
SELECT TRUNCATE(1.58,1);	-> 1.5

GROUP BY

SELECT

type_plat,
COUNT(id),
MIN(prix),
MAX(prix),
AVG(prix),
SUM(stock),
SUM(stock*prix) AS valeur
FROM plats
GROUP BY type_plat;

id	type_plat	nom	prix	stock
1	Sandwich	Poulet-curry	2.50	35
2	Burger	Royal-Cheese	5.10	4
3	Sandwich	Martino	3.00	87
4	Burger	King-Bacon	5.40	0
5	Pizza	4 fromages	8.50	3

type_plat	COUNT(id)	MIN(prix)	MAX(prix)	AVG(prix)	SUM(stock)	valeur
Burger	2	5.10	5.40	5.250000	4	20.40
Pizza	1	8.50	8.50	8.500000	3	25.50
Sandwich	2	2.50	3.00	2.750000	122	348.50

Comme le ORDER BY, le GROUP BY peut se faire sur plusieurs colonnes

Avec HAVING on peut filtrer les groupes

HAVING SUM(stock) < 10 # à la fin de la requête

Jointure (Tjrs DML)

Cartésien VS JOIN

On peut se passer de JOIN avec mais ça donne moins de visibilité au code

```
SELECT types_plat.nom_type, plats.nom, plats.prix  
FROM (plats, types_plat)  
WHERE types_plat.id_type=plats.id_type AND plat.id=5;
```

Il est grandement conseillé de réécrire la requête ci-dessus en utilisant la clause JOIN

```
SELECT types_plat.nom_type, plats.nom, plats.prix  
FROM (plats)  
JOIN (types_plat) ON (types_plat.id_type=plats.id_type )  
WHERE plat.id=5;
```

Par rapport au produit cartésien, la clause JOIN :
- offre une meilleure lisibilité du code
- permet différents types de jointure
- obtient souvent de meilleures performances

ON = condition de jointure

Surnommage

```
SELECT types_plat.nom_type, plats.nom, plats.prix FROM (plats)  
JOIN (types_plat) ON (types_plat.id_type=plats.id_type )  
WHERE plat.id=5;
```

 "surnommage" des tables

```
SELECT t.nom_type, p.nom, p.prix FROM (plats AS p)  
JOIN (types_plat AS t) ON (t.id_type=p.id_type )  
WHERE p.id=5;
```

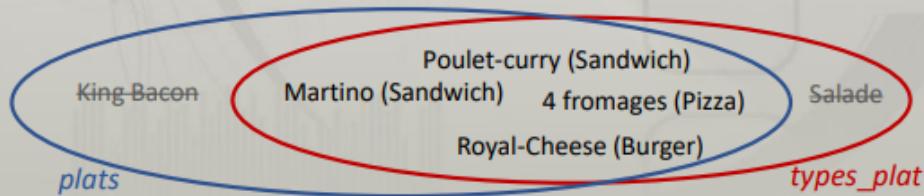
INNER JOIN

JOIN basique si rien est précisé, donc INNER est facultatif

```
SELECT t.nom_type, p.nom, p.prix FROM (plats AS p)  
INNER JOIN (types_plat AS t) ON (t.id_type=p.id_type );
```

id	id_type	nom	prix	id_type	nom_type	nom_type	nom	prix
1	1	Poulet-curry	2.50	1	Sandwich	Sandwich	Poulet-curry	2.50
2	2	Royal-Cheese	5.10	2	Burger	Burger	Royal-Cheese	5.10
3	1	Martino	3.00	3	Pizza	Sandwich	Martino	3.00
4	NULL	King-Bacon	5.40	4	Salade	Pizza	4 fromages	8.50
5	3	4 fromages	8.50	Table <i>types_plat</i>				

Table *plats*



Sélectionne que les éléments commun des 2, ici id_type= 4 est NULL donc on exclut la ligne qui a cette valeur

NATURAL JOIN

```
SELECT t.nom_type, p.nom, p.prix FROM (plats AS p)
NATURAL JOIN (types_plat AS t)
```

Le SGBDR va effectuer la jointure en se basant sur les colonnes portant les même noms et types

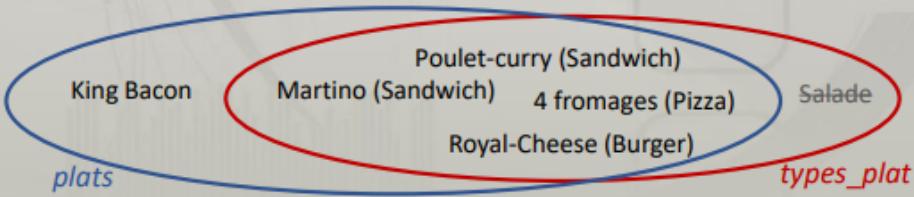
→ Pas ouf car si mêmes plusieurs colonnes ont le même nom en commun, rien ne fonctionne

LEFT (OUTER) JOIN (Externe gauche)

```
SELECT t.nom_type, p.nom, p.prix FROM (plats AS p)
LEFT OUTER JOIN (types_plat AS t) ON (t.id_type=p.id_type );
# Pas obligé de mettre OUTER
```

id	id_type	nom	prix	id_type	nom_type	nom_type	nom	prix
1	1	Poulet-curry	2.50	1	Sandwich	Sandwich	Poulet-curry	2.50
2	2	Royal-Cheese	5.10	2	Burger	Burger	Royal-Cheese	5.10
3	1	Martino	3.00	3	Pizza	Sandwich	Martino	3.00
4	NULL	King-Bacon	5.40	4	Salade	Pizza	4 fromages	8.50
5	3	4 fromages	8.50	Table <i>types_plat</i>			NULL	King-Bacon

Table *plats*



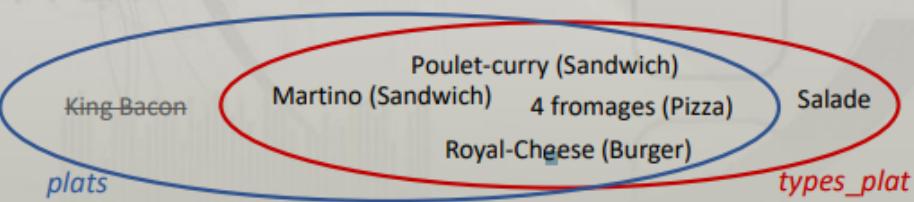
Ici, la partie gauche (table de base) est sélectionnée mais pas la partie droite (table jointe) ne l'est pas

RIGHT OUTER JOIN (Externe droite)

```
SELECT t.nom_type, p.nom, p.prix FROM (plats AS p)
RIGHT OUTER JOIN (types_plat AS t) ON (t.id_type=p.id_type );
# OUTER facultatif
```

id	id_type	nom	prix	id_type	nom_type	nom_type	nom	prix
1	1	Poulet-curry	2.50	1	Sandwich	Sandwich	Poulet-curry	2.50
2	2	Royal-Cheese	5.10	2	Burger	Burger	Royal-Cheese	5.10
3	1	Martino	3.00	3	Pizza	Sandwich	Martino	3.00
4	NULL	King-Bacon	5.40	4	Salade	Pizza	4 fromages	8.50
5	3	4 fromages	8.50	Table <i>types_plat</i>			Salade	NULL

Table *plats*



Ici, la partie droite(table jointe) est sélectionnée mais pas la partie gauche(table de base) ne l'est pas

En gros, inverse de LEFT JOIN

FULL OUTER JOIN (Externe bilatérale)

id	id_type	nom	prix	id_type	nom_type	nom_type	nom	prix
1	1	Poulet-curry	2.50	1	Sandwich	Sandwich	Poulet-curry	2.50
2	2	Royal-Cheese	5.10	2	Burger	Burger	Royal-Cheese	5.10
3	1	Martino	3.00	3	Pizza	Sandwich	Martino	3.00
4	NULL	King-Bacon	5.40	4	Salade	Pizza	4 fromages	8.50
5	3	4 fromages	8.50	Table <i>types_plat</i>			NULL	King-Bacon
						Salade	NULL	NULL

Table *plats*

The diagram illustrates the result of a FULL OUTER JOIN between the 'plats' and 'types_plat' tables. It shows the following combinations:

- King Bacon (from 'plats') is paired with Poulet-curry (Sandwich) and Martino (Sandwich).
- Royal-Cheese (Burger) is paired with Burger.
- 4 fromages (Pizza) is paired with Pizza.
- Salade (from 'plats') is paired with Salade.
- There are also rows from 'types_plat' that have no matches in 'plats': Sandwich, Burger, and Pizza.

En gros, tout est sélectionné, mix de gauche et droite

Pas dispo en MySQL/MariaDB

Opérateur ensembliste (Tjrs DML)

UNION

id	nom
1	Sandwich
2	Burger
3	Pizza
4	Salade
5	Lasagne

Table *plats*

id	menu	prix
1	Sandwich	4.00
2	Burger	7.30
3	Pizza	11.50
4	Mitaillette	5.50
5	Salade	6.00

Table *menus*

nom
Sandwich
Burger
Pizza
Salade
Lasagne
Mitaillette

SELECT nom FROM plats UNION SELECT menu FROM menus

Salade apparaît qu'une fois

Combine les résultats de deux requêtes et supprime les doublons (valeurs répétées).

- Les lignes résultantes sont uniques.
- Les colonnes des deux requêtes doivent avoir le même nombre et les mêmes types de données.

UNION ALL

Table *plats*

id	nom
1	Sandwich
2	Burger
3	Pizza
4	Salade
5	Lasagne

Table *menus*

id	menu	prix
1	Sandwich	4.00
2	Burger	7.30
3	Pizza	11.50
4	Mitaillette	5.50
5	Salade	6.00

SELECT nom FROM plats UNION ALL SELECT menu FROM menus

nom
Sandwich
Burger
Pizza
Salade
Lasagne
Sandwich
Burger
Pizza
Mitaillette
Salade

A girl with yellow hair says: *Avec d'autres SGBDR que MySQL/MariaDB, le second SELECT peut être omis*

Semblable à UNION, mais il ne supprime pas les doublons.

Plus rapide que UNION car il ne nécessite pas de comparaison pour éliminer les doublons.

EXCEPT

L'opérateur EXCEPT permet de récupérer la différence

id	nom
1	Sandwich
2	Burger
3	Pizza
4	Salade
5	Lasagne

Table *plats*

id	menu	prix
1	Sandwich	4.00
2	Burger	7.30
3	Pizza	11.50
4	Mitaillette	5.50
5	Salade	6.00

Table *menus*

nom
Lasagne

SELECT nom FROM plats EXCEPT menu FROM menus

Avec ORACLE
c'est MINUS

MySQL ne supporte pas cette fonction
et MariaDB uniquement à partir de la version 10.3

En l'absence d'EXCEPT, quelle serait
une requête équivalente ?

**SELECT nom FROM plats WHERE nom NOT IN
(SELECT menu FROM menus)**



Retourne les lignes présentes dans le premier ensemble mais qui ne le sont pas dans le second.

INTERSECT

id	nom
1	Sandwich
2	Burger
3	Pizza
4	Salade
5	Lasagne

Table *plats*

id	menu	prix
1	Sandwich	4.00
2	Burger	7.30
3	Pizza	11.50
4	Mitaillette	5.50
5	Salade	6.00

Table *menus*

nom
Sandwich
Burger
Pizza
Salade

SELECT nom FROM plats INTERSECT menu FROM menus

A nouveau, MySQL ne supporte pas cette fonction
et MariaDB uniquement à partir de la version 10.3

En l'absence d'INTERSECT, quelle
serait une requête équivalente ?

**SELECT nom FROM plats WHERE nom IN
(SELECT menu FROM menus)**



Retourne les lignes communes (intersection) entre les résultats des deux requêtes.

Requêtes imbriquées (Plus tard)(page 80)