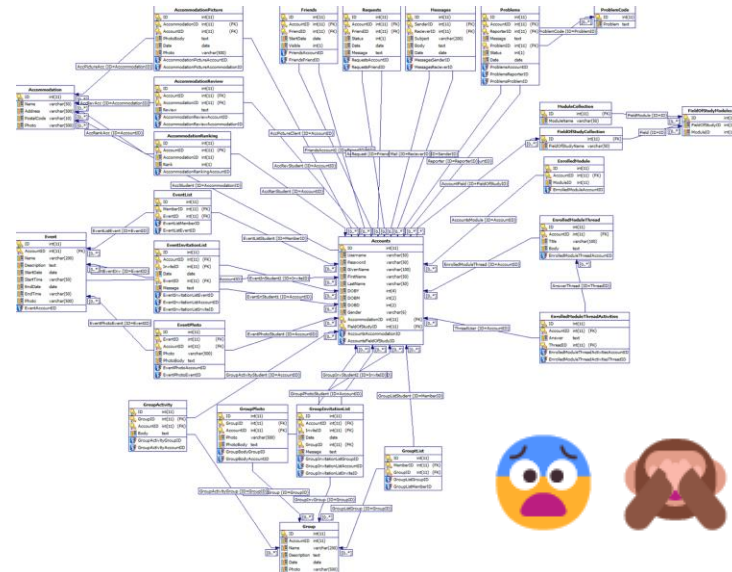


Rencontre 3

Modélisation logique / relationnelle

Bases de données et programmation Web





❖ Conceptuel -> Logique  

❖ Clés 



- ❖ Passer du **modèle conceptuel** au **modèle logique**
 - ◆ C'est l'étape qui précède la création de la **base de données physique**.
 - On détermine la structure de chacune des **tables**.
 - On détermine les **clés primaires** et **étrangères** qui permettront de concrétiser les liens entre les tables et leurs données.
 - ◆ Si notre **modèle conceptuel** est **cohérent** et **complet**, c'est un excellent point de départ pour construire le **modèle logique**.
 - Malgré tout, ce processus ne sera pas qu'une simple recette automatique : nous devons parfois faire des choix et adresser des problématiques.
 - ◆ Une fois le modèle logique terminé, nous pourrons créer la base de données et ses tables.



- ❖ Comme le **modèle logique** est utilisé pour créer les tables de la BD, nous changerons les noms des tables et des champs pour respecter des standards.
 - ◆ En gros, nous utiliserons les standards de Microsoft car nous allons utiliser ses outils.
 - ◆ Il y a aussi les standards adoptés par l'industrie.
 - ◆ Finalement, chaque entreprise ajoute des standards qui lui sont propres.
 - ◆ Voir la présentation [*420_4D5_R03_Standards à respecter.pdf*](#) pour connaître les standards que nous allons adopter ici.



❖ Passer du modèle conceptuel au modèle logique

- ◆ Dans les diapos qui suivent, nous allons aborder les recettes généralement utilisées pour convertir le modèle conceptuel en modèle logique.
 - Entités et attributs
 - Relations
 - Composition et agrégation
 - Généralisation et spécialisation

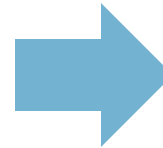
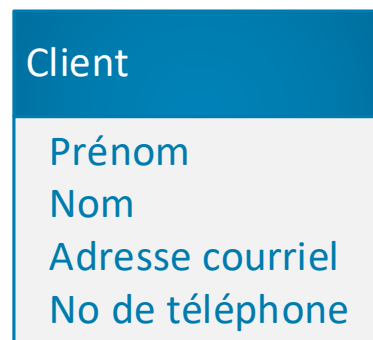


❖ Entité -> Table

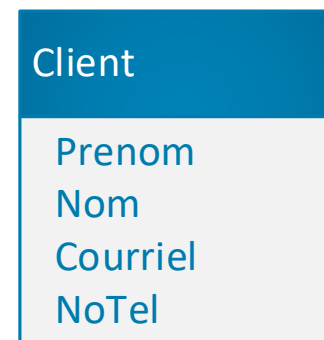
◆ Attributs atomiques 🙌

- Retirer les accents et les espaces des entités et des attributs.
- Quand les noms des attributs sont composés, on utilisera la notation "PascalCase".
- On doit s'assurer que chaque propriété reste facile à comprendre. (No de téléphone peut devenir NoTel, mais pas NT !)

Entité conceptuelle




Entité logique



Exemple : on a simplement réécrit tous les attributs atomiques.



❖ Entité -> Table

- ◆ Il faudra munir la table d'une **clé primaire**. 
 - C'est-à-dire un identifiant unique et stable pour que chaque rangée de données puisse être distinguée des autres.
 - Il existe des clés **naturelles**, **artificielles** ou **composées**.



❖ Entité -> Table

◆ Clé naturelle 🗝️ 🌳

- Sa valeur est **unique** dans la table.
- Sa valeur ne peut pas être **vide**. (*not null*)
- Ce n'est pas une **donnée sensible**, car la clé primaire sera souvent utilisée dans des requêtes / jointures. (Ex. un **NAS**, c'est sensible !)
- Sa valeur est **stable**. (Ne changera pas)
- Sa valeur est **simple** ou courte. (Un entier positif c'est le mieux)

Etudiant	
🗝️ CP	Matricule
	Prenom
	Nom
	Courriel



Exemple : chaque étudiant possède un **Matricule** unique. C'est parfait pour distinguer les étudiants de la table.

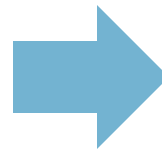



❖ Entité -> Table

◆ Clé **artificielle**

- Même si on a des attributs qui se prêtent bien au rôle de clé **primaire**, on ajoute souvent **une nouvelle propriété qui remplira ce rôle et qui sera un numéro auto-incrémenté.**
- C'est pratiquement toujours le cas de nos jours.
 - Cela fait des « liens » moins intuitifs entre les données mais cela permet d'augmenter la performance de la BD.

Client
Prenom
Nom
Courriel
NoTel



Client
 CP ClientID
Prenom
Nom
Courriel
NoTel



Exemple : Pour la table **Client**, nous avons « créé » la propriété **ClientID**, qui sera tout simplement un nombre entier positif auto-incrémenté et unique pour chaque client.

C'est un des standards de Microsoft de nommer les clés primaires des tables avec le nom de la table suivi de ID.



❖ Entité -> Table

◆ Clé composée 🗝️ 🐙

- Dans certains cas, seule la combinaison de plusieurs propriétés rend unique une rangée. Cette **combinaison** de propriétés peut donc servir de **clé primaire composée**. (Fréquent dans les tables de liaison qui seront vues plus loin.)

Exemple :

« Les paies sont représentées par un **numéro d'employé**, **l'année**, la **semaine** (nombre de 1 à 52), le **nombre d'heures travaillées** et le **taux horaire** de l'employé. »

- **EmployeID** n'est pas une donnée unique : toutes les autres paies de l'employé auront cette valeur. Notons que c'est d'ailleurs une **clé étrangère** : on peut imaginer une table nommée **Employe** qui se sert de cette valeur comme **clé primaire**. **Annee** et **Semaine** ne sont pas des données uniques non plus, bien entendu.

- La combinaison de ces trois valeurs est unique, cela dit ! On a donc une **clé primaire composée**.

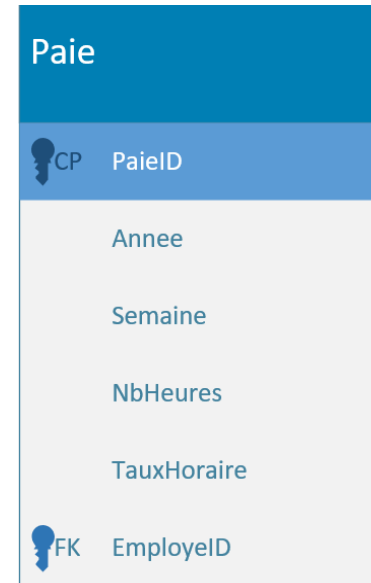
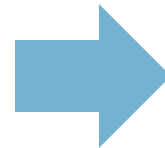
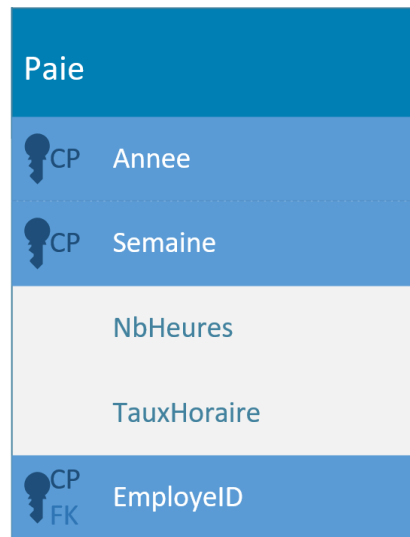
Paie	
🗝️ CP	Annee
🗝️ CP	Semaine
	NbHeures
	TauxHoraire
🗝️ CP 🔑 FK	EmployeID



❖ Entité -> Table

◆ Clé composée 🗝️ 🐙

- **À éviter quand c'est possible** : une clé primaire composée rend plus complexe l'interaction entre certaines données. Une **clé artificielle** peut alors être créée pour la remplacer.

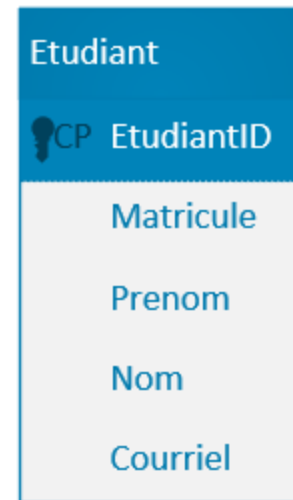
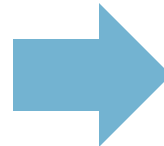
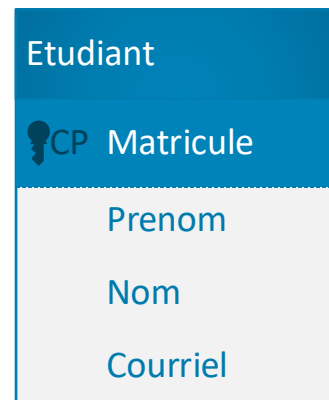


- De plus, si cette entité se retrouve dans une relation 1 à N, la clé artificielle est toujours créée.



❖ De nos jours, pour des raisons de performance, on utilise pratiquement toujours une clé **artificielle**.

◆ Changer Etudiant avec une clé artificielle EtudiantID et Matricule devient un champ.



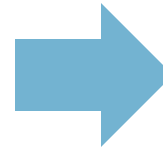


❖ Entité -> Table

◆ Attributs à valeurs multiples 📦

- Si la quantité de valeurs est connue, faible, fixe et que les valeurs sont simples / courtes : on peut ajouter le nombre de champs nécessaires dans la même table.
 - Exemple : si chaque client peut spécifier exactement 1 ou 2 numéros de téléphone, on peut simplement créer deux champs. (Le 2^e sera *null* si nécessaire)

Client
Prénom
Nom
Adresse courriel
No de téléphone [1..2]



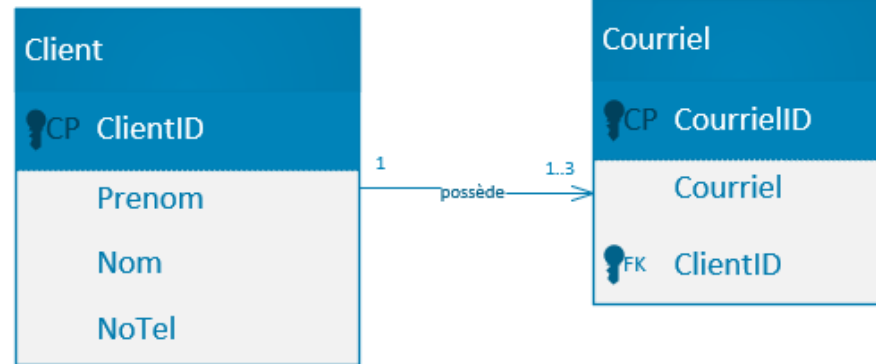
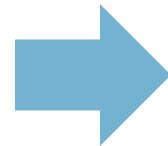
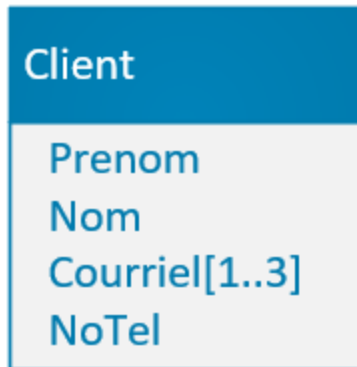
Client
🔑CP ClientID
Prénom
Nom
Courriel
NoTel1
NoTel2



❖ Entité -> Table

◆ Attributs à valeurs multiples 📦

- Si la quantité de valeurs est **indéterminée**, **grande (3 et +)**, **non fixe** ou que les valeurs sont **longues / complexes**, on va plutôt créer une **deuxième table**. Cette deuxième table contient la donnée ainsi qu'une **clé étrangère** faisant référence à la **clé primaire** de la table principale.



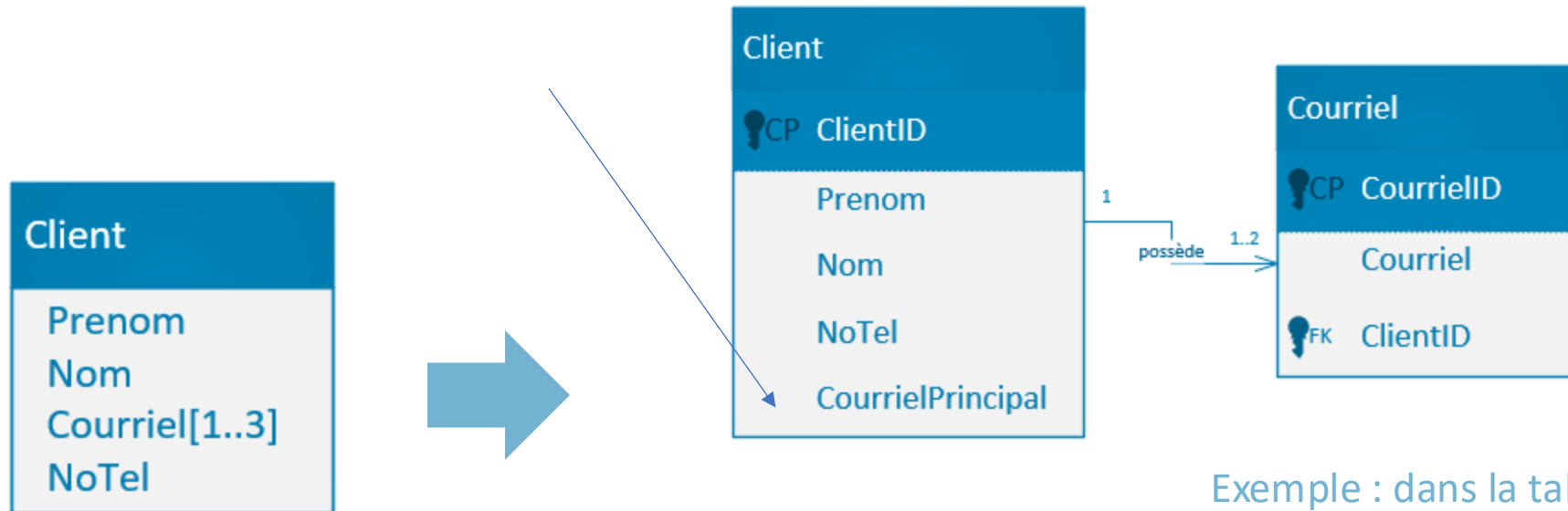
Exemple : dans la table **Courriel**, si un client possède 3 adresses courriel, il y aura trois rangées dont la valeur de **ClientID** sera la même.



❖ Entité -> Table

◆ Attributs à valeurs multiples 📦

- Quand une des valeurs est **plus importante** que les autres, on peut aussi modéliser autrement. Ainsi, si un client a 3 courriels mais réponds plus rapidement à l'un d'entre eux:

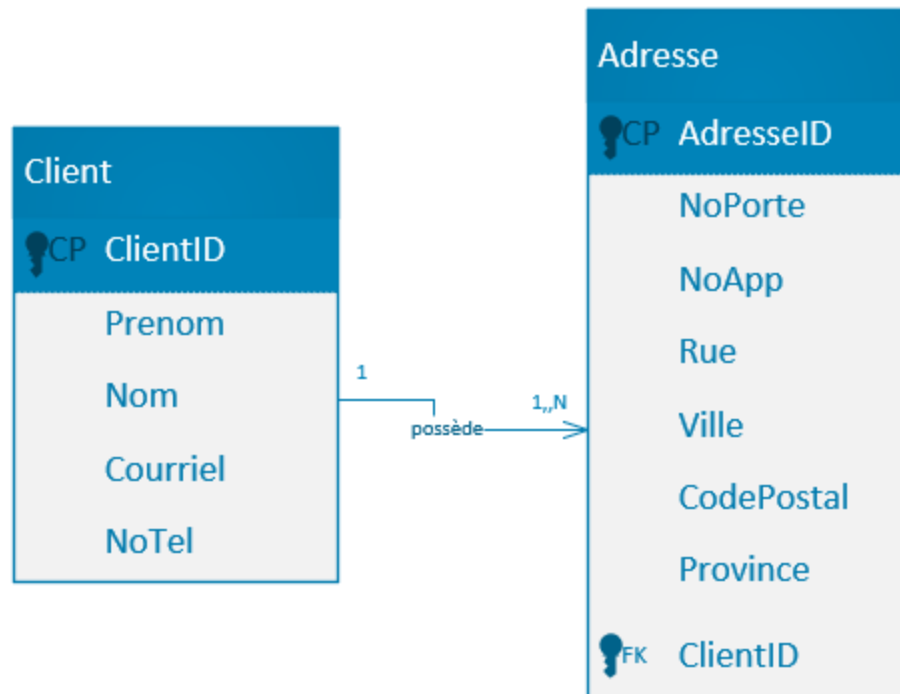


Exemple : dans la table **Client**, il y aura le courriel le plus utilisé et dans la table **Courriel** il y aura les autres courriels du client.



❖ Entité -> Table

- ◆ Attributs **composites** 🐙 : même recette que l'attribut à valeurs multiples !
 - L'attribut composite est **simple** : On met tous ses champs dans la table principale.
 - L'attribut composite est **complexe** : On crée une nouvelle table pour ses champs.
 - Une table avec trop de champs *peut* avoir un impact négatif sur les performances.

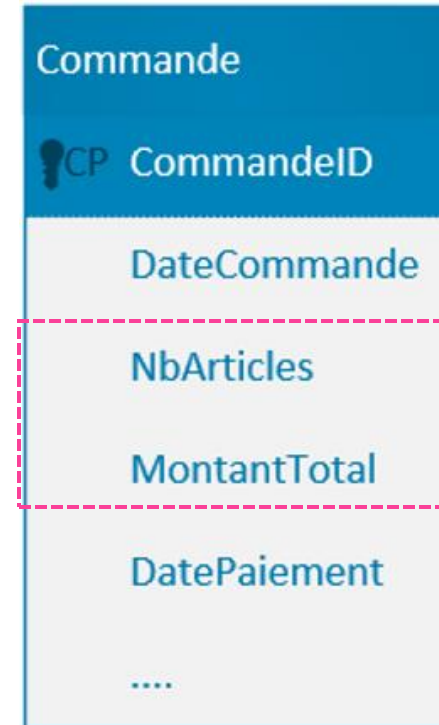
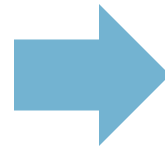


Dans cet exemple, l'**adresse** est divisée en 6 champs précis, alors on l'a isolée dans sa propre table.



❖ Entité -> Table

◆ Attributs **dérivés** 🧐 : On retire tout simplement la \.





❖ Relations

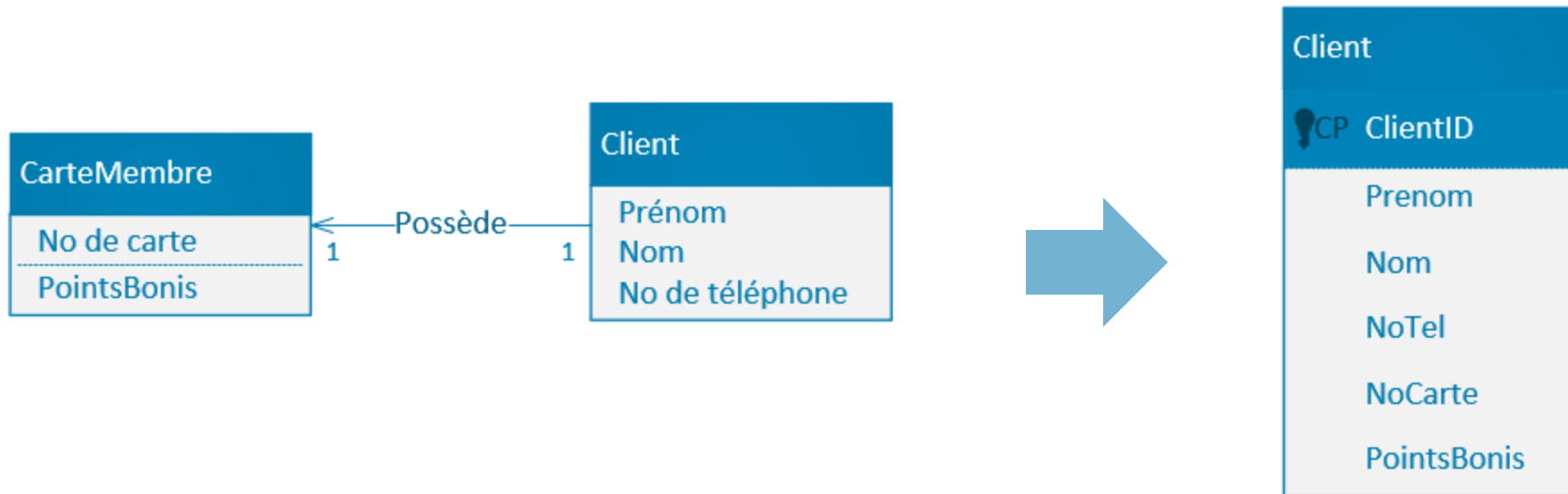
- ◆ Les relations doivent devenir des liens entre des **clés primaires** et des **clés étrangères**, unissant les tables que nous avons produites à l'aide des entités.
 - On se concentre sur les relations entre les clés:
 - Ajouter une clé primaire dans toutes les tables
 - Et **pour le côté plusieurs d'une relation**, ajouter une clé étrangère pour la relier à la clé primaire associée (du côté 1 de la relation).
 - Gardez la direction, l'affichage des **cardinalités** et le **verbe** expliquant la relation dans le cas des associations dirigées.
 - Gardez la composition, agrégation et héritage aussi.



❖ Relations

◆ One-To-One (1-1) : deux choix

1. Une entité est faible / **simple** : on peut carrément l'intégrer à l'autre entité.



Exemple : nous avons une relation One-To-One et l'entité **CarteMembre** était plutôt **simple**. On peut simplement intégrer les propriétés de la carte de membre dans la table du **Client**. Bien entendu, il ne faut pas oublier de choisir une **clé primaire** pour cette table.

Mais si on veut garder les infos des différentes cartes de membre qu'un client aurait eu, on doit absolument les garder dans une entité séparée, avec une date de début et une date de fin...



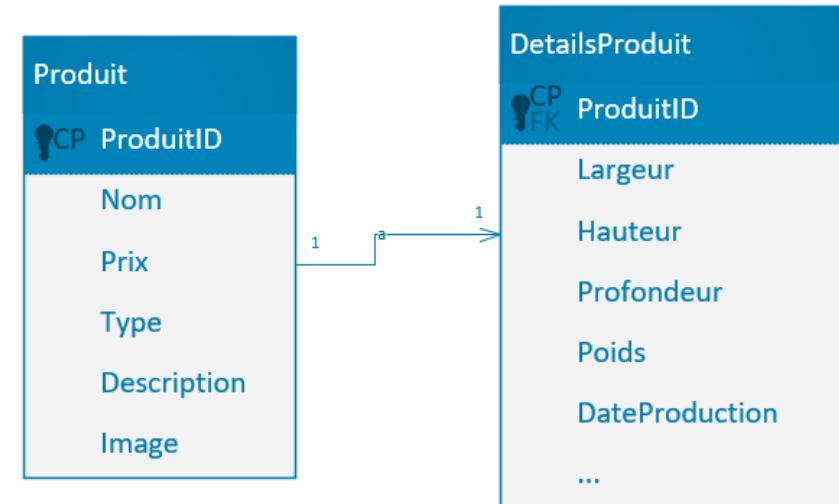
❖ Relations

◆ One-To-One (1-1) : deux choix

2. Si les deux entités sont **complexes**, ont un **cycle de vie** différent ou ont une **granularité** différente, il peut être intéressant de les conserver de manière séparée.

Exemple :

- La table **DetailsProduit** comporte tout simplement de **nombreuses** données supplémentaires pour les produits dans la table **Produit**.
- Pourquoi ne pas unir les tables ? Imaginons que la table **DetailsProduit** contient environ 30 propriétés : cela **alourdit** énormément les **requêtes** sur la table lorsqu'on veut simplement afficher les informations de base de beaucoup de produits.
- Dans cette situation, **DetailsProduit** possède une **granularité** beaucoup plus grande et on sépare donc les tables. Si l'utilisateur souhaite afficher tous les détails d'un produit, nous irons les chercher dans cette autre table.





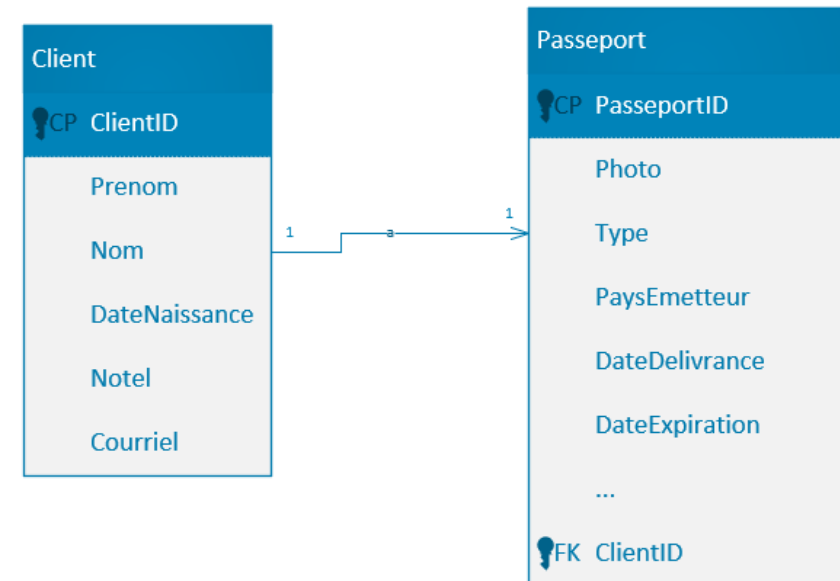
❖ Relations

◆ One-To-One (1-1) : deux choix

1. Si les deux entités sont **complexes**, ont un **cycle de vie** différent ou ont une **granularité** différente, il peut être intéressant de les conserver de manière séparée.

Exemple :

- Les deux entités sont suffisamment **complexes** pour qu'on décide de les conserver séparées pour des raisons de performance.
- Les données du **passport** ont un **cycle de vie** unifié et limité : 5 ou 10 ans. Alors que les données du **client** peuvent être changées individuellement à un rythme différent. Ça peut donc être intéressant de garder les entités séparées pour cette raison.





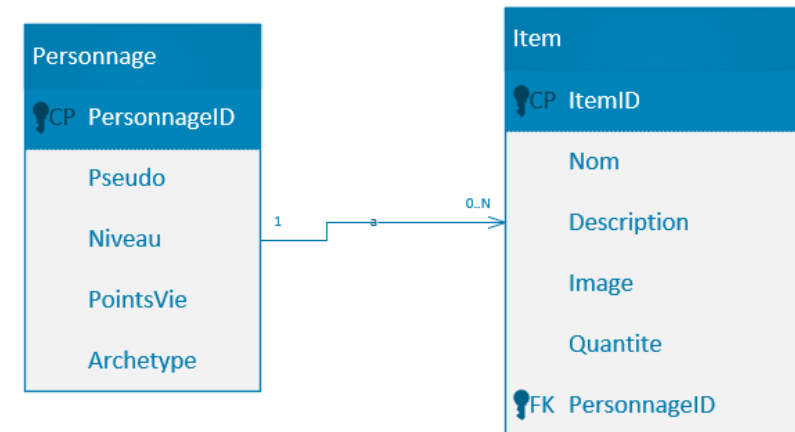
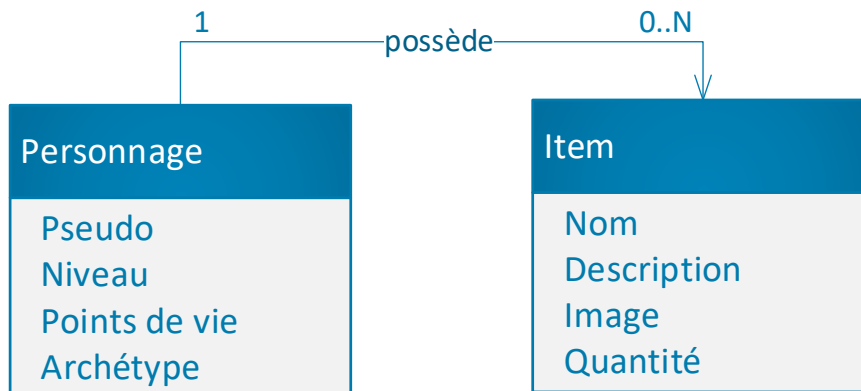
❖ Relations

◆ One-To-Many (1-N)

- Dans ce cas, la solution est **d'ajouter une clé étrangère** dans l'entité du côté Many (N) pour la relier à la clé primaire de l'entité du côté One (1).

Exemple :

- Chaque **personnage** peut **posséder** plusieurs **items** dans son inventaire.
- On ajoute simplement l'**id du personnage** dans la table des items en tant que **clé étrangère** pour identifier à quel **personnage** cet **item** appartient.





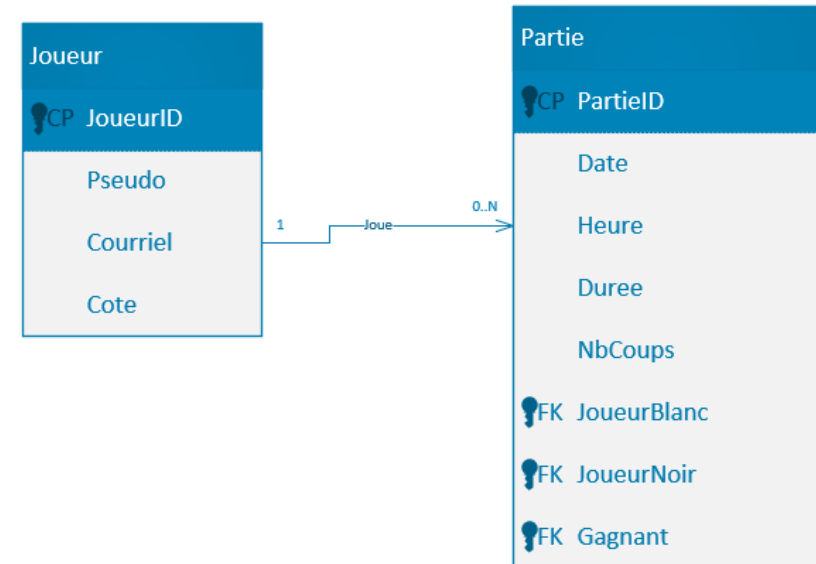
❖ Relations

◆ One-To-Many (1-N)

- Nous avons parfois plusieurs relations (1-N) entre deux tables. Comme ici les joueurs qui participent à une partie d'échec.

- Dans Partie, on a 3 références vers JoueurID de la table Joueur:
 - JoueurBlanc
 - JoueurNoir
 - Gagnant

On est obligé de leur donner des noms différents car une table ne peut pas avoir des champs ayant le même nom. De plus, cela ne serait vraiment pas clair d'avoir 3 X JoueurID dans une table.





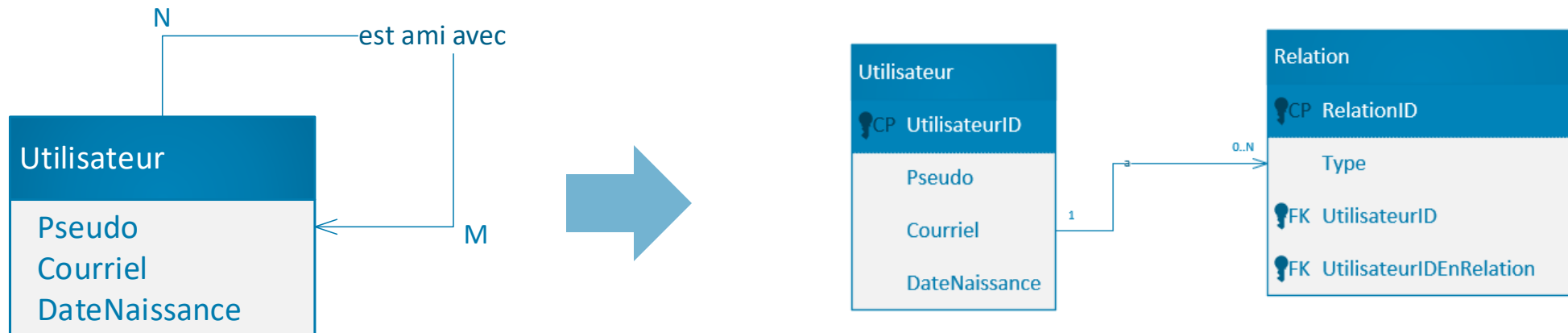
❖ Relations

◆ Many-To-Many (N-M)

- Il faut généralement créer une toute nouvelle table qui pourra contenir des **paires de clés étrangères** afin de représenter les liens entre deux entités.

Exemple 1 :

- Chaque **utilisateur** peut être en relation avec plusieurs autres **utilisateurs**. (Relation récursive N-M)
- On a donc simplement créé une nouvelle table (**Relation**) qui contiendra toutes les **paires d'utilisateurs en relation**.
- Pourrait-on ajouter une **clé artificielle** dans **Relation** pour identifier chaque rangée ? Oui, on peut. C'est utile si on veut faire la liaison avec une autre entité comme **Sortie** pour enregistrer les sorties que feraient ces deux utilisateurs...





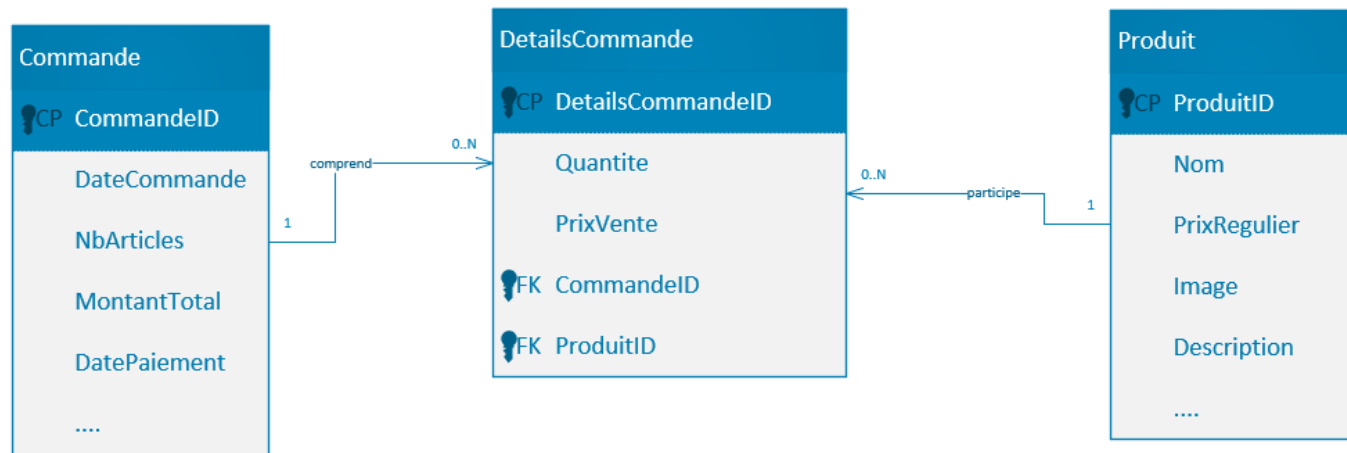
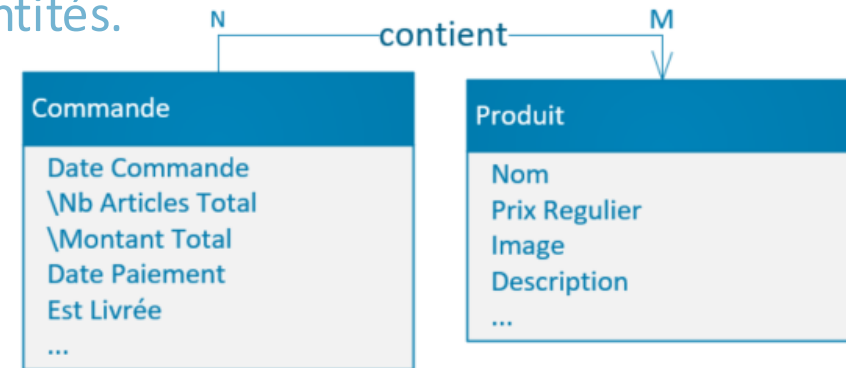
❖ Relations

◆ Many-To-Many (N-M)

- Il faut généralement créer une toute nouvelle table qui pourra contenir des **paires de clés étrangères** afin de représenter les liens entre deux entités.

Exemple 2 :

- Chaque **Commande** peut contenir plusieurs **Produit**.
Chaque **Produit** peut être dans plusieurs **Commande**.
- On crée donc une table de liaison nommée **DetailsCommande**.
On en profite même pour noter des informations supplémentaires : **Quantite, PrixVendu**.

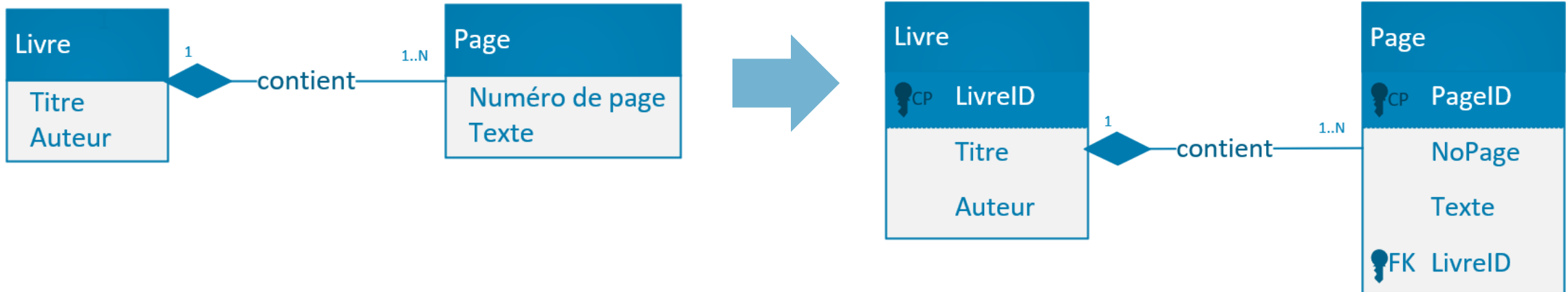




❖ Composition et agrégation

◆ Cette notion ne change pas dans le modèle logique.

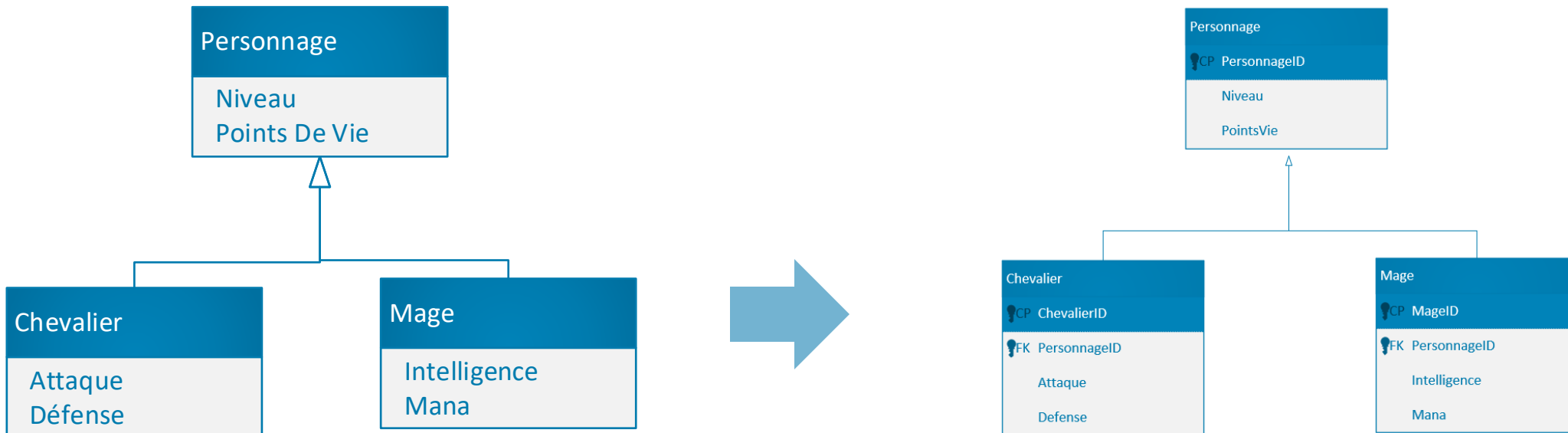
- On ajoute les clés étrangères comme d'habitude dans une relation 1 à N tout en gardant le symbole de composition ou d'agrégation selon le cas.





❖ Généralisation et spécialisation 🐟 🐕

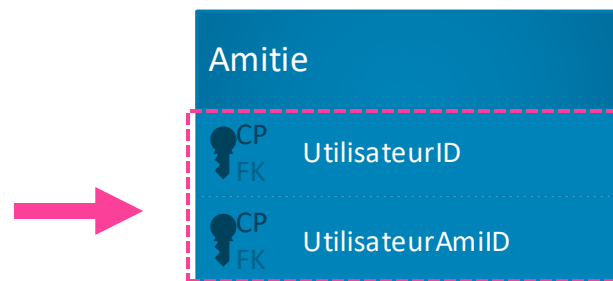
- ◆ Une table par sous-type et une table pour le super-type.
- ◆ Une **clé étrangère** vers le **super-type** dans chaque table des **sous-types**.
 - On ajoute une **clé primaire** aux sous-types. Elle servira pour les relations entre le sous-type et d'autres tables.
 - On met la **clé étrangère** tout de suite après la clé primaire, pour **renforcer** la notion **d'héritage**.



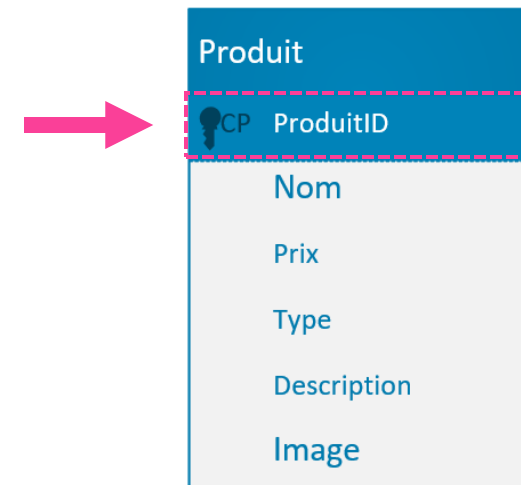


❖ Récapitulons le vocabulaire associé aux clés

- ◆ **Clé primaire** 🗝️ 🏆 (d'une table) : Combinaison d'une ou plusieurs propriétés qui rend **unique** une rangée dans une table. **Identifie** une rangée de données. Attention, c'est la **combinaison** de propriétés qui doit être **unique**, et non chaque propriété, individuellement.
- ◆ **Clé étrangère** 🗝️ 🔗 : Propriété qui fait référence à une clé primaire dans une autre table.
- ◆ **Clé composée** 🗝️ 🦸 / **Clé primaire composée** : Clé primaire composée de plusieurs propriétés.
- ◆ **Clé artificielle** 🗝️ 🧰 : Propriété simple créée spécifiquement pour servir de clé primaire.
- ◆ **Clé naturelle** 🗝️ 🌳 : Propriété déjà présente au stade du modèle conceptuel et qui remplit adéquatement le rôle de clé primaire.



Dans cette table de liaison, nous avons une **clé primaire composée** de deux **clés étrangères** !



Dans cette table, nous avons une **clé primaire artificielle**.