Bases de données

MP2I Lycée Pierre de Fermat
guillaume.rousseau@ac-toulouse.fr

1 Base de données relationnelle

Un système de gestion de base de données (SGBD) est un logiciel permettant de stocker des données et d'y effectuer des requêtes, c'est à dire d'y chercher des informations spécifiques.

A Tables de données

Une table est une structure de données simple permettant de représenter un ensemble d'éléments ayant certains attributs variables.
Par exemple, on considère un club d'informatique, qui stocke les informations de ses membres dans une table : prénom, métier, âge.

Prénom	Métier	Age
Noémie	ingénieure réseau	36
Yacine	maître de conférence	33
Amos	développeur	19
Isabelle	data engineer	27
...	...	...

Dans cette table, chaque **ligne** correspond à une personne différente. On parle d'**enregistrements**.
Les **colonnes**, elles, correspondent aux différentes caractéristiques que l'on veut stocker pour chaque personne : on parle d'**attribut** ou de **champ**.
Dans la table précédente, il y a 3 attributs, et au moins 4 enregistrements.

Il existe différents formats de fichiers permettant de représenter une telle table. Le plus basique est le format CSV (Comma-Separated Values), qui consiste à écrire chaque enregistrement sur une ligne, en séparant chaque attribut d'une virgule. Pour la table précédente, le fichier CSV correspondant serait :

```
Prénom, Métier, Age
Noémie, ingénieure réseau, 36
Yacine, maître de conférence, 33
Amos, développeur, 19
Isabelle, data engineer, 27
...
```

Une telle table peut être vue mathématiquement comme une **relation** : si l'on note $A_1, \ldots, A_p$ les ensembles des différents attributs, alors une table est un sous-ensemble du produit cartésien $A_1 \times \ldots A_p$, autrement dit c'est une relation p-aire.
Par exemple, dans la table précédente, on a $p = 3$, et l'ensemble des valeurs possible pour le premier attribut est l'ensemble de tous les prénoms.
Un SGBD est dit **relationnel** lorsque la base de donnée prend la forme d'un ensemble de tables.

B Langage de requêtes

Supposons que l'on dispose d'un fichier CSV avec les mêmes attributs, mais avec des milliers d'enregistrements, et que l'on veut extraire l'information suivante : "Combien de personnes de moins de 30 ans le fichier contient-il ?". On doit alors écrire un programme, par exemple en C, qui lit le fichier, lit chaque ligne, et compte le nombre de fois où l'attribut "Age" contient un nombre inférieur à 30. Si l'on veut maintenant extraire l'information "Combien le fichier contient-il de personnes de plus de 30 ans dont le prénom commence par une voyelle ?", il faudra aller modifier notre programme, rajouter et modifier des conditions. A chaque nouvelle requête,

on devra donc écrire et compiler un programme spécifiquement fait pour.

Les SGBD sont généralement munis d'un langage de requêtes (ou Query Language), qui permet d'effectuer des requêtes plus ou moins complexes. Pour les SGBD relationnels, le langage le plus connu et le plus utilisé est **SQL** (Structured Query Language). Voici quelques exemples de requêtes que l'on peut effectuer en SQL :

— Renvoyer la liste des prénoms des membres de moins de 30 ans par ordre alphabétique :

```
SELECT Prénom FROM membres WHERE Age <= 30 ORDER BY Prénom
```

— Compter le nombre de membres de moins de 30 ans :

```
SELECT count(*) FROM membres WHERE Age <= 30
```

— Donner la somme des ages des membres s'appelant Camille :

```
SELECT sum(Age) FROM membres WHERE Prénom == "Camille"
```

— Donner pour chaque prénom l'age maximal d'un membre ayant ce prénom :

```
SELECT Prénom, max(Age) FROM membres GROUP BY Prénom
```

Dans ce chapitre, on s'intéresse à l'utilisation du langage SQL pour effectuer des requêtes sur des bases de données relationnelles, ainsi qu'à la modélisation de situations via les diagrammes Entité-Association, qui permettent de générer de manière systématique des bases de données.

2 Base de données, contraintes

Une base de donnée est un ensemble de tables, et chaque table est la donnée de noms de colonnes, appelés **attributs**, ainsi que de lignes comportant des valeurs pour chaque attribut, que l'on appelle des **enregistrements**.

Pour illustrer les différentes notions étudiées, on utilisera une base de données simple, qui stocke des informations sur des villes et des personnes (VOIR ANNEXE). Plus précisément, la base contient 5 tables. Pour désigner une table ayant pour colonnes `colonne_1`, ..., `colonne_k`, on écrit `nom_table(colonne_1, ..., colonne_k)` :

— `agglomeration(code, nom, nb_habitants, pays)` contient les informations basiques de villes. Le code est un identifiant unique par ville.

— `personne(id_pers, prenom, nom, naissance, pays)` contient les informations basiques de personnes. L'identifiant `id_pers` est unique par personne (on pourra imaginer que c'est le numéro de sécurité sociale par exemple).

— `concert(artiste, date, code_agglo, billets)` contient une liste de concerts, et pour chaque concert stocke l'artiste, la date, la ville et le nombre de billets vendus.

— `habite(idhabite, id_pers, code_agglo, debut, fin)` indique lorsqu'une personne d'identifiant `id_pers` a habité dans la ville de code `code_agglo` entre les dates `debut` et `fin`. L'attribut `idhabite` est un identifiant unique pour chaque enregistrement.

— `capitale(id_pays, code_agglo)` indique pour chaque pays sa capitale.

Par exemple, à la ligne d'identifiant 12 de la table `habite`, on peut y lire que la personne d'identifiant 3 a vécu dans la ville de code `ma` entre 2012 et 2020. En allant voir dans les deux autres tables, on voit que la personne en question est Dimitri Jujube, et que la ville en question est Madrid.

A Clé primaire

Dans les trois premières tables, un des attributs permet de déterminer de manière unique les enregistrements. Par exemple dans la table `personne`, l'attribut `id_pers` est unique pour chaque personne. On peut imaginer que c'est par exemple le numéro de sécurité sociale. Dans la troisième table, ça n'est pas le cas : deux concerts différents peuvent avoir la même date, ou bien le même lieu, ou bien être par le même groupe, ou bien avoir le même nombre d'entrées. En revanche, on peut supposer qu'il est impossible pour un groupe de jouer deux concerts le même jour. Alors, le **couple** d'attributs (`groupe`, `date`) détermine de manière unique un concert.

Définition 1. Dans une table $T(c_1, \dots, c_k)$, un sous-ensemble de colonnes $c_{i_1}, \dots c_{i_p}$ est une **clé primaire** s'il est impossible d'avoir deux enregistrements distincts $x = (x_1, \dots, x_k)$ et $y = (y_1, \dots y_k)$ dans T avec $x_{i_1} = y_{i_1}, \dots x_{i_p} = y_{i_p}$.

Lorsque l'on crée une table dans un SGBD, on peut spécifier pour chaque table si certains attributs forment une clé primaire. Alors, lorsque l'on insère une nouvelle ligne dans une table, on vérifie si la clé primaire est bien différente de toutes celles déjà présente. On choisit donc la clé primaire de manière à refléter des contraintes de la vie réelle, ce qui empêche la table de contenir des données brisant ces contraintes. On dit que les clés primaires sont un exemple de **contraintes d'intégrité**.
On se force à toujours manipuler des tables munies de clé primaires. De plus, par convention, on souligne les attributs faisant partie de la clé primaire d'une table. Par exemple, puisque l'on a dit que le couple (artiste, date) formait une clé primaire de la table `concert`, on pourra écrire : `concert(artiste, date, code_agglo, billets)`.

B Clé étrangère

Une **clé étrangère** est un ensemble d'attributs d'une table qui font référence à des attributs de la clé primaire d'une autre table. Par exemple, dans la table `habite`, l'attribut `code_agglo` fait référence à l'attribut `code` de la table `agglomeration`, qui y est une clé primaire.

Dans les SGBD relationnels, on peut spécifier les clés étrangères. Si l'on déclare qu'une table T_1 a une clé étrangère faisant référence à une table T_2, alors lorsque l'on ajoute dans T_1 un nouvel enregistrement, il faut que les valeurs associées à la clé étrangère figurent bien dans une des lignes de T_2.

Exemple 1. Si l'on essaie de rajouter dans `habite` une ligne où le code agglomération est `lo` (comme Londres), cela va causer une erreur, et l'opération sera annulée. En effet, dans la table `agglomeration`, aucune ligne ne correspond au code `lo`. Ainsi, on doit d'abord ajouter un enregistrement dans la table `agglomeration` pour représenter Londres, puis ensuite ajouter la ligne y faisant référence dans `habite`.

Les clés étrangères sont un exemple de **contraintes référentielles**.

3 SQL

A Bases

Dans le cadre de ce cours, une requête SQL commencera systématiquement par le mot-clé `SELECT`. La syntaxe de requête la plus basique est :

```
1 SELECT a1, a2, ... an FROM t;
```

où t est un nom de table et a_1, a_n sont des noms d'attributs de la table t. Cette requête a pour effet d'afficher pour chaque enregistrement de la table t une ligne, sur laquelle figure les valeurs de cet enregistrement pour les attributs $a_1, \ldots, a_n$.

Pour illustrer cette syntaxe, considérons la table `personne` précédente. Si l'on veut sélectionner le prénom et le nom de chaque personne :

Requête :

```
1 SELECT prenom, nom
2 FROM personne;
```

Résultat :

prenom	nom
Alice	Grelos
...	...
Claire	Banane
Claire	Banane

Remarquons que plusieurs lignes identiques apparaissent. En effet, plusieurs personnes s'appellent Claire Banane. Si l'on veut une liste sans doublon, on peut rajouter le mot-clé `DISTINCT` après `SELECT`. Par exemple, pour obtenir la liste des prénoms existants dans la table :

Requête :

```
1 SELECT DISTINCT prenom
2 FROM personne ;
```

Résultat :

prenom
Alice
...
François

Si l'on veut sélectionner tous les attributs, on peut écrire `*` au lieu de recopier les noms des attributs. Par exemple, pour afficher toutes les informations de la table `personne` :

Requête :

```
1 SELECT * FROM personne
```

Résultat :

id_personne	prenom	nom	naissance	pays
0	Alice	Grelos	1991	France
...	...	...	...	...
7	Claire	Banane	1953	Italie

B Conditions

On peut également conditionner les listes à renvoyer, en utilisant le mot-clé `WHERE`. Par exemple, si l'on veut afficher les informations des personnes nées après 1990 :

Requête :

```
1 SELECT *
2 FROM personne
3 WHERE naissance >= 1990;
```

Résultat :

id_pers	prenom	nom	naissance	pays
0	Alice	Grelos	1991	France
2	Claire	Igname	1995	Italie
5	François	Laitue	1992	Belgique

On peut combiner plusieurs conditions dans la clause `WHERE` d'une requête, en utilisant les opérateurs booléens `AND`, `OR` et `NOT`. Par exemple, pour trouver la liste des personnes nées après 1990, et nées ni France ni en Italie :

Requête :

```
1 SELECT *
2 FROM personne
3 WHERE naissance >= 1990
4 AND NOT (pays=France OR pays=ITALIE);
```

Résultat :

id_pers	prenom	nom	naissance	pays
5	François	Laitue	1992	Belgique

C Ordre, limites

On peut également ordonner les résultats d'une requête selon un certain attribut. Pour cela, on utilise le mot clé `ORDER BY`. On peut ensuite préciser si l'on veut un ordre croissant (`ASC`) ou décroissant (`DESC`). On peut trier selon un unique attribut ou selon un tuple d'attributs, ce qui triera les résultats selon l'ordre lexicographique sur les attributs précisés.

Par exemple, pour obtenir la liste des personnes par année de naissance décroissante puis par ordre alphabétique de nom en cas d'égalité :

Requête :

```
1 SELECT *
2 FROM personne
3 ORDER BY naissance DESC,
4        nom         ASC;
```

Résultat :

id_pers	prenom	nom	naissance	pays
2	Claire	Igname	1995	Italie
...	...	...	...	...
1	Bob	Hosta	1988	France
4	Elena	Kale	1988	Espagne
...	...	...	...	...

On peut également limiter le nombre de lignes renvoyées par la requête. Pour cela, on utilise le mot clé `LIMIT`. On peut également sauter les premières lignes en utilisant le mot clé `OFFSET`.

Par exemple, pour afficher les 5 personnes les plus vieilles, en sautant les 2 plus vieilles :

Requête :

```
1 SELECT *
2 FROM personne
3 ORDER BY naissance ASC
4 LIMIT 5 OFFSET 2;
```

Résultat :

id_pers	prenom	nom	naissance	pays
3	Dimitri	Jujube	1982	Italie
1	Bob	Hosta	1988	France
4	Elena	Kale	1988	Espagne
0	Alice	Grelos	1991	France
5	François	Laitue	1992	Belgique

D Opérations ensemblistes

Le résultat d'une requête est une table, c'est à dire un ensemble de lignes. Étant donné plusieurs requêtes, on peut effectuer des opérations ensemblistes usuelles sur leurs résultats : l'union avec `UNION`, l'intersection avec `INTERSECT` et la différence avec `EXCEPT`. On peut faire l'union, l'intersection et la différence de résultats de requêtes effectuées sur des tables différentes, du moment que les résultats sont de même dimensions et que les attributs ont le même type.

Pour illustrer cela, faisons une requête utilisant la table personne ainsi que la table agglomération. Essayons d'obtenir la liste des pays pour lesquels au moins une ville est enregistrée mais où aucune personne de la base n'est née. Il faut donc faire la différence entre la liste des pays apparaissant dans la table `agglomeration` et la liste des pays apparaissant dans la table `personne`. On peut donc écrire :

Requête :

```
1 SELECT pays
2 FROM agglomeration
3 EXCEPT SELECT pays
4 FROM personne;
```

Résultat :

pays
Angleterre

Remarque 1. En SQL, on peut **renommer les colonnes** en utilisant le mot-clé AS :

```
1 SELECT pays AS pays_ville
2 FROM agglomeration
3 EXCEPT SELECT pays AS pays_personne
4 FROM personne;
```

On remarque d'ailleurs que lors d'une opération ensembliste, les colonnes du résultat auront les même noms que les colonnes de la première requête.

Essayons maintenant d'obtenir la liste des pays où au moins une personne est née avant 1970 et où il y a une ville ayant entre 100 000 et 200 000 habitants :

Requête :

```
1 SELECT pays
2 FROM personne
3 WHERE naissance > 1990
4 INTERSECT SELECT pays
5 FROM agglomeration
6 WHERE 100000 <= nb_habitants
7 AND nb_habitants <= 200000;
```

Résultat :

pays
Belgique
France

De manière logique, le "et" de l'énoncé en langage naturel s'est traduit en une intersection.

E Produit cartésien

SQL permet d'effectuer des requêtes sur plusieurs tables à la fois, ce qui revient à faire une requête sur le produit cartésien de tables. Par exemple, regardons la requête suivante :

Requête :

```
1 SELECT prenom, nb_habitants
2 FROM personne, agglomeration
3 ORDER BY prenom ASC,
4          nb_habitants DESC;
```

Résultat :

prenom	nb_habitant
Alice	3200000
Alice	2800000
...	...
Alice	13000
Bob	3200000
Bob	2800000
...	...
François	13000

Cette requête génère donc la liste de tous les couples (prenom, nombre d'habitant) pour chaque couple (personne, agglomeration). Le résultat n'a pas d'interprétation particulière car il relie chaque personne avec chaque ville.

En SQL, on peut renommer non seulement les attributs mais aussi les noms de tables au sein d'une requête. Cela permet par la suite de spécifier de quelle table on veut prendre les attributs. Par exemple, les tables `personne` et `agglomeration` ont toutes les deux un attribut `nom`, mais qui n'a pas du tout le même sens ! Affichons la liste des couples (nom de personne, nom de ville) pour lesquels la ville est dans le pays où la personne est née :

Requête :

```
1 SELECT p.nom AS nom_personne,
2      v.nom AS nom_ville
3 FROM personne AS p,
4    agglomeration AS v
5 WHERE v.pays = p.pays ;
```

Résultat :

nom_personne	nom_ville
Grelos	GRENOBLE
...	...
Grelos	TOULOUSE
...	...
Laitue	BRUGES
Laitue	BRUXELLES
Banane	FORLIMPOPOLI
...	...
Banane	ROME

On a donc renommé les tables avec AS, ce qui permet de distinguer les attributs `nom` des deux tables, en utilisant la notation `table.attribut`.

Tentons une requête plus complexe, en utilisant la table `habite`, où sont stockées les différents lieux où chaque personne a habité. Pour chaque enregistrement de cette table, on

stocke l'identifiant d'une personne, le code de la ville où elle a habité, et les années de début et de fin de son séjour. Notons que l'on ne stocke pas le prénom ou le nom des personnes, ni le nom des villes. De manière générale, dans les bases SQL, il est de bonne pratique d'utiliser un identifiant unique servant de clé primaire au sein de chaque table, et d'utiliser cet identifiant comme clé étrangère si l'on veut faire référence à cette table.
Si l'on veut exploiter cette table d'une manière plus lisible pour les humains, il faut utiliser le produit cartésien pour récupérer le nom des personnes et le nom des villes. Par exemple, récupérons la liste des personnes ayant habité en France en 2000 :

Requête :

```
1  SELECT p.prenom,
2         p.nom as nom_personne,
3       h.debut,  h.fin,
4       v.nom AS nom_ville
5  FROM personne AS p,
6       habite AS h,
7       agglomeration AS v
8  WHERE h.id_pers = p.id_pers
9  AND   h.code_agglo = v.code
10 AND   v.pays = 'France'
11 AND   h.debut <= 2000
12 AND 2000 <= h.fin;
```

Résultat :

prenom	nom_personne	debut	fin	nom_ville
Alice	Grelos	1991	2012	RENNES

F Jointure

Dans l'exemple précédent, nous avons pris le produit cartésien de trois tables, puis avons filtré les résultats selon certaines conditions. Comme chaque table fait moins de 20 lignes, le produit cartésien en fait au plus 8000, ce qui est gérable. Mais pour des tables de plusieurs milliers de lignes, faire de tels produits cartésiens devient vite ingérable pour le logiciel. Par ailleurs, les premières conditions du `WHERE` sont très particulières : elles identifient plusieurs colonnes (par exemple, la condition `h.code_agglo = v.code`).

En SQL, plutôt que d'utiliser des produits cartésiens, on utilise des **jointures**, avec les mots clés `JOIN` et `ON`, qui permettent de fusionner des tables selon des conditions.

La syntaxe est la suivante :

```
SELECT ...
FROM table1 AS nom1
JOIN table2 AS nom2 ON conditionA
JOIN table3 AS nom3 ON conditionB
...
WHERE ...
```

Une telle requête produit le même résultat que :

```
SELECT ...
FROM table1 AS nom1, table2 AS nom2, table3 AS nom3
...
WHERE conditionA AND conditionB AND ...
```

Par exemple, si l'on réécrit la requête précédente avec des jointures :

```
SELECT p.prenom, p.nom as nom_personne,
     h.debut, h.fin,
     v.nom AS nom_ville
FROM personne AS p
JOIN habite AS h ON h.id_pers = p.id_pers
JOIN agglomeration AS v ON h.code_agglo = v.code
WHERE v.pays = 'France'
AND h.debut <= 2000
AND 2000 <= h.fin;
```

Notons que le résultat sera identique. De manière générale, une requête utilisant `JOIN` peut se réécrire en utilisant uniquement des produits cartésiens, pour un résultat identique. Cependant, en interne, utiliser le mot-clé JOIN peut permettre au logiciel d'accélérer la requête, selon l'implémentation de SQL utilisée.

Voyons quelques autres exemples . Pour commencer, affichons la liste des villes dans lesquelles une personne de la base a habité avant 1995. Afin de ne pas avoir de doublons, on peut utiliser le mot clé `DISTINCT` :

Requête :

```
SELECT DISTINCT v.nom
FROM habite AS h
JOIN agglomeration AS v ON h.code_agglo = v.code
WHERE h.debut <= 1995;
```

Résultat :

nom
RENNES
PARIS
MILAN
ROME
MADRID
BRUGES

Exercice 1.

Question 1. Compléter la requête suivante pour qu'elle renvoie le nom de la 1ère ville dans laquelle Claire Igname a habité :

```
1 SELECT v.nom
2 FROM
3 JOIN
4 JOIN
5 WHERE
6
7 ORDER BY
8 LIMIT 1
```

Question 2. Compléter la requête suivante pour qu'elle renvoie tous les couples de personnes ayant vécu à un moment dans la même ville.

```
1 SELECT
2 FROM personne p1
3 JOIN habite h1 ON
4 JOIN personne p2 ON
5 JOIN habite h2 ON
6 WHERE
7 AND
```

G Jointure externe

Utilisons la table `capitale(id_pays, code_agglo)`. Si l'on veut afficher, pour chaque ville, le pays dont elle est capitale, on peut utiliser la requête suivante :

Requête :

```
1 SELECT a.nom, c.id_pays
2 FROM agglomeration a
3 JOIN capitale c ON c.code_agglo = a.code;
```

Résultat :

nom	id_pays
PARIS	France
ROME	Italie
MADRID	Espagne
BRUXELLES	Belgique

Remarquons que les villes qui ne sont capitales d'aucun pays n'apparaissent pas. En effet, le mot clé `JOIN` effectue une **jointure interne**, dans laquelle une ligne n'apparaît que si elle apparaît dans les deux tables jointes. Comme aucune ligne de `capitale` ne fait apparaître Toulouse, elle n'apparaît pas dans la jointure.

Par opposition, le mot clé `LEFT JOIN` permet de faire une **jointure externe gauche**, dans laquelle toutes les lignes de la table de gauche seront gardées, même si elles ne correspondent à aucune ligne de la table de droite. Dans ce cas, les valeurs de la ligne dans la jointure relatifs aux champs de la table de droite seront vides, ce qui en SQL est représenté par la valeur `NULL`. Par exemple, en reprenant la requête précédente, si l'on utilise une jointure externe :

Requête :

```
1 SELECT a.nom, c.id_pays
2 FROM agglomeration a
3 LEFT JOIN capitale c ON c.code_agglo = a.code;
```

Résultat :

nom	id_pays
PARIS	France
...	...
BRUXELLES	Belgique
TOULOUSE	NULL

Toutes les lignes de `agglomeration` apparaissent, même celles qui ne correspondent pas à des capitales.

Dans une table, lorsqu'un attribut est `NULL`, on ne peut pas le comparer avec $<, \leq, \ldots$. On ne peut même pas utiliser $=$, car NULL n'est pas égal à NULL en SQL : toute comparaison faisant intervenir NULL renvoie faux. Il faut utiliser `x IS NULL` pour savoir si l'attribut x vaut `NULL`. Par exemple, pour obtenir la liste des villes qui ne sont pas des capitales, on peut utiliser `EXCEPT`, ou bien reprendre la requête précédente et utiliser `IS NULL` pour ne garder que les lignes n'étant pas des capitales :

```
1 SELECT nom FROM agglomeration
2 EXCEPT SELECT a.nom
3 FROM agglomeration a
4 JOIN capitale c ON c.code_agglo=a.code
```

```
1 SELECT a.nom
2 FROM agglomeration a
3 LEFT JOIN capitale c
4      ON c.code_agglo = a.code
5 WHERE c.id_pays IS NULL;
```

Voyons une autre manière d'utiliser `NULL`. Si l'on veut obtenir la plus grande ville de la base, on peut utiliser un `ORDER BY` et un `LIMIT 1`.
Une alternative est de se dire que la plus grande ville est la seule ville pour laquelle aucune ville n'est strictement plus grande. On peut donc faire un `LEFT JOIN` entre deux exemplaires a_1 et a_2 de la table `agglomeration`, avec comme condition de jointure $a_1.nb\_habitants < a_2.nb\_habitants$. Cela nous donnera une liste de couples (`ville_1`, `ville_2`) où `ville_2` a strictement plus d'habitants que `ville_1`. Or, pour v_0 la ville avec le plus d'habitants, il n'existe aucune autre ville w telle que (v_0, w) peut apparaître dans la jointure. Ainsi, v_0 sera la seule ville qui fera apparaître des `NULL` :

```
SELECT a1.nom
FROM      agglomeration a1
LEFT JOIN agglomeration a2
   ON a1.nb_habitants < a2.nb_habitants
WHERE a2.nom IS NULL;
```

On retrouve alors que la ville la plus peuplée est Madrid.

H Agrégation

Les fonctions d'agrégation permettent d'effectuer un calcul sur tout un ensemble de valeurs : la somme, le max, la moyenne... En SQL, il y en a plusieurs implémentées par défaut : `max`, `min`, `count`, `sum`, `avg` (comme average, i.e. la moyenne).

Par exemple, si l'on veut compter le nombre de personnes nées en Italie :

```
SELECT count(*)
FROM personne
WHERE pays='Italie';
```

Ce qui donne 4. `count(*)` signifie que l'on compte chaque ligne du résultat. Si l'on veut obtenir le nombre de prénoms différents de personnes nées en Italie, on peut spécifier que l'on compte uniquement les prénoms distincts :

```
SELECT count(DISTINCT prenom)
FROM personne
WHERE pays='Italie';
```

On obtient alors 2. Pouvoir spécifier l'attribut sur lequel appliquer la fonction d'agrégation permet donc aussi de faire des sommes, des moyennes, etc... Par exemple, pour obtenir la population totale des villes italiennes de la base :

```
SELECT sum(nb_habitants)
FROM agglomeration
WHERE pays ='Italie';
```

On obtient 4163000.

Exercice 2.

Compléter le code suivant et obtenir la première année pour laquelle on a des données sur l'adresse de Dimitri Jujube :

```
SELECT
FROM personne p
JOIN habite h ON h.id_pers = p.id_pers
WHERE
```

I Regroupements

Le mot-clé `GROUP BY` est un des outils les plus puissants que l'on verra cette année en SQL. Il permet de regrouper les lignes d'une requête selon les valeurs d'un ou plusieurs attributs. Une fois les groupes formés, les opérations d'agrégation se font sur chaque groupe plutôt que sur l'ensemble des lignes. Par exemple, la requête suivante donne pour chaque pays le nombre d'habitants total des villes enregistrées pour ce pays :

```
1 SELECT pays, sum(nb_habitants)
2 FROM agglomeration
3 GROUP BY pays;
```

pays	sum(nb_habitant)
Angleterre	150000
Belgique	1368000
Espagne	3963000
France	2981000
Italie	4163000

On peut renommer les colonnes créées, ce qui permet alors de les utiliser pour ordonner les résultats :

```
1 SELECT pays, sum(nb_habitants) AS pop
2 FROM agglomeration
3 GROUP BY pays
4 ORDER BY pop DESC
```

pays	pop
Italie	4163000
Espagne	3963000
France	2981000
Belgique	1368000
Angleterre	150000

Si l'on veut obtenir le nombre de personnes de la base ayant habité dans chaque ville, en ne gardant que les 5 villes les plus populaires :

```
1 SELECT v.nom, count(DISTINCT h.id_pers) as pop
2 FROM agglomeration v
3 JOIN habite h ON h.code_agglo = v.code
4 GROUP BY v.nom
5 ORDER BY pop DESC
6 LIMIT 5
```

nom	pop
ROME	4
MADRID	3
SEVILLE	2
PARIS	2
OXFORD	2

On peut aussi utiliser les colonnes agrégées dans les conditions, mais on utilise alors le mot clé `HAVING` plutôt que `WHERE`. Par exemple, pour obtenir la liste des pays ayant exactement une ville où quelqu'un appelé Claire a habité :

```
1 SELECT a.pays, count(DISTINCT a.code) as nombre
2 FROM agglomeration a
3 JOIN habite h ON h.code_agglo = a.code
4 JOIN personne p ON p.id_pers = h.id_pers
5 WHERE p.prenom = "Claire"
6 GROUP BY a.pays
7 HAVING nombre = 1;
```

Cette requête est exécutée dans l'ordre suivant :

1. D'abord, la jointure crée une grande table contenant une ligne par couple (ville, personne) tel que la personne a habité dans la ville.
2. Ensuite, le `WHERE` ne garde que les lignes concernant les personnes s'appelant Claire.
3. Puis, le `GROUP BY` regroupe les lignes restantes par pays de la ville, et le `count(DISTINCT a.code)` compte le nombre distinct de ville au sein de chaque groupe.
4. Enfin, le `HAVING` filtre pour ne garder que les pays qui ont un compte égal à 1.
5. En tout dernier, seules les colonnes correspondant au pays et au compte sont gardées.

Il faut donc faire très attention lorsque l'on mélange `WHERE` et `HAVING` :

— Les conditions du `WHERE` s'appliquent **avant** les groupements, et ne peuvent donc pas faire référence à des attributs dépendant des groupes. En particulier, interdit d'y utiliser les colonnes créées à partir de fonctions d'agrégation.

— Les conditions du `HAVING` s'appliquent **après** les groupements, et ne peuvent donc pas faire référence à des attributs mal définis au sein de chaque groupe. On ne peut donc y utiliser que les attributs utilisés pour le regroupement et les attributs créés à partir de fonctions d'agrégation.

Par exemple, la requête suivante :

```
SELECT pays, count(*)
FROM agglomeration
GROUP BY pays
HAVING nb_habitants > 100000;
```

n'est pas valide, car la condition du `HAVING` fait référence à `nb_habitants`, mais les groupes sont faits selon l'attribut `pays`, et donc un groupe donné n'a pas un attribut `nb_habitants` bien défini.
En revanche, si l'on utilise un `WHERE` à la place, la requête renvoie pour chaque pays le nombre de villes de plus de 100000 habitants qu'il contient :

```
SELECT pays, count(*)
FROM agglomeration
GROUP BY pays
WHERE nb_habitants > 100000;
```

Voyons un exemple d'utilisation valide de `HAVING`. On veut récupérer la liste des pays ayant plus de 3000000 habitants au total :

```
SELECT pays,
     sum(nb_habitants) AS population
FROM agglomeration
GROUP BY pays
HAVING population > 3000000;
```

On peut mélanger `WHERE` et `HAVING`. Par exemple, pour obtenir la liste des pays ayant exactement une ville où quelqu'un appelé Claire a habité :

```
SELECT p.pays, count(DISTINCT h.code_agglo) as nombre
FROM agglomeration a
JOIN habite h ON h.code_agglo = a.code
JOIN personne p ON p.id_pers = h.id_pers
WHERE p.prenom = "Claire"
GROUP BY p.pays
HAVING nombre = 1;
```

D'abord, le `WHERE` sert à ne prendre que les lignes concernant les personnes s'appelant Claire. Puis, le `GROUP BY` sert à regrouper les lignes par pays , et le compte du nombre distinct de ville est fait au sein de chaque groupe. Enfin, le `HAVING` filtre pour ne garder que les pays qui ont un compte égal à 1.

4 Modèle Entité-Association

Le modèle Entité Association, ou modèle EA, est une méthode de modélisation et de conceptualisation. Il consiste à modéliser une situation en la représentant par différentes entités ayant des attributs, et reliées par des associations. Le modèle EA est utile car il permet de simplifier la conception de bases de données et d'en rendre certaines parties plus systématiques.

A Entités

Une **entité** est une chose, un objet, un concept. Par exemple, dans un lycée, chaque élève, chaque professeur, chaque cours, chaque classe, est une entité. Une entité possède des attributs qui permettent de la décrire. Par exemple :

— Un élève peut avoir un prénom, un nom, un âge, un numéro INE, etc...

— Un cours peut avoir un horaire, une matière associée, etc...

— Une salle peut avoir un bâtiment, un numéro, une capacité max d'élèves, etc...

Chaque entité a une valeur associée à chacun de ses attributs. Certains attributs d'une entité peuvent la caractériser entièrement. Par exemple, deux élèves distincts ne peuvent pas avoir le même numéro INE, et deux salles ne peuvent pas avoir le même numéro et être dans le même batiment.

On appelle **clé primaire** d'une entité un sous-ensemble de ses attributs qui caractérisent de manière unique l'entité. Dans l'exemple précédent, une clé primaire des entités "élèves" est le numéro INE, et une clé primaire des entités "salles" est le couple (numéro, bâtiment).
La clé primaire d'une entité reflète des contraintes de la situation concrète que l'on modélise.

Voici un exemple représentant schématiquement une situation avec 4 entités :

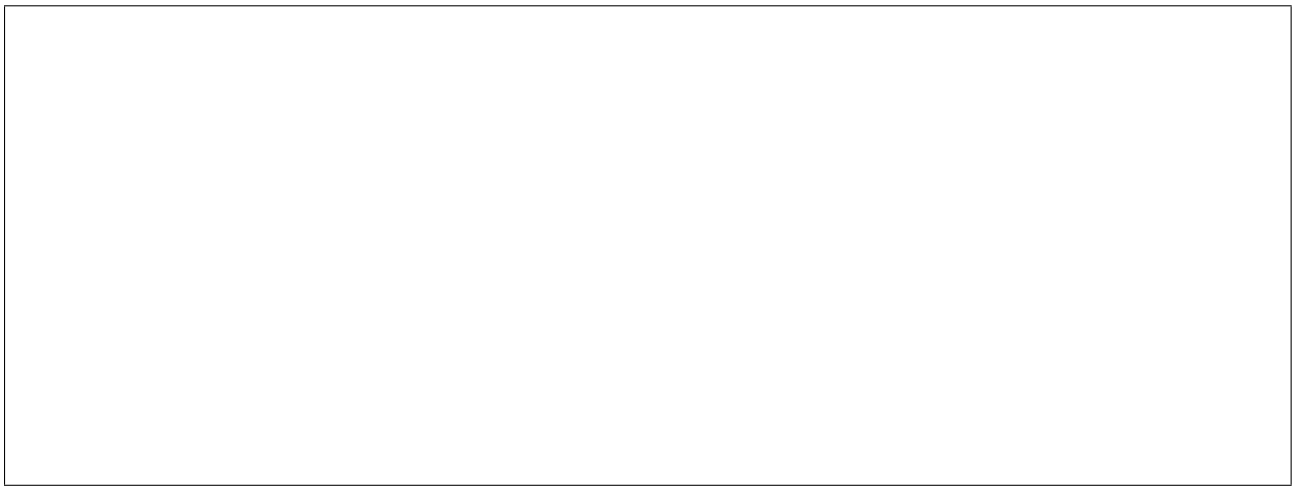
— `eleve(prenom, nom, INE)`

— `professeur(id, prenom, nom)`

— `cours(id, intitule, horaire, matiere)`

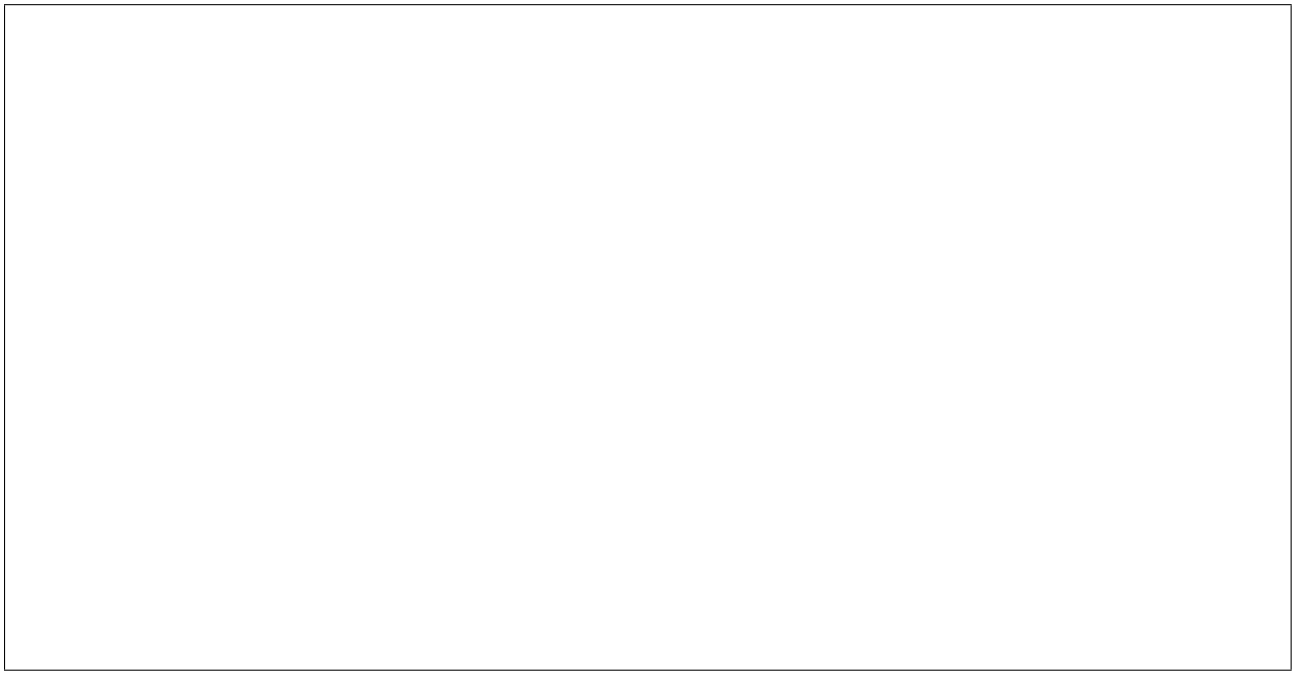
— `salle(numero, batiment, effectif)`

Chaque entité a plusieurs attributs, et les attributs faisant partie de la clé primaire de leur entité sont soulignés. On représentera les entités par des rectangles :

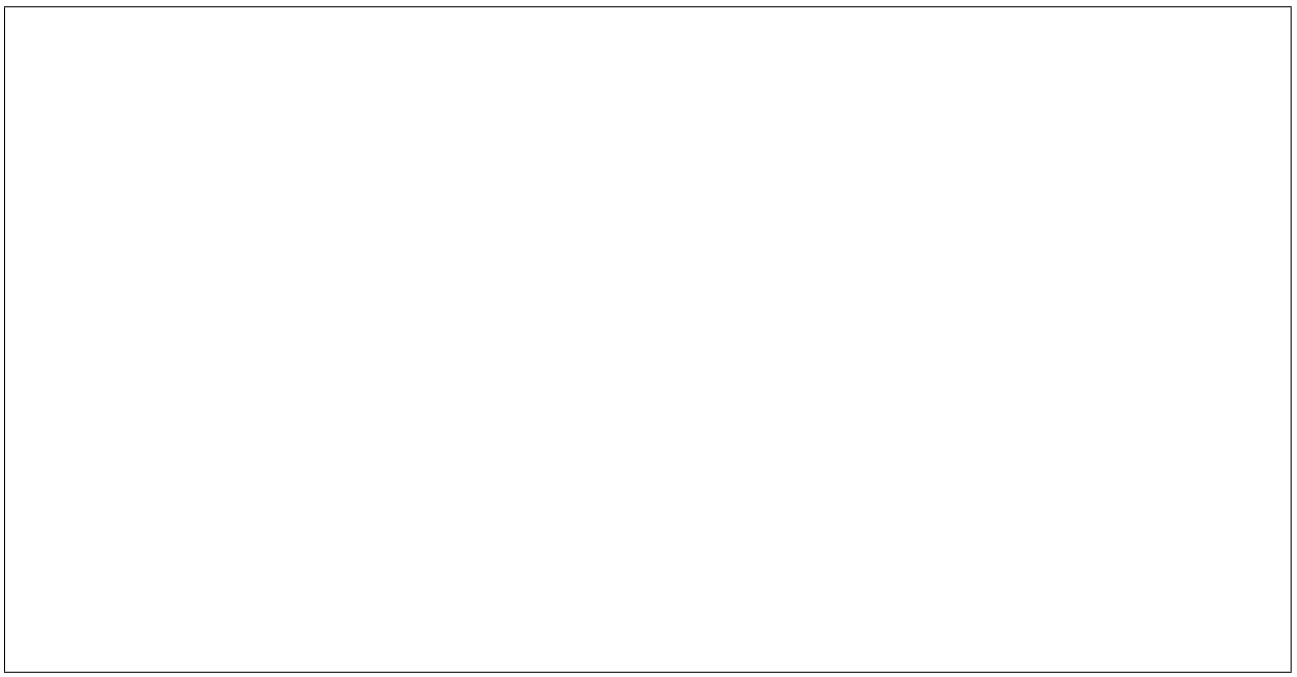


B Associations

Une **association** est un lien entre deux entités. Par exemple, on peut considérer l'association "est inscrit à" liant les élèves aux cours, et l'association "se situe en" liant les cours aux salles. On représente les associations par des losanges posés sur des flèches reliant les deux entités concernées. Mettons à jour le diagramme précédent en faisant figurer ces associations :



Rien n'empêche une association de relier une entité à elle-même. Par exemple, on peut considérer l'association "est tutoré par" qui relie les professeurs aux professeurs. De plus, les associations peuvent elles-aussi être munies d'attributs. Par exemple, l'association "est inscrit à" entre les élèves et les cours peut avoir un attribut "note" correspondant à la note finale d'un élève à un cours où il est inscrit. Mettons à jour notre diagramme, et rajoutons également l'association "enseigne" qui relie les professeurs aux cours :



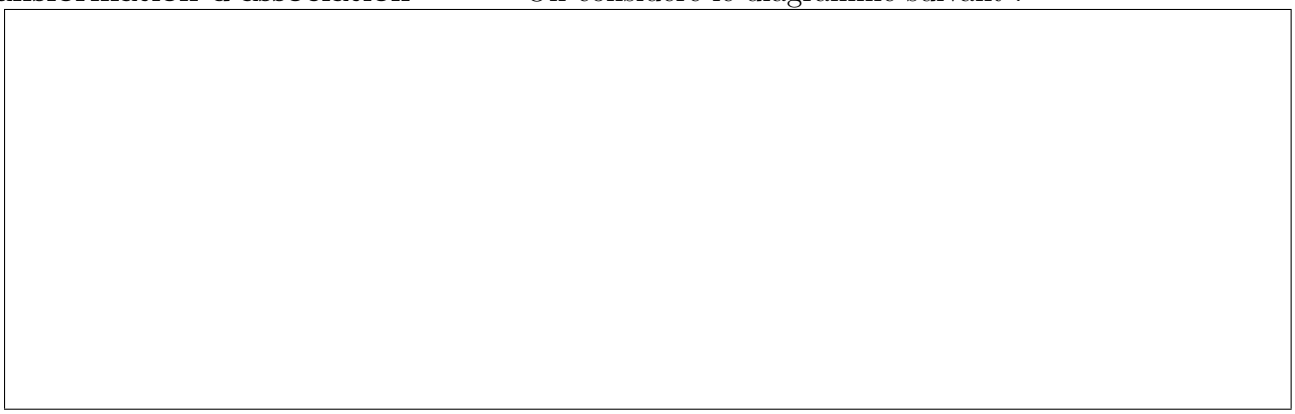
C Cardinalité

On peut classifier les associations selon le nombre d'entités qui sont concernés aux deux bouts de la relation. Par exemple, l'association "est inscrit à" reliant Élève à Cours peut lier un élève à plusieurs cours différents, et un cours à plusieurs élèves différents. En revanche, l'association "se situe en" reliant Cours à Salle ne peut relier un cours qu'à une seule salle : un cours ne peut pas avoir lieu dans plusieurs salles à la fois. En revanche, une salle peut accueillir plusieurs cours.

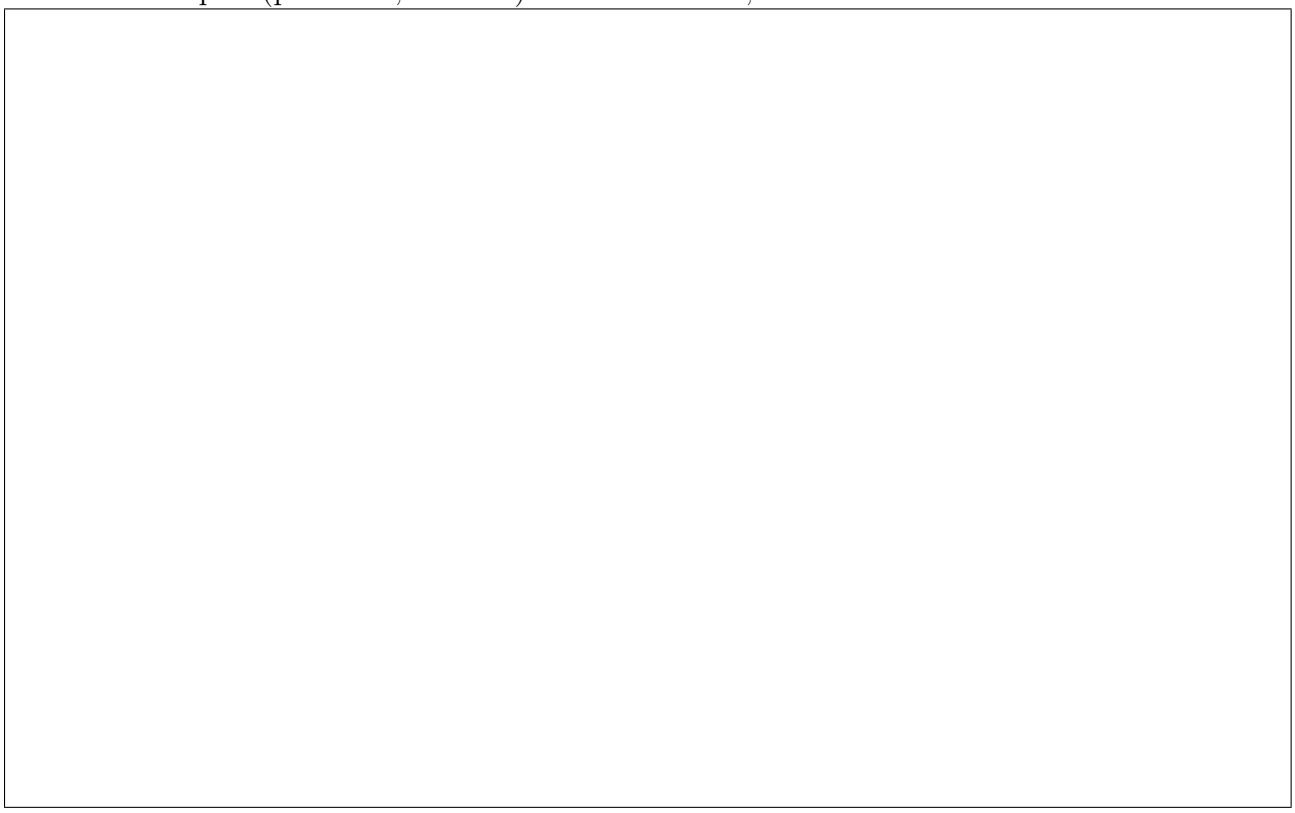
La **cardinalité** d'une association par rapport à une entité liée par cette association est le nombre maximal de liaisons distinctes qu'un représentant de l'entité peut avoir via l'association. Une association liant deux entités aura donc deux cardinalités, que l'on représente sur le diagramme EA aux deux bouts de la flèche. On distinguera dans le cours deux cardinalités possibles : 1 et ∞. On notera $*$ pour une cardinalité infinie.

Exemple 2. Mettons à jour le diagramme précédent avec les cardinalités des différentes associations.

Transformation d'association $* - *$ On considère le diagramme suivant :

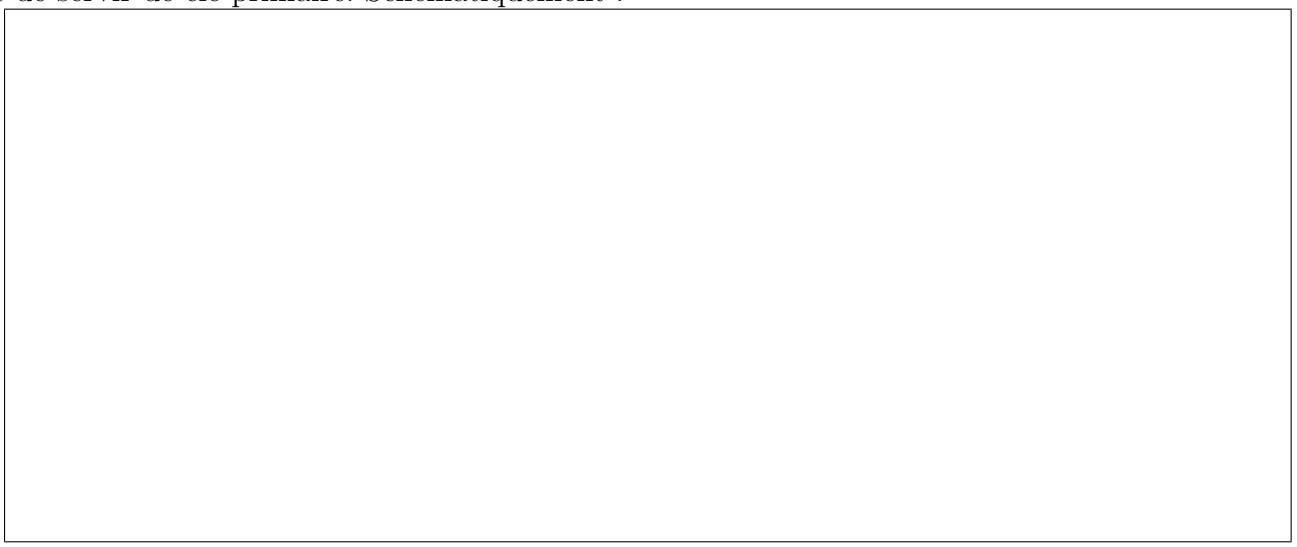


Ce diagramme représente des personnes et les aliments qu'elles aiment. L'association "aime bien" est de type $* - *$ car une personne peut aimer plusieurs aliments, et un aliment peut être aimé par plusieurs personnes. On peut alors remplacer cette association par une entité qui représente les couples (personne, aliment) de l'association, comme suit :



Remarquons que dans cette nouvelle entité, le couple (id_personne, id_aliment) est une clé primaire.

Remplaçons maintenant l'association par "a mangé", et rajoutons un attribut donnant la date à laquelle une personne a mangé un aliment. On ne peut pas appliquer le même procédé qu'avant, car le couple (id_personne, id_aliment) n'est plus une clé primaire : une personne peut avoir mangé un aliment à deux moments différents.
Dans ce cas de figure, nous allons simplement ajouter un nouvel attribut qui aura pour seul but de servir de clé primaire. Schématiquement :



Exemple 3. Appliquons cette méthode à la relation "est inscrit à" de notre diagramme.

D Du diagramme aux tables

La modélisation d'une situation par un diagramme EA permet ensuite de créer de manière assez systématique un schéma relationnel, c'est à dire un schéma de BDD relationnelle.

Entités Chaque entité du diagramme donne lieu à une table, dont les attributs sont ceux de l'entité, et dont la clé primaire est celle de l'entité.

Pour notre diagramme, on aura donc 4 tables pour les entités :

— `eleve(INE, prenom, nom)`

— `professeur(id, prenom, nom)`

— `cours(id, intitule, horaire, matiere)`

— `salle(numero, batiment, effectif)`

Associations On considère A une association reliant deux entités E_1, E_2. On la représente par une table dont les attributs sont la réunion de :

— les attributs de la clé primaire de E_1

— les attributs de la clé primaire de E_2

— les attributs additionnels éventuels de A

Par exemple, l'association `est inscrit a` de notre exemple filé deviendrait `inscription(INE, id_cours, note)`

Cependant, selon les cardinalités de l'association, on peut simplifier la situation. Par exemple, si l'association est de type $1 - *$, cela signifie que chaque entité de E_1 a une seule entité de E_2 qui lui est liée. C'est en fait une relation fonctionnelle. On peut alors rajouter aux attributs de E_1 une copie de la clé primaire de E_2, qui formera alors une clé étrangère faisant référence à E_2.

Par exemple, l'association `se situe en` peut se modéliser en rajoutant à la table `cours` deux attributs `batiment` et `numero_salle`, correspondant ainsi à la clé primaire de la table `salle`. Il en va de même pour l'association `enseigne` entre les professeurs et les cours : un cours n'est enseigné que par un unique professeur.

La table pour les cours deviendra donc :
`cours(id, intitule, horaire, matiere, id_professeur, numero_salle, batiment)`
avec deux clés étrangères : `id_professeur` faisant référence à `id` dans `professeur` et (`numero_salle`, `batiment`) faisant référence à (`numero, batiment`) dans `salle`.

Pour une association de cardinalités $1 - 1$, on peut au choix créer une nouvelle table ou ajouter à l'une des deux tables des entités un attribut faisant référence à l'autre entité, comme pour les associations $1 - *$.

Exemple 4. Appliquons cette méthode sur le reste de l'exemple filé, pour en faire un schéma relationnel complet :

E Application

Prenons un exemple de situation concrète, puis modélisons-la sous la forme d'un diagramme EA, avant d'en extraire un schéma relationnel :

> L'enseigne Boutique Nook a ouvert un magasin sur la MP2'Île. Cette boutique vend de très nombreux articles, dont certains existent en plusieurs couleurs. Par ailleurs, elle propose aussi une gamme de papiers peints. Avide de pouvoir, le PDG, Tom Nook, veut imposer sa marque comme monopole, et recueillir autant d'informations que possible sur ses clients. Pour cela, Boutique Nook propose un programme de fidélité : chaque client s'inscrivant à ce programme peut référer ses amis, et gagne des avantages si ceux-ci s'inscrivent à leur tour. Lorsque l'on s'inscrit au programme de fidélité, le magasin offre un papier peint du choix du client.
>
> En outre, afin de fidéliser encore plus sa clientèle, le magasin fait des promotions très régulières. Ces promotions durent un mois entier et peuvent concerner plusieurs articles à la fois.

Modéliser la situation par un diagramme EA puis en déduire un schéma relationnel, en suivant les étapes suivantes :

1. Lister les différentes données que l'on veut pouvoir représenter
2. Déterminer la liste des entités, leurs clés primaires
3. Déterminer la liste des associations, leurs attributs, leurs cardinalités
4. Dessiner le diagramme EA
5. Transformer chaque entité et chaque association en tables.

Annexe : Tables exemples

— `agglomeration(code, nom, nb_habitants, pays)` contient les informations basiques de villes. Le code est un identifiant unique par ville.

code	nom	nb_habitant	pays
pa	PARIS	2000000	France
gr	GRENOBLE	160000	France
na	NANTES	303000	France
re	RENNES	215000	France
ro	ROME	2800000	Italie
mi	MILAN	1350000	Italie
fo	FORLIMPOPOLI	13000	Italie
ma	MADRID	3200000	Espagne
to	TOLEDO	83000	Espagne
se	SEVILLE	680000	Espagne
ox	OXFORD	150000	Angleterre
br	BRUGES	118000	Belgique
bx	BRUXELLES	1250000	Belgique
tl	TOULOUSE	303000	France

— `personne(id_pers, prenom, nom, naissance, pays)` contient les informations basiques de personnes. L'identifiant `id_pers` est unique par personne.

id_pers	prenom	nom	naissance	pays
0	Alice	Grelos	1991	France
1	Bob	Hosta	1988	France
2	Claire	Igname	1995	Italie
3	Dimitri	Jujube	1982	Italie
4	Elena	Kale	1988	Espagne
5	François	Laitue	1992	Belgique
6	Claire	Banane	1853	Italie
7	Claire	Banane	1953	Italie

— `concert(artiste, date, code_agglo, billets)` contient une liste de concerts, et pour chaque concert stocke l'artiste, la date, la ville et le nombre de billets vendus.

artiste	date	code_agglo	billets
BLACKPINK	29-04-2020	pa	80000
Bigflo et Oli	08-10-2019	to	25000
Rage Against the Machine	17-05-2015	bx	41000
Queen	12-12-1980	bx	90000
Queen	13-12-1980	bx	85000

— `habite(id_habite, id_pers, code_agglo, debut, fin)` indique lorsqu'une personne d'identifiant `id_pers` a habité dans la ville de code `code_agglo` entre les dates `debut` et `fin`. L'attribut `idhabite` est un identifiant unique pour chaque enregistrement.

id_habite	id_pers	code_agglo	debut	fin
0	0	re	1991	2012
1	0	pa	2012	2018
2	0	ox	2018	2021
3	0	gr	2021	2022
4	1	pa	1990	1995
5	1	mi	1995	1998
6	1	ro	2000	2020
7	1	fo	2020	2022
8	2	mi	1995	2012
21	2	ro	2012	2020
9	2	br	2020	2022
10	3	ro	1984	2002
11	3	ox	2002	2012
12	3	ma	2012	2020
13	3	se	2020	2022
14	4	ma	1988	2004
15	4	se	2006	2015
16	4	to	2015	2022
17	5	br	1992	1993
18	5	ma	1993	1994
19	5	ro	1994	2018
20	5	gr	2018	2022

— `capitale(id_pays, code_agglo)` indique pour chaque pays sa capitale.

pays	code_agglo
France	pa
Italie	ro
Espagne	ma
Belgique	bx