Bases de Données Relationnelles et langage SQL.

Table des matières

I. Définitions

1) Base de données relationnelle

En informatique, une **base de données** (BD) est un **ensemble de données** qui ont été stockées sur un support informatique et qui ont été **organisées et structurées** de manière à ce qu'on puisse facilement consulter et modifier leur contenu.

Par exemple, l'ensemble des clients d'un fournisseur d'accès internet ou l'ensemble des étudiants d'une université peuvent constituer des bases de données. De façon générale, l'intérêt d'une base de données est qu'elle peut contenir un nombre extrêmement important de données informatiques.

Une **base de données relationnelle** (BDR) est organisée en **tables**, encore appelées **relations**. Celles-ci peuvent être représentées graphiquement sous la forme de tableaux de type EXCEL. En voici un exemple ci-dessous qui peut être la liste des clients d'une entreprise. Nous appellerons cette table la table **Clients** :

Table Clients

id	nom	prenom	age	adresse_mail	date_inscript
1	LUSSAC	François	32	lfran1@gmail.com	2021-06-07
2	MALSHERBES	Jeanne	48	mljean@wanadoo.fr	2020-10-19
3	KATZ	Roger	22	lxsir@hotmail.com	2022-03-24
...	...	...	...	...	...

2) Organisation d'une BDR : le modèle entité - association E/A

a) Introduction

Considérons une entreprise de vente en ligne ; des clients s'inscrivent et passent des commandes pour se faire livrer différents types de produits. Supposons que :

- chaque client soit identifié par son nom, prénom, un numéro de téléphone ;
- chaque produit soit caractérisé par son nom, sa réference et son prix unitaire.

L'entreprise gère cela grâce à une BDR. Une première idée est de créer une seule table, par exemple de la façon suivante où on suppose que chaque commande possède un numéro de référence :

Table Commandes_clients

nom	prenom	tel	ref_commande	nom_prod	ref_produit	prixU
Laroche	Charles	065678	126L4	livre	03456	23
Katz	Chloé	076244	149D2	disque	05971	15
Laroche	Charles	065678	126L4	livre	02781	52
Atalaya	Dalila	069324	291C8	classeur	09367	3
Katz	Chloé	076244	529U6	livre	03456	23
...	...	...	...	...	...	

Clairement, cette représentation possède de nombreux défauts :

- Les informations sur les clients et sur les produits sont redondantes : la même information (nom, prenom, tel) ou (nom_prod, ref_prod, prixU) est stockée plusieurs fois sur des lignes différentes : il y a un gâchis de mémoire.

- En cas de modification des coordonnées d'un client (ici son numéro de téléphone), il faut reécrire son nouveau numéro de téléphone sur toutes les lignes le concernant. Idem si on change le prix unitaire d'un produit.
- Si un client annulle sa commande, il faudra supprimer toutes les lignes concernant cette commande, ce qui risque de supprimer le client s'il n'a fait qu'une seule commande.
- De plus, un client peut posséder plusieurs numéros de téléphones, une commande est aussi caractérisée par une date de commande, une date de livraison et on pourrait imaginer qu'un livre par exemple soit aussi caractérisé par un auteur, une date de parution, une maison d'édition, etc ... Bref, il faudrait multiplier le nombre de colonnes de la table ce qui donne rapidement de grosses tables difficiles à gérer.

On souhaite donc organiser la BDR de sorte qu'elle économise de la place mémoire (suppression des redondances), qu'elle soit facile maintenir et qu'elle permette des modifications sûres (c'est à dire sans perdre de données essentielles).

C'est pour cela qu'une BDR contient la plupart du temps plusieurs tables qui sont reliées entre elles par des associations. Une bonne description théorique de cette organisation est le modèle entité-association développé dans les années 1970.

b) Entité - Classe d'entités

Définition 1. *Entité*

Une **entité** est un objet, concret ou abstrait, qui peut être identifié de façon précise et distincte d'autres objets en énumérant un ensemble de propriétés.

Exemples :

Définition 2. *Classe d'entités*

Une **classe d'entités** est un ensemble d'entités de *même nature* et décrites par les *mêmes propriétés*.

Remarque :

Il résulte de la définition qu'une entité élément d'une classe d'entités donnée n'y est présente qu'une seule fois (c'est à dire en un seul exemplaire). Si E est une classe d'entités et e une entité de cette classe on écrira : $e \in E$.

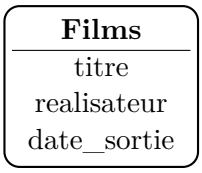
Chaque propriété est désignée par un *identifiant* qu'on appelle **attribut**. Une classe d'entité est représentée graphiquement par un rectangle à bords arrondis qui contient :

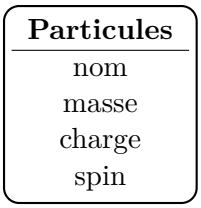
- En haut le nom de la classe d'entités
- En bas, la liste des attributs désignant les propriétés

Exemple : la classe d'entités **Personnes** pourra être représentée de la façon suivante :

Chaque entité appartenant à la classe **Personnes** aura donc 5 propriétés qui seront des *valeurs particulières* des attributs de cette classe. Bien entendu, lorsqu'un attribut est présent dans la liste, il n'apparaît qu'une seule fois !

Voici deux autres exemples de classes d'entités :





Définition 3. *Domaine d'un attribut*

Étant donné un attribut `a` d'une classe d'entités, on appelle **domaine** de `a`, et on note `Dom(a)`, *l'ensemble des valeurs particulières* qui peuvent être choisies pour cet attribut, au niveau des entités éléments de cette classe.

En pratique pour nous c'est très simple : le domaine d'un attribut sera quelque chose parmi : entier, réel, texte, date.

Exemple : la classe d'entités **Personnes** contient les attributs avec leurs domaines suivants :

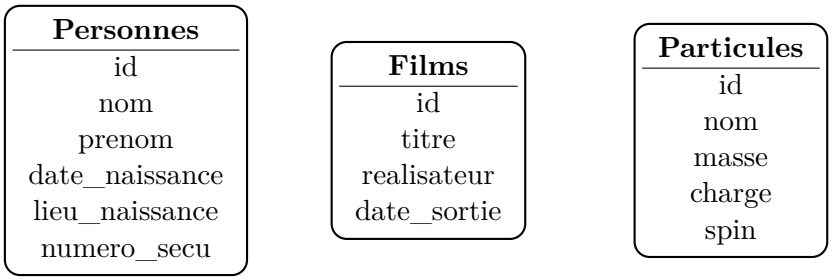
- `nom` :
- `prenom` :
- `date_naissance` :
- `lieu_naissance` :
- `numero_secu` :

Définition 4. *Clé d'une classe d'entités*

Une **clé** d'une classe d'entités est une liste d'attributs de cette classe dont les valeurs particulières permettent de désigner une et une seule entité de cette classe.

Exemple avec la classe **Personnes**

En pratique, pour éviter tout problème, on crée un attribut spécial qu'on appelle id de domaine entier positif ($\mathbb{N}$) qui va servir de clé : chaque entité sera caractérisée par une unique valeur particulière de **id** qui sera son identifiant. Les classes d'entités **Personnes**, **Films** et **Particules** prendront donc la forme :

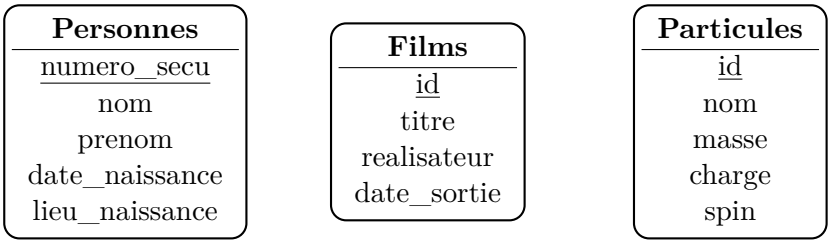


Définition 5. *Clé primaire d'une classe d'entités*

La **clé primaire** d'une classe d'entité est une de ses clés possibles de cette classe qu'on a désignée comme clé primaire. Les autres clés possibles seront alors appelées *clés secondaires*.

Remarques :

- Le choix de la clé primaire est totalement libre : on prend une clé parmi toutes les clés possibles.
- En pratique on prend très souvent **id** comme clé primaire.
- Par convention, la clé primaire sera placée en premier dans la liste des attributs d'une classe et sera soulignée. On pourra donc obtenir par exemple :



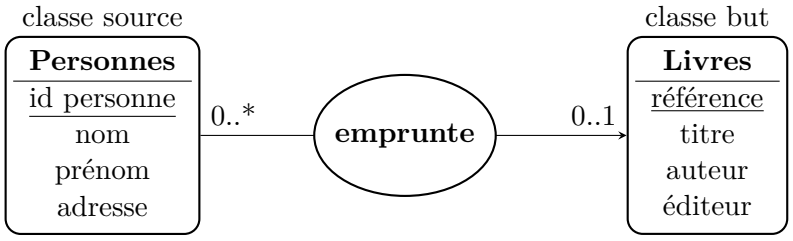
c) Association

Définition 4. *Association*

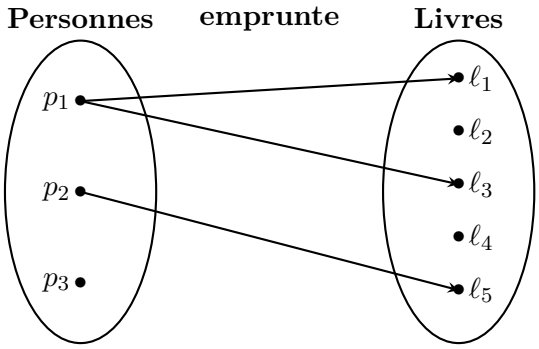
Une **association** est un lien entre deux classes d'entités. Plus précisément, si E_1 et E_2 désignent deux classes d'entités alors une association est un ensemble de couples (e_1, e_2) avec $e_1 \in E_1$ et $e_2 \in E_2$. E_1 est la classe source et E_2 la classe but.

Plusieurs représentations graphiques sont possibles pour une association. Prenons le cas d'une bibliothèque dont la BDR contient les deux classes d'entités **Personnes** et **Livres**. L'association entre ces deux classe est **emprunte**.

- Représentation conventionnelle :

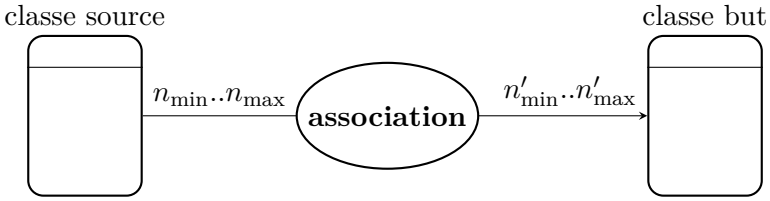


- Représentation sagittale (peu utilisée mais permet d'éclaircir la notion)



Cardinalités d'une association

On peut remarquer sur le premier schéma précédent les symboles 0..* et 0..1 : ils représentent les **cardinalités** de l'association. Celles-ci se présentent toujours sous la forme :



ce qui signifie que :

-
-

Dans l'exemple précédent cela donne :

- chaque personne peut emprunter
- chaque livre peut être emprunté par

En pratique $n_{\min}$ et $n'_{\min}$ sont toujours 0 ou 1 ; $n_{\max}$ et $n'_{\max}$ valent toujours 1 ou $*$, ce dernier symbole signifiant *tout nombre strictement supérieur à 1* (on n'est jamais intéressé par savoir quel est exactement ce nombre).

On a donc toujours affaire à l'une des quatre cardinalités suivantes (aussi bien du côté de la classe source que de la classe but) :

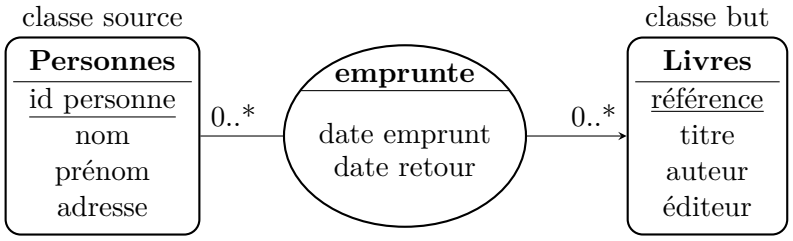
- 0..1 : une entité est reliée à au plus une entité de l'autre classe ;
- 0..* : une entité est reliée à 0 ou à plusieurs (>1) entités de l'autre classe ;
- 1..1 : une entité est reliée à une et une seule entité de l'autre classe ;
- 1..* : une entité est reliée à au moins une entité de l'autre classe ;

Attributs d'une association

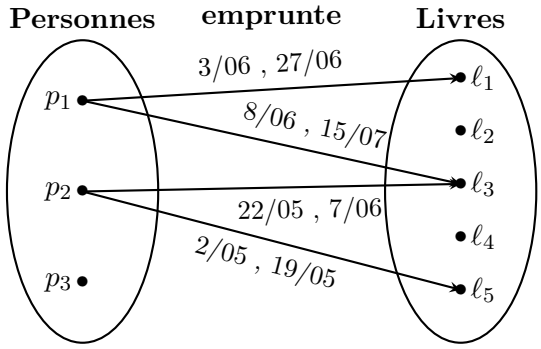
Pour augmenter l'efficacité et la précision du modèle E/A, les associations peuvent aussi posséder des attributs. Dans le diagramme sagittal de l'association les valeurs particulières de ces attributs sont notées à côté des flèches.

Reprenons l'exemple de la bibliothèque et de la relation **emprunte**. On sait qu'un emprunt est caractérisé par une date d'emprunt et une date de retour du livre. On peut donc munir l'association de ces deux attributs. Ceci permet alors de changer la cardinalité du côté Livres : un livre peut alors être emprunté par 0 ou plusieurs personnes. Cela donne :

- Représentation conventionnelle :



- Représentation sagittale :



- Fondamentalement, l'association est devenue un ensemble de 4-uplets de la forme :

$$\textbf{emprunte} = \{\,(p_1, \ell_1, 3/06, 27/06), (p_1, \ell_3, 8/06, 15/07), (p_2, \ell_1, 22/05, 7/06), (p_2, \ell_5, 2/05, 19/05)\,\}$$

Association inverse

Toute association admet une association inverse. En effet, une association étant un ensemble de couples d'entités $(e_1, e_2) \in E_1 \times E_2$ (E_1 = classe source et E_2 = classe but), on peut poser :

Définition 5. *Association inverse*

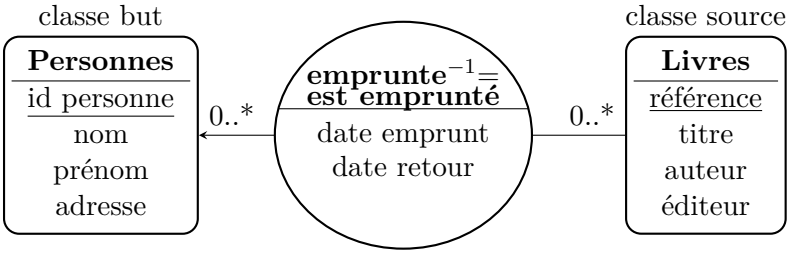
Soit A une association. On appelle **association inverse** de A et on note A^{-1} l'ensemble des couples ci-dessous :

$$A^{-1} = \{\,(e_2, e_1) \mid (e_1, e_2) \in A\,\}$$

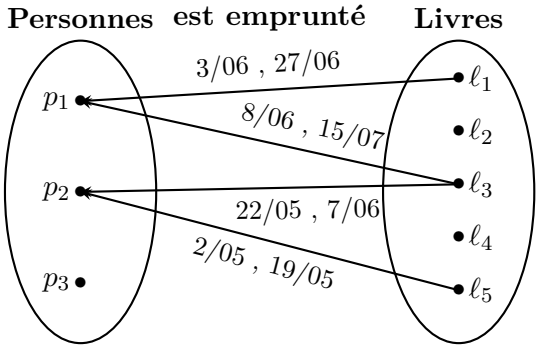
Si A possède des attributs, alors A^{-1} possède les mêmes attributs.

Reprenons l'exemple précédent de la relation **emprunte** entre la classe **Personnes** et la classe **Livres**. Voici les représentations graphiques de l'association inverse :

- Représentation conventionnelle :

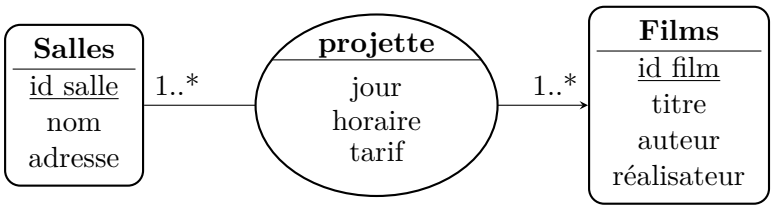


- Représentation sagittale :

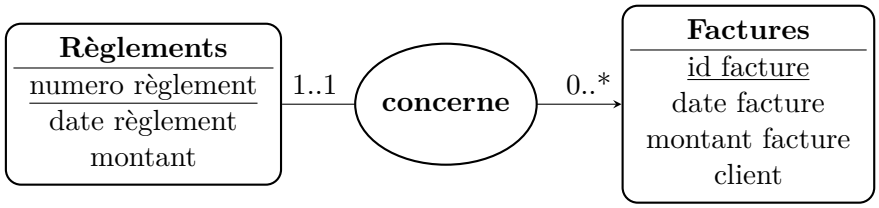


On voit donc que les cardinalités ne sont pas changées ; seul le sens des flèches ainsi que les classes source et but sont est inversés.

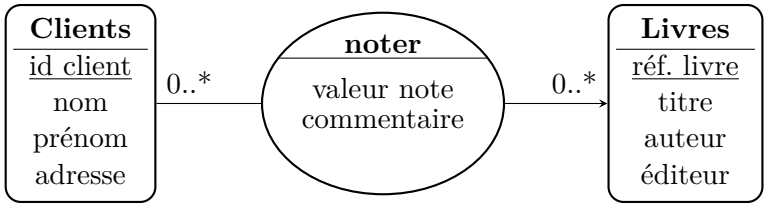
Exemple 1 : des salles de cinéma projettent des films. Dans un modèle E/A, cela peut donner une représentation graphique de la forme :



Exemple 2 : une entreprise émet des facture et contrôle leur règlement par ses clients. Un schéma possible E/A pourrait être :



Exemple 3 : des clients d'une librairie peuvent s'ils le souhaitent noter et commenter des livres. On peut modéliser cela par :

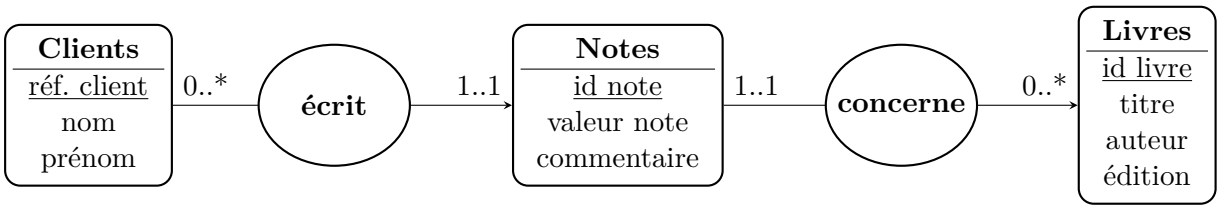


d) Lois de transformation d'une association en classe d'entités

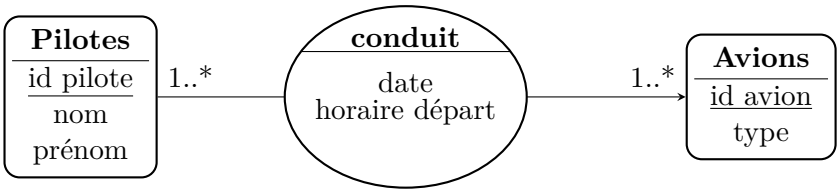
Loi 1 : **transformation** x..*⟶ y..* **en** x..*⟶ 1..1 **suivi de** 1..1 ⟶ y..*

*Une association entre deux classes E_1 et E_2, avec des **cardinalités** 0..∗ **ou** 1..∗ **sur les deux pattes** peut être remplacée par une classe entité avec des **cardinalités 1..1 de chaque côté de celle-ci**. Ceci est même fortement conseillé pour des raisons de clarté et de facilité de maintenance comme on va le voir plus tard. Si l'association contient des attributs, ceux-ci sont affectés à la classe d'entités qui la remplace.*

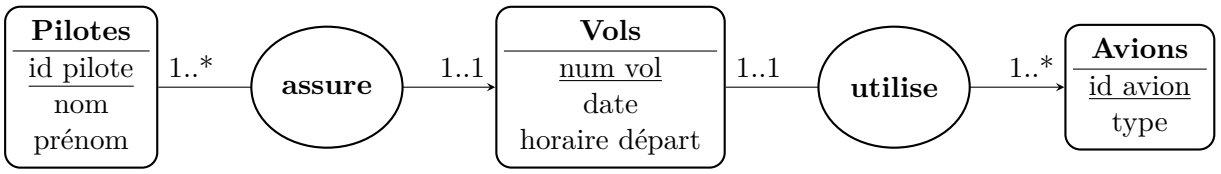
Exemple 1 : revenons à l'exemple 3 précédent qui concernait les clients d'une librairie qui notent et commentent des livres. On peut avantageusement remplacer l'association noter par :



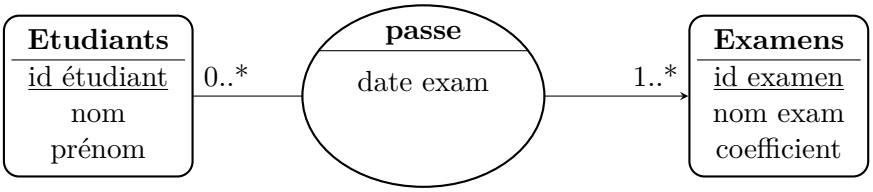
Exemple 2 : une compagnie aérienne possède un contingent de pilotes ainsi que plusieurs avions. Pour chaque vol on doit associer un pilote et un avion. Une première possibilité est de créer des E/A comme ci-dessous :



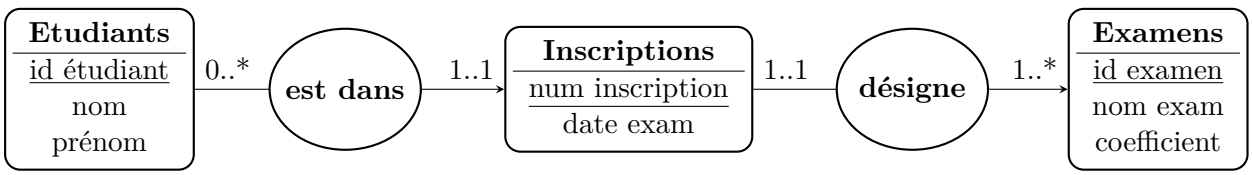
... mais on préfère ceci :



Exemple 3 : une université doit gérer le passage des examens par ses étudiants. Une idée est de procéder de la façon suivante :



... mais il est préférable de faire :



Loi 2 : **pas d'attributs pour les associations !**

Il est toujours possible d'écrire un modèle E/A dans lequel les associations ne contiennent pas d'attributs. Toutes les informations qui étaient véhiculées par les attributs d'une association sont transférées dans des classes d'entités

C'est déjà ce qu'on a fait dans les trois exemples ci-dessus mais on verra par la suite que c'est ce choix qui est systématiquement fait lorsqu'on transpose le modèle E/A aux bases de données relationnelles.

II. Application aux BDR

1) Définitions

Suite à notre étude du modèle entité/association, nous pouvons maintenant poser :

Définition 1. *Base de donnée relationnelle*

Une base de données relationnelle (BDR) est une *collection de classes d'entités* qu'on appelle **tables** ou encore **relations** (c'est donc les noms particuliers donnés aux classe d'entités dans le cas d'une BDR) et qui sont reliées entre elles par des associations.

Les associations ne possèdent aucun attribut.

Les tables (relations) sont la transcription directe des classes d'entités mais une partie du vocabulaire est spécifique. Prenons la table **Clients** d'une entreprise, introduite au début du **I.** :

- Représentation d'une table sous la forme d'un tableau (type EXCEL) :

Table Clients

id	nom	prenom	age	adresse_mail
0	LUSSAC	François	32	lfran1@gmail.com
1	MALSHERBES	Jeanne	48	mljean@wanadoo.fr
2	KATZ	Roger	22	lxsir@hotmail.com
...	...	...	...	...

- Représentation héritée du modèle E/A :

La notion de **clé** d'une table correspond exactement à la notion de clé d'une classe d'entités.

En particulier la **clé primaire** d'une table est la transcription exacte de la clé primaire d'une classe d'entités. Lorsqu'elle existe, celle-ci sera donc donnée en premier et soulignée dans la liste des attributs.

Remarque :

Tout comme pour une classe d'entités, une table de BDR peut très bien ne pas avoir de clé primaire. En général on en crée une spécialement en munissant la table d'un attribut spécial `id`.

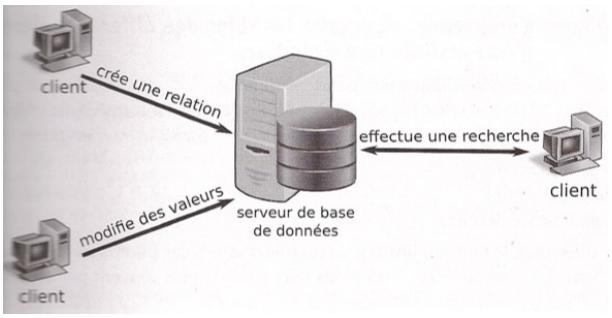
2) SGDBR - Modèle Client / Serveur

Pour communiquer avec une BDR, il est nécessaire d'utiliser un système logiciel appelé **Système de Gestion de Base de Données Relationnelle : SGBDR**.

La plupart des SGDBR fonctionnent sur le **modèle client - serveur** :

- Un **serveur** est un gros ordinateur doté de beaucoup de mémoire de masse et donc capable de stocker beaucoup d'informations : c'est là que sont stockées toutes les tables et les associations format la BDR.
- Un **client** est un autre ordinateur qui se connecte au serveur, via un réseau local (dans une entreprise) ou via internet (dans le cas général), et lui envoie des **requêtes** destinées à :
  - — créer, modifier ou suprimer des tables et associations dans la BDR ;
  - — chercher des données dans une ou plusieurs tables de la BDR, les modifer ou encore les supprimer ;

  Le serveur exécute ces requêtes et envoie éventuellement une réponse au client.
- Plusieurs clients peuvent se connecter simultanément au serveur.
- Les requêtes sont rédigées dans un **langage** qui à l'heure actuelle est très souvent le **langage SQL** : *Structured Query Langage*. C'est le langage le plus répandu pour interagir avec les BDR. Il a été créé dans les années 1970 et c'est devenu un standard en 1986.



Le SGDBR est le logiciel grâce auquel on écrit les requêtes, qui les envoie au serveur et qui reçoit les réponses de celui-ci.

Il existe actuellement plusieurs SGBDR : Oracle Database, SQLite, PostegreSQL, MS Access et MySQL. En ce qui nous concerne, nous utiliserons SQLite.

3) Contraintes du langage SQL

Dans le langage SQL les noms des tables et des attributs doivent obéir au format suivant :

- Ils ne peuvent utiliser que des caractères appartenant à l'ensemble :

$$E = [a \dots z] \cup [A \dots Z] \cup [0 \dots 9] \cup \{\_\}$$

- ils doivent commencer par une lettre ;
- deux caractères _ (soulignement) ne peuvent pas se suivre ;
- ils ne peuvent pas contenir plus de 128 caractères ;
- la casse n'a pas d'importance. Autrement dit : `nom`, `Nom`, `nOm` ou `NOM` c'est la même chose !

Autrement dit, les caractères accentués `é`, `è`, `à`, *etc ..., spéciaux* `ç`, `#`, *etc ..., de ponctuation* `?`, `!` *ainsi que l'espace sont interdits.*

Exemples :

C'est pour cela que les attributs de la table **Clients** ont été nommés `prenom` (et non `prénom`), `adresse_mail` (et non `adresse mail`).

On a déjà vu que le domaine d'un attribut était l'ensemble des valeurs particulières qu'il est possible de choisir pour cet attribut. Dans le langage SQL, les domaines sont **prédéfinis** et ne peuvent pas être choisis librement. Le choix est néanmoins assez vaste et nous n'en citerons que quelques uns [1] :

- Le domaine `INTEGER` : il correspond aux entiers relatifs ($\mathbb{Z}$).
- Les domaines `FLOAT` (codage sur 4 octets, simple précision) et `DOUBLE` (codage sur 8 octets, double précision) : ils correspondent aux nombres flottants ($\mathbb{F} \subset \mathbb{R}$).
- Le domaine `TEXT` : c'est l'ensemble des chaînes de caractères (permet de stocker des chaînes ayant au plus jusqu'à 2^{16} caractères).
- Le domaine `DATE` : il permet de stocker des dates au format AAA-MM-JJ

Tous ces domaines sont munis d'une relation d'ordre totale naturelle permettant de réaliser des comparaisons : relation d'ordre classique pour les entiers et les flottants, ordre lexicographique pour les chaînes de caractères, ordre chronologique pour les dates.

Bien que faisant partie intégrante de la mise en œuvre d'une BDR, aucune connaissance précise sur les domaines n'est exigible aux concours.

Schéma relationnel d'une table :

Le schéma relationnel d'une table est la liste de ses attributs suivis de leur domaine. Par exemple pour la table **Clients** nous aurons :

Clients = (id : `INTEGER`, nom : `TEXT`, prenom : `TEXT`, age : `INTEGER`, adresse_mail : `TEXT`)

4) Implémentation des associations : clés étrangères

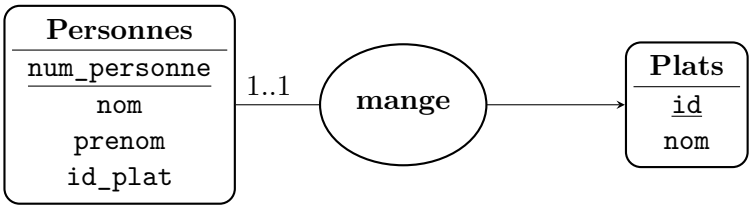
On a déjà dit que dans une BDR, les associations ne contiennent jamais d'attributs. De plus, on n'implémente que les associations ayant les cardinalités suivantes :

x et y valant 0 ou 1. *Autrement dit une cardinalité de la forme* $x..* \longrightarrow y..*$ *n'est pas envisagée : elle est automatiquement transformée en* $x..* \longrightarrow 1..1$ *suivi de* $1..1 \longrightarrow y..*$ *selon la loi 1 vue précédemment.*

En réalité, l'association n'existe pas en tant que telle dans la BDR. On l'implémente en créant un attribut particulier qu'on appelle **clé étrangère** placée soit dans la table source soit dans la table but.

Exemple 1 : considérons par exemple une table **Personnes** et une table **Plats** en supposant qu'une personne ne mange qu'un seul plat :

1. Cette liste n'est pas exhaustive, seuls les principaux types sont décrits ici. Pour plus d'informations, faire une recherche sur internet.



L'attribut `id_plat` de la table **Personnes** (table source) est une clé étrangère ; elle est associée à la clé primaire `id` de la table **Plats** (table but) et permet d'associer à chaque entité de **Personnes** un plat bien précis. Par exemple on peut avoir :

Plats

id	nom
0	couscous
1	paella
2	choucroute
3	tartiflette

Personnes

num_personne	nom	prenom	id_plat
0	Lam	Georges	1
1	Ulrich	Charlotte	2
2	Weyl	Sylvie	0
3	Dufour	René	1
4	Kriss	Dalila	3
...	...	...	...

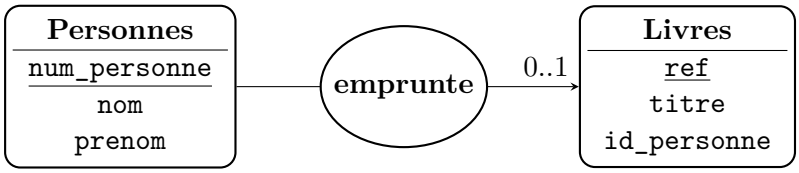
Supposons maintenant que certaines personnes ne souhaitent pas manger : dans ce cas la cardinalité à gauche devient 0..1 et certaines entités de **Personnes** ne seront pas en relation avec une entité de la table **Plats**.

Dans ce cas, le langage SQL dispose de la constante prédéfinie `NULL` qui signifie **rien** et qui joue le même rôle que `None` en langage Python. La clé étrangère `id_plat` pourra alors contenir soit l'identifiant d'un plat (clé primaire), soit la constante `NULL` en l'absence d'association.

Personnes

num_personne	nom	prenom	id_plat
0	Lam	Georges	1
1	Ulrich	Charlotte	2
2	Weyl	Sylvie	NULL
3	Dufour	René	1
4	Kriss	Dalila	3
...	...	...	...

Exemple 2 : on revient à l'exemple de la bibliothèque en supposant qu'un livre est emprunté par 0 ou 1 personne. On aura donc l'association :



La clé étrangère `id_personne` est placée dans la table **Livres** (table but) et fait référence à la clé primaire `num_personne` de la table **Personnes** (table source) pour réaliser l'association. Comme sa cardinalité vaut 0..1, `NULL` peut être une valeur particulière de `id_personne`

Personnes

num_personne	nom	prenom
0	Lam	Georges
1	Ulrich	Charlotte
2	Weyl	Sylvie
3	Dufour	René
4	Kriss	Dalila
...	...	...

Livres

ref	titre	id_personne
0	Les misérables	2
1	Cuisinez sans stress	3
2	Le hussard sur le toit	NULL
3	Dune	3
...	...	...

III. Requêtes portant sur une seule table

Une BDR ayant été installée sur un serveur notre but est maintenant d'étudier quelques requêtes qu'on peut lui dresser en langage SQL. Conformément au programme, seules des requêtes permettant d'effectuer une lecture ou une recherche dans une table seront abordées. Les requêtes de création, modification, suppression de données ou de tables sont hors programme.

Nous allons supposer que nous disposons d'une table **Clients** de la forme ci-dessous :

Clients

id	nom	prenom	genre	age	date_init
0	LUSSAC	François	M	32	2019-08-02
1	MALSHERBES	Jeanne	M	48	2021-06-27
2	KATZ	Roger	M	62	2020-02-15
3	MULLER	Rémi	M	28	2022-04-12
4	REIBALDI	Charlotte	F	25	2021-09-04
5	LIENART	Patricia	F	18	2020-11-22
6	BAUCHY	Yann	M	32	2021-12-08
7	MUSSET	Roger	M	29	2022-03-17

1) La projection

La requête de **projection** permet de *demander uniquement certaines colonnes d'une table.* Le serveur renvoie les colonnes demandées qui sont affichées sous la forme d'un tableau du style EXCEL.

La projection selon les colonnes (`prenom`,`age`) de la table **Clients** affichera :

prenom	age
François	32
Jeanne	48
Roger	62
Rémi	28
Charlotte	25
Patricia	18
Yann	32
Roger	29

En langage SQL elle s'écrit :

Remarques :

Une requête spéciale permet de demander toutes les colonnes de la table. C'est un racourci permettant d'éviter de donner la liste de tous les attributs :

.

Mot clé `DISTINCT` :

On peut souhaiter éliminer les doublons d'une colonne, par exemple n'afficher que les prénoms distincts deux à deux. Cela est possible grâce au mot clé `DISTINCT` :

renvoie le résultat suivant :

prenom
François
Jeanne
Roger
Rémi
Charlotte
Patricia
Yann

2) La sélection

Une requête de **sélection** permet de *demander uniquement les lignes d'une table qui vérifient une* ***condition***.

Exemple 1 : dans la table **Clients** nous demandons uniquement les lignes dont l'age est inférieur à 30 ans. La requête se formule de la façon suivante :

ce qui affiche :

id	nom	prenom	genre	age	date_init
3	MULLER	Rémi	M	28	2022-04-12
4	REIBALDI	Charlotte	F	25	2021-09-04
5	LIENART	Patricia	F	18	2020-11-22
7	MUSSET	Roger	M	29	2022-03-17

Exemple 2 : dans la table **Clients** nous demandons uniquement les femmes dont l'age est inférieur à 30 ans. La requête se formule de la façon suivante :

ce qui donne :

id	nom	prenom	genre	age	date_init
4	REIBALDI	Charlotte	F	25	2021-09-04
5	LIENART	Patricia	F	18	2020-11-22

On écrit une requête de sélection en écrivant le mot-clé `WHERE` derrière le nom de la table, suivi d'une condition C qui est un **booléen** ; C ne peut donc prendre qu'une des deux valeurs vraie ou fausse.

Cette condition C peut être formulée avec :

1. Des opérateurs de comparaison :

Opérateur	**Signification**
=	Identique à
<	Strictement inférieur à
<=	Inférieur ou égal à
>	Strictement supérieur à
>=	Supérieur ou égal à
<> ou !=	Différent de
<=>	Identique à (valable pour `NULL` aussi)

2. Des opérateurs de combinaison d'expressions logiques :

Opérateur	**Autre syntaxe**	**Signification**
AND	&&	**et**
OR	||	**ou**
XOR		**ou exclusif**
NOT	!	**non**

3. Des opérateurs arithmétiques : cela ne concerne que les colonnes contenant des nombres (entiers ou flottants)

Opérateur	**Signification**
*	Multiplication
+	Addition
-	Soustraction
/	Division
DIV	Division entière
%	Reste d'une division entière
POW(attribut, e)	attribut exposant e (ex : POW(age,3))
SQRT(attribut)	racine carrée
ROUND(attribut)	arrondi à l'entier le plus proche

Les deux exemples qui suivent ne servent à rien mais ils sont là pour indiquer ce qu'on peut écrire.

Exemple 3 :

Exemple 4 :

Un exemple plus utile.

Exemple 5 : supposons que nous ayons une table **Marchandises** avec un attribut `nom : TEXT`, un attibut `prix_unitaire : FOAT` et un attribut `quantite : INTEGER`. On pourrait formuler une requête comme ci-dessous :

4. L'opérateur `IN`

   Si on veut construire une condition dans laquelle un élément est dans une liste de possibilités il est préférable d'utiliser l'opérateur `IN` au lieu d'empiler des `OR`

   Exemple 6 : plutôt que d'écrire :

```
SELECT * FROM Clients WHERE prenom = 'Jeanne' OR prenom = 'Rémi' OR prenom = 'Yann'
```

   il vaut mieux écrire :

   La négation de l'opérateur `IN` est `NOT IN`. On pourra donc formuler des requête de la forme suivante :

5. L'opérateur `BETWEEN`

   Exemple 7 : au lieu d'écrire :

```
SELECT * FROM Clients WHERE age >= 25 AND age <= 40
```

   on préférera :

   Remarque :

   Cela fonctionne avec tous les attributs dont les domaines sont munis d'un ordre total (c'est à dire pour nous avec quasiment tous les attributs !).

   Exemple 8 :

3) Combinaison des requêtes de projection et de sélection

Le langage SQL permet de réaliser une requête de projection et une requête de sélection en une seule fois. On remplace simplement l'opérateur * par la liste des attributs (colonnes) concernés par la projection.

Exemple 9 :

va afficher :

nom	prenom
MULLER	Rémi
REIBALDI	Charlotte
LIENART	Patricia
MUSSET	Roger

La syntaxe générale d'un requête combinant projection et sélection est donc :

```
SELECT attribut1, attribut2, ... FROM Nom_table WHERE condition_C
```

Remarques :

4) Trier les résultats selon un ordre croissant ou décroissant

Quand on formule la requête `SELECT nom, prenom FROM Clients` ou bien `SELECT age FROM Clients` le résultat affiché n'est en général pas l'ordre alphabétique des noms, ni des prénoms ou encore ce n'est pas l'age par valeurs croissantes.

Pour forcer un affichage trié selon un certain ordre il faut utiliser les mots-clés `ORDER BY` tout à la fin de la requête.

Exemple 10 :

va afficher une liste triée par noms croissants (c'est à dire dans l'ordre alphabétique).

va afficher la même liste mais triée cette fois-ci par ages croissants.

Il est tout à fait possible de demander un tri multi-critères (c'est à dire qui porte sur plusieurs colonnes) :

Exemple 11 :

va afficher une liste triée par :

- ages croissants ;
- puis s'il y a plusieurs personnes du même age, elles seront affichées par ordre alphabétique du nom.

Qui dit tri doit s'empresser de préciser : par ordre croissant ou décroissant ? Par défaut le tri est fait par ordre croissant [2] mais si on le veut en sens inverse il faut l'indiquer grâce à la clause `DESC`.

Ainsi lorsqu'on veut afficher tous les prénoms par ordre décroissant on écrira :

5) Limiter le nombre de lignes affichées

Il arrive que le nombre de lignes affichées par une requête soit tellement grand qu'on ait envie de restreindre l'affichage à une quantité limitée de lignes afin d'en faciliter la lecture. Ceci est possible grâce au mot-clé `LIMIT` qui doit être placé à la fin de la requête.

Exemple 12 :

ne va afficher que les dix premiers enregistrements (lignes) vérifiant la condition `age > 20`.

Dans le langage SQL, un second mot-clé est utilisé en complément de `LIMIT` : le mot `OFFSET`. Il permet de décaler les lignes affichées du nombre indiqué derrière `OFFSET` :

Exemple 13 :

permet d'afficher de la $6^{\text{ème}}$ ligne à la $15^{\text{ème}}$ ligne (donc 10 lignes en tout !)

Attention ! :

- Quand on spécifie `OFFSET` n, le résultat affiché commence à la $(n+1)^{\text{ème}}$ ligne.
- Lorsqu'on utilise `LIMIT`, la bonne pratique consiste à utiliser également la clause `ORDER BY` pour s'assurer que ce sont toujours les bonnes données qui sont présentées. En effet, si le système de tri est non spécifié, alors il est en principe inconnu et l'ordre d'affichage est totalement imprévisible !

Exemple 14 :

2. Il existe un mot-clé qui force le tri par ordre croissante, c'est **ASC**. Comme c'est le mode par défaut, il est inutile de l'écrire et nous n'en parlerons pas plus.

IV. Requêtes concernant plusieurs tables : union, intersection, différence, produit cartésien et jointure

Nous allons supposer dans cette partie que nous disposons de deux tables **Clients1** et **Clients2** correspondant aux clients de deux entreprises. Ces deux tables possèdent exactement les mêmes nombres de colonnes et les mêmes attributs, dans le même ordre.

Clients1
id
nom
prenom
genre
age
tel
date_inscription

Clients2
id
nom
prenom
genre
age
tel
date_inscription

Nous allons commencer par décrire les opérations ensemblistes *union*, *intersection* et *différence*.

1) Union

L'union des tables **Client1** et **Clients2** est la table dont les lignes appartiennent soit à **Clients1**, soit à **Clients2** (soit aux deux).

En langage SQL le mot-clé est `UNION` :

- `SELECT * FROM Clients1` renvoie toute la table Clients1. De même `SELECT * FROM Clients2` renvoie toute la table Clients2.
- Pour afficher la réunion des deux tables, on écrit :

On peut :

- utiliser un `WHERE` après chaque table pour limiter la réunion uniquement à certaines lignes ;
- utiliser une liste d'attributs à la place du `*` mais alors **celle-ci doit être la même pour les deux tables**.

Exercices

1. Écrire une requête qui affiche la réunion des lignes ne concernants que les clientes des deux entreprises.
2. Même question mais uniquement pour les clients masculins d'age < 30 ans.
3. Écrire une requête qui affiche uniquement la colonne **nom** de la table Clients1 $\cup$ Clients2.

2) Intersection

L'intersection des tables **Client1** et **Clients2** est la table dont les lignes appartiennent à la fois à **Clients1** et à **Clients2**.

En langage SQL le mot-clé est `INTERSECT` :

On peut :

- utiliser un `WHERE` après chaque table pour limiter l'intersection uniquement à certaines lignes ;
- utiliser une liste d'attributs à la place du `*` mais alors **celle-ci doit être la même pour les deux tables**.

Exercices :

1. Avec l'intersection, il faut toujours avoir en tête quelque chose ... Par exemple comment écrire correctement la requête qui affiche la table des clients des deux entreprises ?
2. Écrire une requête qui affiche les noms et prénoms des clients1 qui sont aussi présents dans Clients2 et dont l'age est supérieur à 35 ans.
3. Écrire une requête qui affiche les prénoms des clients1 qui sont aussi présents dans Clients2 et qui ne s'appellent pas Roger.

3) Différence

La différence des tables **Client1** et **Clients2** est la table dont les lignes appartiennent **Clients1** mais pas à **Clients2**.

En langage SQL le mot-clé est `EXCEPT` :

On peut comme avec les deux autre opérations ensemblistes :

- utiliser un `WHERE` après chaque table pour limiter la différence uniquement à certaines lignes ;
- utiliser une liste d'attributs à la place du `*` mais alors **celle-ci doit être la même pour les deux tables**.

Exemple :

Exercice : écrire la requête qui affiche les prénoms de **Clients1** qui ne sont pas dans **Clients2**.

4) Produit Cartésien

Introduction :

Soient **T1** et **T2** deux tables. Chaque ligne de **T1** est de la forme $(x_1, x_2, ..., x_n)$ et chaque ligne de **T2** est de la forme $(y_1, y_2, ..., y_p)$.

On appelle **produit cartésien** de ces deux tables la relation **T** dont les colonnes sont formées des colonnes de **T1** suivies de celles de **T2** et dont les lignes sont de la forme $(x_1, x_2, ..., x_n, y_1, y_2, ..., y_p)$.

Illustration et limitations de cette opération :

On dispose de deux tables **Coord_x** et **Coord_y** qui décrivent les coordonnées sur des axes Ox et Oy d'un ensemble d'atomes. On veut obtenir une seule table permettant d'avoir les coordonnées (x, y) de chaque atome :

Coord_x
id
x

Coord_y
id
y

Pour afficher toute la table produit cartésien **Coord_x** $\times$ **Coord_y**, on utilise la syntaxe :

Qu'affiche exactement cette requête ?

5) Jointure

a) Définition

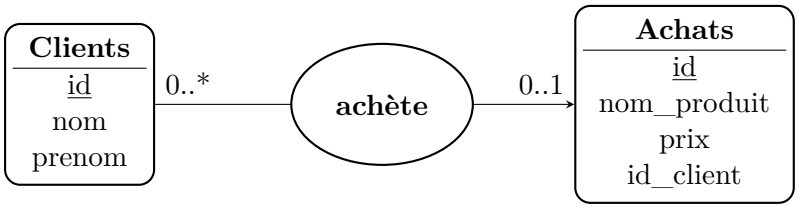
Exemples introductifs :

- On reprend l'exemple précédent des coordonnées des atomes. Comment afficher uniquement la liste des coordonnées (x, y) de chaque atome (et son identifiant) ?

Définition *Jointure*

On appelle **jointure** l'opération composée d'un produit cartésien suivi d'une sélection de lignes sur une condition d'égalité entre deux attributs.

- Une entreprise dispose dans sa BDR d'une table **Clients** qui recense la liste de ses clients et d'une table **Achats** qui donne la liste des achats effectués. On a le schéma E/A suivant en supposant que chaque entité de **Achat** ne peut être achetée que par 0 ou 1 client :



On souhaite connaître la liste des produits achetés par un client nommé Paul Bauchy. Comment faire ?

b) Syntaxe définitive d'une jointure avec JOIN ... ON

Pour réaliser une jointure il est fortement conseillé d'utiliser les deux mots-clés **JOIN** et **ON**, spécialement dédiés aux jointures. On écrit une jointure de la façon suivante :

On peut très bien n'afficher que certaines colonnes et certaines lignes de la table (appelée **jointure**) obtenue. On reprend le problème précédent : afficher la liste des produits achetés par Paul Bauchy :

Exercice :

Écrire une requête qui affiche les noms et prénoms des clients ayant acheté un appareil photo ; on supposera que dans les achats, le nom de ce produit est `'app_photo'`.

6) Renommage

Le **renommage** consiste **changer le nom** de certains *attributs* d'une relation R mais uniquement lors de l'affichage de la table avec une requête **SELECT** .. **FROM** ... **WHERE** : cela ne change pas le nom des attributs de la table qui est stockée dans la mémoire du serveur mais uniquement ce qui est affiché.

En pratique, le renommage peut devenir utile lorsqu'on réalise une jointure avec deux tables car la table obtenue peut contenir deux attributs de même nom. C'est le cas de l'attribut **id** dans l'exemple précédent **Clients** et **Achats** :

```
SELECT Clients.id, nom, Achats.id, prix FROM Clients JOIN Achats
                    ON Clients.id = Achats.id.id
```

va afficher :

id	nom	id	prix
1	LUSSAC	4	237
2	BAUCHY	9	42
...	...	...	...

Comment renommer les attributs de certaines colonnes ?

Remarque :

On peut toujours renommer un attribut avec **AS** (même en dehors d'une jointure) pour des raisons de commodité, goût personnel, etc ... Rappelons-le : *cela n'affecte que l'affichage.*

7) Jointure multiples

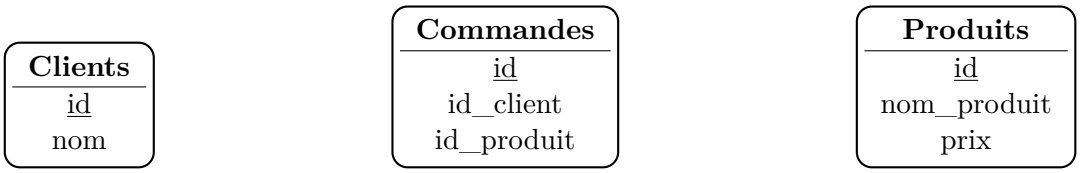
La syntaxe **JOIN** ... **ON** permet aussi de réaliser des *jointures multiples.*

Syntaxte générale :

On dispose de 3 tables **A**, **B** et **C** et on souhaite réaliser leur jointure : la syntaxe Mysql s'écrit en cascade. Pour plus de lisibilité, "chaque **JOIN** possède son **ON**" placé à sa verticale et immédiatement en dessous. Les indentations sont utilisées pour la clarté du programme (on a vu que le langage SQL n'est en réalité pas sensible aux indentations) :

```
SELECT ... FROM   A
                  JOIN   B
                  ON     A.attr1 = B.attr2
                         JOIN C
                         ON B.attr3 = C.attr4
```

Exemple : on dispose des trois tables :



1. Écrire une requête qui affiche les noms des clients ayant commandé le DVD dont le nom produit est `Guerre_des_Etoiles1`.

2. Écrire la requête qui affiche le nom et le prix de tous les produits commandés par les clients s'appelant Bauchy et Katz.

8) Sous-requête

Une requête de la forme **SELECT** ... **FROM** ... **WHERE** renvoie une table mais celle-ci n'existe qu'à l'affichage : *elle n'est pas stockée dans le disque dur du serveur* (celui-ci ne fait qu'envoyer les informations à l'ordinateur client connecté qui les affiche sous forme d'un tableau style EXCEL).

Cependant, dans certains cas, on voudrait pouvoir utiliser ce tableau affiché comme si c'était une vraie table stockée sur le serveur, par exemple pour faire une jointure ou bien n'importe quelle autre requête.

Le langage SQL permet de faire cela : on met la requête `SELECT ... FROM ... WHERE ...` entre deux parenthèses et *on lui donne un nom grâce à* **AS**. Cela donne :

```
(SELECT ... FROM ... WHERE ...) AS NomTable
```

Une fois cela fait, on peut utiliser `NomTable` comme s'il s'agissait d'une véritable table stockée dans la base de donnée. Par exemple, pour faire une jointure avec une table **A** on pourrait écrire :

```
SELECT * FROM  A
               JOIN (SELECT ... FROM B WHERE condition) AS MaTablePerso
               ON A.attr1 = MaTablePerso.attr2
```

La requête entre les deux parenthèses porte le nom de **sous-requête**.

V. Fonctions d'agrégation - Regroupement

1) Fonctions d'agrégation

Le langage SQL dispose d'un certain nombre de fonctions qui permettent faire des *opérations sur un ensemble de lignes dans une table* : ce sont les **fonctions d'agrégation**.

a) Définition

Soit **NomTable** une table possédant une colonne d'attribut **NomCol** et qui se présente donc de la façon suivante :

...	**NomCol**	...
...	e_1	...
...	e_2	...
...	...	...
...	e_n	...

La table possède n lignes et nous notons e_1, e_2, ... e_n les valeurs de ces lignes pour la colonne **NomCol**.

Définition *Fonction d'agrégation*

Une fonction d'agrégation **FA** est une fonction qui à un nom de colonne **NomCol** donné associe un résultat $f(e_1, e_2, ..., e_n)$ obtenu par une opération sur toutes les lignes $e_1, .e_2, ..., e_n$:

$$\mathbf{FA}: \mathbf{NomCol} \longmapsto \mathbf{FA}(\mathbf{NomCol}) = f(e_1, e_2, ..., e_n)$$

L'affichage du résultat se fait de la façon suivante :

```
SELECT FA(NomCol) FROM NomTable
```

b) Exemples au programme

Les fonctions d'agrégation au programme sont : `COUNT`, `AVG`, `MAX`, `MIN`, `SUM`. On travaille par exemple sur la table `Clients` suivante :

Table Clients

id	nom	prenom	age	genre	date_init
1	ARDITI	Lorène	22	F	NULL
2	CARUAL	Paul	25	M	2012-11-07
3	DOMBASLE	Etienne	37	M	2013-09-22
4	SCHERER	Marie	27	F	2014-05-13
5	TAOUSS	Cathy	29	F	2014-09-02
6	RODRIGUEZ	Cathy	19	F	NULL
7	BURGER	Vincent	22	M	NULL
8	RIOUL	Eric	34	M	2014-04-30
9,	KLOVAK	Alexandra	36	F	2013-06-02
10	KEROUAN	Sébastien	28	M	2013-12-24

1. La fonction `COUNT`

2. Les fonctions `MIN` et `MAX` :

   Tous les types de SQL possèdent une relation d'ordre total. Ces fonctions marchent aussi si la colonne est de type `TEXT` (l'ordre est alors défini l'ordre lexicographique) ou `DATE`.

   Exemples :

3. La fonction `SUM` :

4. La fonction `AVG` :

Remarque :

Toutes ces requêtes produisent une table formée d'une seule colonne et d'une seule ligne avec comme attribut le nom de la fonction d'agrégation utilisée.

b) Amélioration de l'affichage du résultat : renommage

On peut renommer le nom de la colonne affichée grâce au mot-clé `AS`.

2) Restriction des lignes sur lesquelles travaille la fonction d'agrégation

On peut faire en sorte que la fonction d'agrégation **FA** n'agisse que sur certaines lignes (et pas sur toutes) de la colonne **NomCol** : il faut ajouter un `WHERE` derrière le nom de la table qui permet de sélectionner certaines lignes selon *une condition C*.

```
SELECT FA(NomCol) AS ... FROM NomTable
WHERE condition C
```

Exemples :

- On veut l'age moyen uniquement des femmes de la table Client :

- On veut le nombre d'hommes dont l'âge est inférieur à 35 ans :

3) Regroupement : GROUP BY ...

On peut demander à une fonction d'agrégation de regrouper les lignes e_1, e_2, ..., e_n de la colonne **NomCol** en plusieurs sous-ensembles et de faire les calculs sur chacun de ces sous-ensembles. La fonction renverra alors *autant de résultats qu'il y a de sous-ensembles.*

Les sous-ensembles sont définis à partir des valeurs d'une autre colonne **AutreCol** : *tous les éléments d'un sous-ensemble donné possèdent la même valeur dans la colonne* **AutreCol**.

Exemple : dans la table `Clients` on a les deux colonnes

age	genre
32	M
48	F
62	M
28	M
25	F
18	F
38	M
29	M

On veut calculer les ages moyens des clients homme et femme séparément. Il faut donc regrouper les lignes de **age** selon les valeurs 'M' ou 'F' de la colonne **genre**. La syntaxe est alors :

La table affichée contient deux lignes qui représentent les ages moyens des femmes et des hommes. Comment savoir exactement à quelles valeurs 'M' ou 'F' ces deux lignes sont associées ?

La syntaxe générale avec une clause `WHERE` et un regroupement `GROUP BY` est :

```
SELECT AutreCol, FA(NomCol) AS ... FROM NomTable
WHERE Condition C
GROUP BY AutreCol
```

Remarques :

- `GROUP BY` vient toujours après le `WHERE` (s'il y en a un).
- À la place de `NomTable`, on peut mettre une jointure.

Une remarque importante :

On peut indiquer **plusieurs noms** de colonnes après GROUP BY. Dans ce cas ils doivent être séparés par des virgules. Expliquons le processus avec deux colonnes :

```
... GROUP BY colonne1, colonne2
```

Cela signifie que les lignes commencent par être regroupées par valeurs identiques de colonne1, puis, à *l'intérieur de chacun de ces goupes*, on regroupe les lignes par valeurs identiques de colonne2. L'ordre dans lequel sont écrites colonne1 et colonne2 est donc important.

Exemple :

Les données statistiques d'un pays donnent une table **Urbains** qui donnent la liste des habitants des grandes villes du pays avec des informations sur leur nom, age, genre.

Urbains
id
nom
age
genre
ville

On souhaite connaître l'age moyen des habitants de chaque grande ville du pays et, de plus, on souhaite le calculer séparément pour les hommes et pour les femmes.

On commence donc à regrouper les lignes par valeurs commune de `ville`, puis à l'intérieur de chaque ville par `genre`. Cela donne la requête :

```
SELECT AVG(age) FROM Urbains
GROUP BY ville,genre
```

Tout comme un `GROUP BY` avec une seule colonne, il est aussi possible d'afficher le résultat avec les deux colonnes qui ont servi à faire le regroupement. On pourra donc écrire :

```
SELECT ville, genre, AVG(age) FROM Urbains
GROUP BY ville,genre
```

ce qui donnera [3] :

ville	genre	AVG(age)
Lyon	F	35.5
Lyon	M	37.2
Strasbourg	F	33.8
Strasbourg	M	33.1
Paris	F	37.4
Paris	M	36.5

4) Condition sur l'affichage des résultats de la fonction d'agrégation

Terminons par un dernier mot-clé : `HAVING`.

La table que nous avons construite selon :

```
SELECT genre, AVG(age) AS age_moyen FROM Clients
GROUP BY genre
```

a donné pour résultat :

genre	age_moyen
F	26.6
M	29.2

On peut vouloir sélectionner uniquement certaines lignes comme par exemple celles telles que age_moyen < 28.

3. Les chiffres donnés n'ont aucune réalité. Ils servent juste à titre d'exemple.

Comme la condition porte sur la valeur d'une fonction d'agrégation il faut utiliser le mot-clé `HAVING` et jamais `WHERE` pour exprimer cette condition.

La syntaxe SQL correcte est donc :

```
SELECT genre, AVG(age) AS age_moyen FROM Clients
GROUP BY genre
HAVING age_moyen < 28
```

ce qui affiche :

genre	age_moyen
F	26.6

La syntaxe générale devient donc :

```
SELECT AutreCol1, AutreCol2, FA(NomCol) AS ... FROM NomTable
WHERE Condition C
GROUP BY AutreCol1, AutreCol2
HAVING FA(NomCol) opérateur valeur
```