Cours: Bases de données

25 septembre 2023

1 Introduction : données plates et leurs limites.

Un centre pokémon stocke les données des Pokémon qui viennent se faire soigner dans le centre. Les Pokémon sont regroupés par dresseur. Ils ont certains attributs :

- leur espèce,
- leur dresseur,
- leur type,
- leur lieu de capture,
- le nombre de visites au centre

Par exemple, le début du registre est :

Espèce	Dresseur	Type	Lieuc	Nombre
Carapuce	Sacha	Eau	Jadielle	3
Ratatta	Sacha	Normal	Bourg Palette	2
Leveinard	Sacha	Normal	Jadielle	1
Poissirène	Ondine	Eau	Azuria	4
Onix	Pierre	Pierre	Argenta	2
Evoli	Pierre	Normal	Manoir Céladon	5

Pour l'instant on ne se préoccupe pas de la dernière colonne. Il y a plusieurs manières de représenter ce registre en python. On peut utiliser un dictionnaire ou des listes de listes. On pourrait grouper les Pokémon par dresseur puis par type :

```
pokemon_venus_au_centre=[[[['Carapuce','Sacha','Eau','Jadielle']],
[['Rattata','Sacha','Normal','Bourg Palette'],
['Leveinard','Sacha','Normal','Jadielle']]],
[[['Poissirene','Ondine','Eau','Azuria']]],
[[['Onix','Pierre','Pierre','Argenta']],
[['Evoli','Pierre','Normal','Manoir Celadon']]]]
```

Par exemple, pokemon_venus_au_centre[0] donne et pokemon_venus_au_centre[0][0] donne . Si maintenant on veut récupérer la liste des espèces de pokémon de type eau qui sont venues au centre il faut travailler un peu plus :

```
liste_pokemon_eau=[]
for dresseur in pokemon_venus_au_centre:
   for ptype in dresseur:
      for pokemon in ptype:
         if pokemon[2]=="Eau":
            liste_pokemon_eau.append([pokemon[0])
```

On obtient bien :

```
>>>liste_pokemon_eau
 ['Carapuce', 'Poissirene']
```

On peut choisir de regrouper plutôt les pokemon par type mais dans ce cas il est plus compliqué d'obtenir la liste des pokemon pour chaque dresseur. Si maintenant on veut prendre en compte plus d'informations en utilisant les données suivantes :

lieu	région	ligue
Jadielle	Kanto	Indigo
Argenta	Kanto	Indigo
Bourg Palette	Kanto	Indigo
Mauville	Johto	Johto

nom de la ligue	conseil 1	conseil 2	conseil 3	conseil 4	maître
Indigo	Olga	Aldo	Agatha	Peter	Blue
Johto	Clément	Koga	Aldo	Marion	Peter

Si on essaie de regrouper toutes les données dans un même document et qu'on utilise des listes sous Python il sera laborieux de prendre en compte toutes les informations et d'y avoir accès. Les bases de données relationnelles vont permettre de séparer les données en plusieurs sous tableaux et de faire des liens entre les différents tableaux.

2 Vocabulaire des bases de données relationnelles.

Définition 1

- *Une **base de données relationnelle** est un gros ensemble d'informations structurées et mémorisées sur un support permanent.*
- *Un **Systéme de Gestion de Bases de Données** (SGBD) est un logiciel de haut niveau qui permet de manipuler les informations stockées dans une base de données.*

Il existe de nombreux types (élémentaires) de données :

- entiers (*int*, *bigint*, *smallint*, ...)
- réels (*float*, *real*, ...)
- chaînes de caractères (*string*, *text*, *char(m) varchar(m)*, ...)
- dates (*date*, *time*, *datetime*, ...)
- booléens (*bool*, ...)

On a précisé dans la définition d'une base de données relationnelles que les données étaient structurées. Concrètement les données sont stockées sous forme de tableaux à deux dimensions appelés **relation** ou **table** et il peut y avoir des liens entre les différents tableaux. Chaque relation a un nom.

Définition 2

On considère une relation, on appelle :

- ***Attributs :** nom donné aux colonnes de la relation.*
- ***Domaine d'un attribut :** ensemble des valeurs de type élémentaire que peut prendre l'attribut. Cet ensemble est défini au moment de la création de la relation.*
- ***Enregistrement,n-uplet ou tuple** : une ligne de la relation.*
- ***Schéma de la relation** : Nom de la relation suivi de la liste des attributs et de leur domaine.*

Par exemple la relation donnée par le premier tableau qu'on appellera CENTRE a cinq attributs. Son schéma est donc :

Les enregistrements doivent tous être différents. Pour garantir la non-répétition des enregistrements, les bases de données réelles contiennent un concept de clé qui permet d'identifier chaque enregistrement. Lors de la création de la relation les clés primaires et les clés étrangères sont déclarées.

Définition 3

*La **clé primaire** d'une relation est le plus petit sous ensemble d'attributs permettant d'identifier chaque enregistrement de manière unique.*

Prenons l'exemple de la relation CENTRE. Plusieurs cas possibles :

- On considère que chaque dresseur capture au maximum une fois un même pokémon. Dans ce cas l'ensemble { espèce,dresseur } peut être une clé primaire.
- On considère que chaque dresseur capture au maximum une fois un même pokémon au même endroit. Dans ce cas l'ensemble { espèce, dresseur, lieu } peut être une clé primaire.
- Sinon il y a un problème lors de la conception de notre base de données et si Sacha capture deux carapuces à Jadielle et qu'il veut les faire soigner on ne pourra pas enregistrer le deuxième carapuce.

Il existe un moyen de contourner ce problème,on peut créer artificiellement un identifiant qui sera la clé primaire, on parle dans ce cas aussi de clé artificielle. Par exemple on considère notre nouvelle relation CENTRE donnée par :

id	espèce	dresseur	type	lieuc	nombre
1	Carapuce	Sacha	Eau	Jadielle	3
2	Ratatta	Sacha	Normal	Bourg Palette	2
3	Leveinard	Sacha	Normal	Jadielle	1
4	Poissirène	Ondine	Eau	Azuria	4
5	Onix	Pierre	Pierre	Argenta	2
6	Evoli	Pierre	Normal	Manoir Céladon	5

Dans ce cas le schéma de la relation est :
CENTRE(<u>id</u> : int , espèce : string,dresseur :string,type :string,lieuc : string, nombre : int)
Par convention, **les attributs de la clé primaire** sont **soulignés**. Enfin il existe un autre type de clé pour pouvoir prendre en compte les liens entre les relations. Ce sont les clés étrangères.

Définition 4

Une clé étrangère, c'est un attribut (ou groupe d'attributs) d'une relation qui fait référence à la clé primaire d'une autre relation, afin de modéliser le lien entre les enregistrements de ces deux relations.

Les clés étrangères sont souvent préfixées d'un # .

Définition 5

*Une base de données est représentée par son **schéma relationnel**. C'est la description de l'ensemble des relations constituant la base.*

Le schéma relationnel de notre base de données POKEMON est :

- CENTRE(<u>id</u> :int, espèce :string,dresseur :string,type :string, # lieuc :string,nombre :int)
- LIEU(<u>lieu</u> :string,région :string,# ligue :string)

- LIGUE(ligue :string,conseil 1 : string,conseil 2 : string, conseil 3 :string,conseil 4 : string,maître : string)

ou de manière équivalente :

2.1 Pourquoi insister sur le relationnel ?(Hors programme).

Il existe d'autres bases de données qui ne sont pas structurées (les bases de données Not Only SQL (NoSQL)). Les transactions d'une bases de données relationnelles (opération au sein d'une base de donnée comme la création d'un nouvel enregistrement ou une mise à jour des données) doivent vérifier les propriétés ACID : ACID signifie atomicité, cohérence, isolation et durabilité. Globalement cela signifie que les données sont stockées de manière sûre et sécurisée et que les transactions sont traitées de manière fiable et cohérente. Pour les bases de données NoSQL les règles ACID ne sont pas suivies. Pour le NoSQL les données peuvent être distribuées et répliquées sur plusieurs serveurs (ce qui n'est pas le cas en SQL) ce qui nous fait perdre soit la disponibilité soit la cohérence. Les bases de données NoSQL vont vérifiées les propriétés BASE :

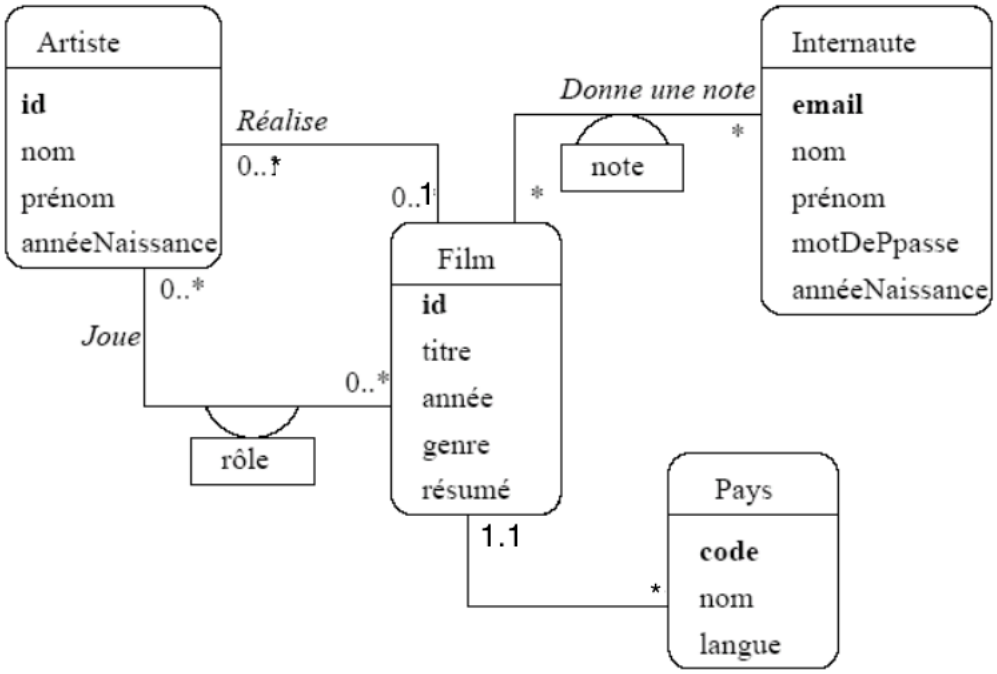
- Basically Available : Peu importe notre requête, la base de donnée doit fournir une réponse (disponibilité).
- Soft-state : La base peut changer lors des mises à jour ou lors d'ajout/suppression de serveurs. La base NoSQL n'a pas à être cohérente à tout instant.
- Eventually consistent : À terme, la base atteindra un état cohérent.

2.2 Entités/associations.

La création d'une base de données relationnelle est parfois précédée de la création d'un schéma **entités-associations**.

Exemple 1

Le schéma ***entités-associations*** *d'une bases de données* FILMS



Par exemple ici, Artiste est une **entité** qui a quatre **propriétés** qui sont : id,nom,prénom et annéeNaissance. Il y a une **association** entre les entités Artiste et Film. Une association est caractérisée de manière concise en ne donnant que les cardinalités maximales aux deux extrémités. Donc on oublie les 0.. et les 1...

Cependant, bien que considérant comme acquis que :

- Un artiste peut réaliser plusieurs films.
- Un film est réalisé par un unique artiste

L'association peut être notée, comme sur le schéma ci-dessus :

$$\boxed{\text{Artiste}} - * - \text{réalise} - \mathbf{1} - \boxed{\text{Film}}$$

Règles de passage du schéma **entités-associations** au modèle **relationnel** :

Propriété 1 (*Règle n°1*)

Chaque entité B devient une relation R_B. Chaque propriété de l'entité devient un attribut de la relation. L'identifiant de l'entité B devient la clé primaire de R_B.

Ainsi, dans l'exemple, sont créées les relations :
FILM (`idFilm` ; `titre` ; `année` ; `genre` ; `résumé`)
INTERNAUTE (`email` ; `nom` ; `prénom` ; `motDePasse` ; `annéeNaissance`)
ARTISTE (`idArtiste` ; `nom` ; `prénom` ; `annéeNaissance`)
PAYS (`code` ; `nom` ; `langue`)

Propriété 2 (*règle n°2*)

Une association $\boxed{B - 1 : * - C}$ *(un à plusieurs) de l'entité B vers l'entité C est modélisée par la création d'un attribut, formant une clé étrangère, dans R_B associée à l'identifiant de C, ie*

la clé primaire de R_C.

Ainsi, avec Film 1 : réalise : ∗ Artiste et Film 1 : ∗ Pays, on a les ajouts suivants :
FILM (idFilm ; titre ; année ; genre ; résumé ; # réalisateur ; # codePays)

Propriété 3 (*Règle n°3*)

Une association $B - * : * - C$ (plusieurs à plusieurs) de l'entité B vers l'entité C est modélisée par la création d'une relation R_{BC}.Les clés primaires de R_B et R_C deviennent des attributs de R_{BC}, ce couple d'attributs pouvant devenir la clé primaire de R_{BC}.Si l'association possédaient des propriétés, elles deviennent des attributs de R_{BC}.

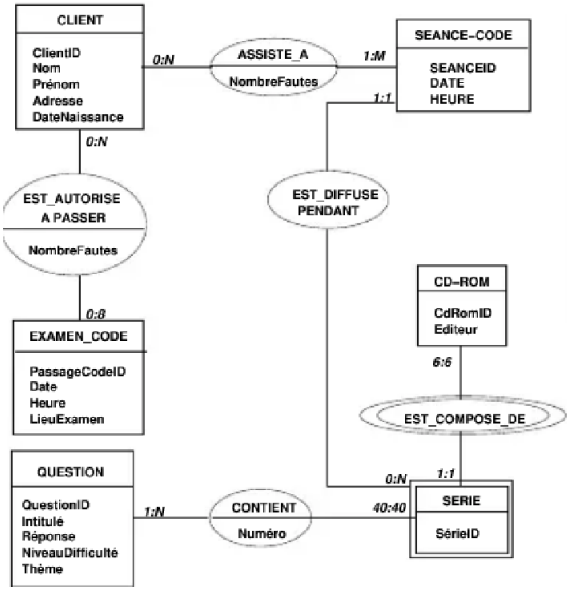
Donc B – ∗ : ∗ – C devient B – ∗ : 1 –BC et BC – 1 : ∗ – C puis on utilise la règle 2. Ainsi, avec Artiste – ∗ : joue : ∗ – Film et internaute – ∗ : ∗ – Film, sont créées les deux tables :
NOTATION (#email ; #idFilm ; note) et ROLE (#idFilm ; #idActeur ; nomRole)

Exercice 1

On considère ci-contre le diagramme entités/associations associé à la modélisation d'une base de données d'une auto-école. En particulier :

- *Une même question peut apparaître dans plusieurs séries avec un numéro d'ordre différent.*
- *La base doit permettre de répondre à la question : "Quel est le nombre moyen de fautes fait à la question 15 de la série 6 du CdRom 14 ?".*

Interpréter ce diagramme et écrire le schéma relationnel associé.



•

3 Langage SQL.

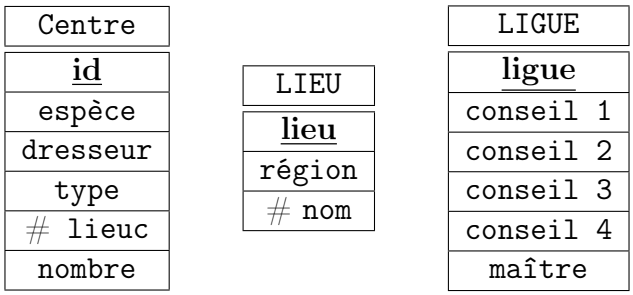
3.1 Généralités.

Définition 6

*Le **langage SQL** (Structured Query Language) est le principal langage d'interrogation et de mise à jour des bases de données relationnelles.*

- SQL est un langage non procédural (déclaratif) : L'utilisateur spécifie uniquement " quoi faire" et **non** "comment faire ".
- PYTHON est un langage procédural(impératif) : L'utilisateur doit spécifier " quoi faire" et **également** "comment faire" par une suite d'instructions séquentielles.

On considère désormais la base de données Pokémon :



3.2 Création et modification de tables :HORS PROGRAMME.

La table CENTRE(id :int, espèce :string,dresseur :string,type :string, # lieu :string,nombre :int) peut être créée avec :

```
CREATE TABLE Centre (
id  INTEGER NOT NULL UNIQUE,
espece TEXT,
dresseur TEXT,
type TEXT
lieuc TEXT,
nombre INTEGER,
PRIMARY KEY(id),
FOREIGN KEY(lieuc REFERENCES Lieu (lieu) )
```

On déclare à chaque fois le domaine de l'atttribut, on précise les clés primaires et étrangères. On remarque au passage que les attributs n'ont pas forcément le même nom selon la relation. Une clé primaire ne peut pas être vide d'où la précision NOT NULL (NULL est utilisée lorsque la donnée est inconnue) et doit être unique sinon ce n'est pas une clé primaire. On peut ensuite insérer des enregistrements dans notre relation

```
INSERT INTO Centre VALUES (1,'Carapuce','Sacha','Eau','Jadielle',3);
INSERT INTO Centre VALUES (2,"Ratatta','Sacha','Normal','Bourg Palette',2)
```

Exercice 2

1. *Combien de fois Rattata a-t-il été soigné dans notre centre pokémon ?*

2. *Insérer dans la bonne relation les informations, référencées par l'entier 25, concernant le Tortank de Sylver qui vient se faire soigner pour la première fois. Le lieu de capture est inconnu.*
3. *Insérer dans la bonne relation les informations précisant que la ville de Jadielle se situe dans la région de Kanto et que la ligue de cette région est la ligue Indigo.*

3.3 Syntaxe de base.

La forme la plus simple d'une requête en SQL pour une recherche dans une seule relation est de la forme :

```
SELECT ..... FROM relation ......;
```

Par habitude, on écrit les mots-clés en majuscule. Une requête en SQL commence TOUJOURS par SELECT ! Le mot-clé FROM permet de préciser le nom de la relation à utiliser. On finit toujours une requête par un point virgule.

3.3.1 Les projections.

Les requêtes en SQL sont étroitement liées à l'algèbre relationnelle (basée sur la théorie des ensembles). L'algèbre relationnelle est hors programme mais le vocabulaire associé reste très pratique à manipuler. Une projection en SQL consiste à ne garder qu'un certain nombre d'attributs de la relation. Le mot-clé SELECT permet de projeter (ie ne garder que les attributs ou des fonctions des attributs qui nous intéressent). Il suffit juste de préciser après SELECT les attributs que l'on veut garder séparés par des virgules. Si $A1, A2, ..., Ap$ sont les attributs que l'on souhaite garder de la relation Exemple on utilise

```
SELECT A1,A2,A3,...,Ap FROM Exemple;
```

Si l'on souhaite garder tous les attributs alors on utilise * :

```
SELECT * FROM Exemple;
```

Attention en SQL les doublons résultant de projections ne sont pas supprimés, il faut utiliser les mots-clefs SELECT DISTINCT pour supprimer les doublons.

Exercice 3

1. *Ecrire une requête permettant de récupérer la liste de tous les dresseurs ayant soigné au moins un pokémon dans notre centre.*
2. *Ecrire une requête permettant de récupérer la liste de tous les dresseurs sans doublons ayant soigné au moins un pokémon dans notre centre.*

3.3.2 Sélection puis sélection et projection.

La sélection permet de ne garder que les enregistrements de la relation qui vont vérifier certaines conditions. En SQL, on utilise le mot-clé WHERE placé après le nom de la relation. Pour écrire la condition on peut utiliser comme en Python :

- Les opérateurs de calculs : $+, -, *, /$
- Les opérateurs de comparaisons : $=, <, <=, >, >=, <>$
- Les opérateurs booléens : `NOT`, `AND` et `OR`

Généralement on utilise ensemble la sélection et la projection sous la forme

```
SELECT Attributs FROM Relation WHERE Conditions;
```

Remarque 1

La sélection est effectuée avant la projection. On retire les enregistrements qui ne satisfont pas notre condition puis on ne garde que les colonnes qui nous intéressent.

Exercice 4

1. *Ecrire une requête qui renvoie la liste des dresseurs ayant soigné au moins un pokémon eau au centre.*
2. *Ecrire une requête qui renvoie la liste des Pokémon eau qui ont été soignés une seule fois au centre ainsi que leurs dresseurs.*
3. *Ecrire une requête qui renvoie la liste des pokémons qui ont été capturés par Sacha ou à Jadielle. On suppose qu'il n'y a pas d'échanges de Pokémon.*

Le point suivant est hors-programme mais vous pourriez un jour y être confronté. On peut aussi utiliser la commande LIKE pour comparer deux chaînes de caractères. Dans la pratique on a

```
SELECT Attributs FROM Relation WHERE Chaine_de_caractere LIKE __A%B;
```

avec _ qui remplace n'importe quel caractère et % qui remplace n'importe quelle chaîne de caractère. Par exemple si on veut la liste de tous les pokémons qui ont été soignés au centre et dont la première lettre comme par un c on utiliserait la commande :

3.3.3 Renommage et commentaire.

Il peut parfois être utile de commenter ses requêtes SQL. On utilisera pour cela le double tiret du 6 −−. Par la suite nous allons effectuer des requêtes impliquant plusieurs relations. Il peut parfois être nécessaire de renommer les relations ou les attributs d'une relation.
Pour **renommer (temporairement) un ou plusieurs attributs** on utilise le mot clé AS (ou rien) de la manière suivante :

```
SELECT A1 AS NOUVEAUNOM1,A2 AS NOUVEAUNOM2, A3,....,Ap FROM Relation;
```

On peut aussi juste écrire le nouveau nom à la suite en les séparant d'un espace :

```
SELECT A1 NOUVEAUNOM1,A2 NOUVEAUNOM2, A3,....,Ap FROM Relation;
```

On peut aussi **renommer temporairement une relation**. C'est particulièrement utile lorsque notre requête implique deux relations qui ont chacune un attribut qui porte le même nom. Si la relation R1 et la relation R2 possèdent chacune l'attribut A1 alors pour lever l'ambiguïté il faudra utiliser R1.A1 ou R2.A1 dans notre requête sinon le logiciel ne pourra pas savoir à quel attribut nous faisons référence.

```
SELECT .....FROM Relation AS R;
```

ou

```
SELECT ..... FROM Relation R;
```

3.4 Agrégats et fonctions d'agrégations.

3.4.1 Les fonctions d'agrégation.

On a vu précédemment qu'on pouvait appliquer des fonctions aux attributs d'une relation. La fonction s'applique à cet attribut dans chaque enregistrement. Par exemple si on avait une relation Devoir(id : int, nom :string,prenom :string,note_ sur _ 20 :int) alors pour avoir la relation avec les notes sur 10 on utiliserait :

```
SELECT nom, prenom, note_sur_20/2 FROM Devoir;
```

Il est également possible d'appliquer des fonctions utilisant un ensemble d' enregistrements d'une relation appelées **fonctions d'agrégations**. Les fonctions de "base" sont les suivantes :

- AVG() pour calculer la moyenne sur un ensemble d'enregistrement.
- COUNT() pour compter le nombre d'enregistrements.
- COUNT(*) permet de compte le nombre d'enregistrements de la relation.
- MAX() pour récupérer la valeur maximum d'une colonne sur un ensemble d'enregistrements. Cela s'applique à la fois pour des données numériques ou alphanumériques.
- MIN() pour récupérer la valeur minimum de la même manière que MAX().
- SUM() pour calculer la somme sur un ensemble d'enregistrements.

3.4.2 Agrégation sans regroupement.

Concrètement une requête simple sera du type :

```
SELECT fonction(attributs) FROM Relation;
```

Cette requête renvoie une relation avec un unique enregistrement et un unique attribut. C'est la valeur prise par la fonction. Par exemple, pour obtenir la moyenne sur 20 au devoir on utilise :

```
SELECT AVG(note_sur_20) FROM Devoir;
```

3.4.3 Agrégation avec regroupement.

On peut regrouper les enregistrements ayant certaines attributs en commun pour ensuite utiliser une fonction d'agrégation. Pour cela on utilise le mot-clé GROUP BY. On l'utilise de la manière suivante :

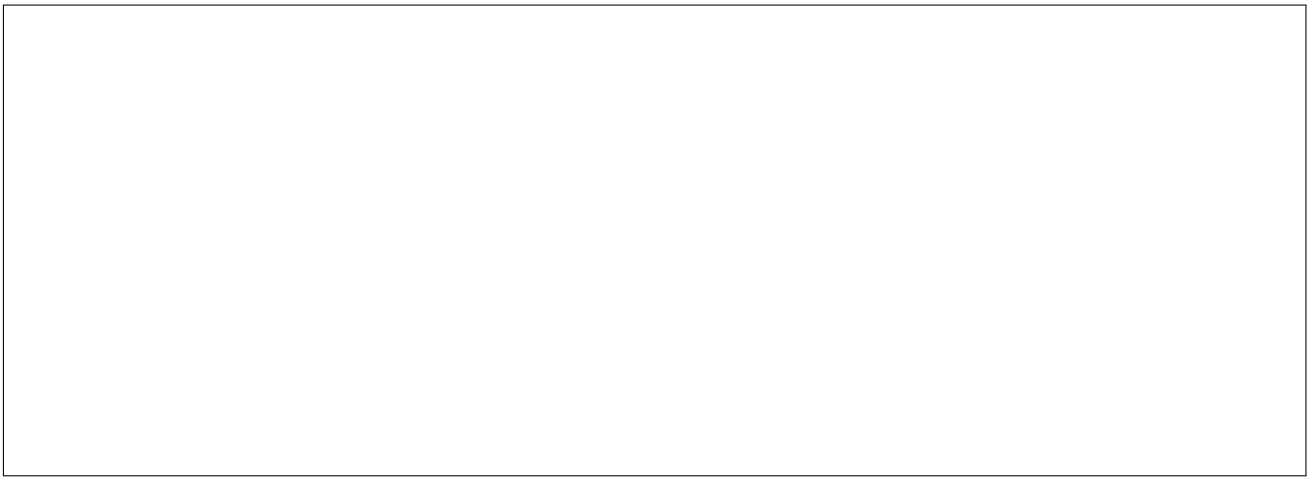
```
SELECT A1, fonction(A2)
FROM Relation
GROUP BY A1;
```

Un seul enregistrement est renvoyé par groupe.

Exercice 5

En reprenant la base de donnée POKEMON de la page 7 :

1. *Ecrire une requête qui renvoie la moyenne du nombre de fois qu'un pokémon se fait soigner à notre centre en fonction de son type.*
2. *Ecrire une requête qui renvoie pour chaque dresseur le nombre maximum de fois qu'un dresseur est venu pour un même pokémon.*



On peut tricher un peu avec la syntaxe mais attention à la cohérence. Il faut que le contenu du SELECT soit cohérent : soit des attributs du GROUP BY ou qui sont identiques pour chaque élément du groupe, soit des fonctions d'agrégation (qui porteront sur chaque groupe). Prenons un exemple concret. Si on effectue la requête suivante :

```
SELECT dresseur ,AVG(nombre)
FROM Centre
GROUP BY type;
```

Le problème est le suivant : lorsque l'on effectue le regroupement par type de Pokémon il y a par exemple des Pokémon eau qui ont pour dresseur Sacha et des Pokémon eau qui ont pour dresseur Ondine. Les deux éléments ne sont pas identiques donc on ne peut pas mettre l'attribut dresseur

dans le SELECT. Par contre si chaque type était associé à un seul et unique dresseur alors il n'y aurait pas de problème.

3.5 Sélections et agrégation.

La sélection (imposer des conditions aux enregistrements) peut se faire avant ou après l'agréation en SQL. Pour effectuer une sélection avant l'agrégation c'est le mot-clé WHERE qu'on utilise tandis que pour faire une sélection après l'agrégation on utilise le mot-clé HAVING. Par exemple pour ne garder que les types des Pokémon qui ont été soignés en moyenne au moins trois fois dans le centre ainsi que la moyenne associée on utilise :

```
 SELECT type, AVG(nombre) as moyenne
 FROM Centre
GROUP BY type
HAVING moyenne>=3;
```

Si maintenant on ne veut garder que les types de Pokémon de Sacha qui ont été soignés en moyenne au moins trois fois dans le centre ainsi que la moyenne associée on utilise :

```
SELECT type, AVG(nombre) as moyenne
FROM Centre
WHERE dresseur='Sacha'
GROUP BY type
HAVING moyenne>=3;
```

Exercice 6

Ecrire une requête donnant la liste des dresseurs qui viennent au maximum quatre fois pour un pokémon feu au centre ainsi que ce maximum.

Comme pour le SELECT, dans la condition du HAVING doivent apparaître soit des attributs du GROUP BY qui sont identiques pour chaque élément du groupe, soit des fonctions d'agrégations (qui porteront sur chaque groupe).

Remarque 2

Il ne faut pas confondre WHERE et HAVING :

- *S'il n'y a pas de fonction d'agrégation dans la requête il n'y a pas besoin de HAVING.*
- *WHERE est appliqué avant l'agrégation tandis que HAVING est appliqué après l'agréation.*
- *Il est parfois possible d'écrire la requête avec l'un ou l'autre !*

Remarque 3

On pourra retenir que :

- *On n'écrit pas de GROUP BY s'il n'y a pas de fonction d'agrégation.*
- *On n'écrit pas de HAVING s'il n'y a pas de GROUP BY.*

3.6 Gérer l'affichage des résultats.

On peut rajouter certains mots-clés à la fin de notre requête pour modifier l'affichage des résultats.

3.6.1 ORDER BY : Ordonner les résultats.

La syntaxe est la suivante :

```
SELECT Attributs,fonctions(Attributs) FROM Relation ORDER BY Attributs;
```

On peut :

- Classer les résultats par ordre croissant ou décroissant à l'aide des mots clés ASC et DESC juste après l'attribut ou la fonction d'agrégation dans le ORDER BY.
- Si on met plusieurs attributs ou fonctions d'agrégation appliquées à des attributs séparés par des virgules après ORDRE BY la relation est triée par ordre lexicographique.

3.6.2 Affichage d'une plage d'enregistrements.

Le mot-clé LIMIT plage avec plage un entier naturel limite le nombre de résultats à plage. On peut préciser la première ligne que l'on veut voir apparaître grâce à OFFSET. Si l'on veut commencer à la ligne l avec l un entier naturel non nul on utilise OFFSET $l-1$. Ainsi LIMIT plage OFFSET l affiche les enregistrements des lignes $[\![l+1, l+plage]\!]$. Par défaut $l=0$. Par exemple pour afficher les cinq premiers Pokémon eau classés dans l'ordre alphabétique a être venus se faire soigner au centre on utilise :

```
SELECT espece
FROM Centre
WHERE type='Eau'
ORDER BY espece ASC
LIMIT 5
```

3.7 Résumé pour une requête sur une seule relation.

Les différents mots-clés ne sont pas toujours présents. SELECT et FROM sont toujours INDISPENSABLES. L'ordre des mots-clés est à connaitre !

```
SELECT Attributs, fonction(Attributs)
FROM Relation
WHERE Conditions
GROUP BY Attributs
HAVING Conditions
ORDER BY attribut ordre
LIMIT entier naturel non nul
OFFSET entier naturel
```

3.8 Composition de requête.

On se pose la question suivante : Quels sont les types de pokémons qui vérifient la propriété suivante : "En moyenne un pokémon vient se faire soigner trois fois dans notre centre." ? Il y a plusieurs manières de répondre à la question. On voudra récupérer aussi le nombre moyen de visite pour un pokémon de ce type. La première :

```
SELECT type, AVG(nombre) moy
FROM Centre
GROUP BY type
HAVING moy= 3
```

Une autre solution est de combiner deux requêtes. On rappelle que le résultat d'une requête est une relation !

```
SELECT Type,moy
FROM(
SELECT Type, AVG(nombre) moy FROM Centre GROUP BY Type) AS Relation2
WHERE moy=3
```

On effectue une première requête où l'on utilise la relation Centre. On regroupe les données selon le type et on calcule la moyenne du nombre de visites par type. La relation Relation2 a deux attributs : le type et le nombre moyen de visites par type et ensuite on sélectionne dans cette table les enregistrements dont la moyenne est égale à 3.

3.9 Requête impliquant plusieurs relations.

3.9.1 Produit cartésien et jointure.

On considère notre base de donnée pokémon :

- CENTRE(id :int, espèce :string,dresseur :string,type :string, # lieuc :string,nombre :int)
- LIEU(lieu :string,région :string,# ligue :string)
- LIGUE(ligue :string,conseil 1 : string,conseil 2 : string, conseil 3 :string,conseil 4 : string,maître : string)

On va considérer un extrait des deux premières relations :

Identifiant	Espèce	Dresseur	Type	Lieu de capture	Nombre de visite
1	Carapuce	Sacha	Eau	Jadielle	3
5	Onix	Pierre	Pierre	Argenta	2

Lieu	Région	ligue
Jadielle	Kanto	Indigo
Argenta	Kanto	Indigo
Bourg Palette	Kanto	Indigo

Pour savoir dans quelle région a été capturé le Carapuce de Sacha on doit mettre en relation les deux tables. La première idée est d' effectuer le produit cartésien de ces deux relations. Chaque enregistrement est un élément de la relation. Ainsi on doit avoir l'ensemble des couples $(e_1, e_2), e_1 \in$Centre,$e_2 \in$ Lieu :

Id	espèce	dresseur	type	lieuc	nombre	lieu	région	ligue
1	Carapuce	Sacha	Eau	Jadielle	3	Jadielle	Kanto	Indigo
1	Carapuce	Sacha	Eau	Jadielle	3	Argenta	Kanto	Indigo
1	Carapuce	Sacha	Eau	Jadielle	3	Bourg Palette	Kanto	Indigo
5	Onix	Pierre	Pierre	Argenta	2	Jadielle	Kanto	Indigo
5	Onix	Pierre	Pierre	Argenta	2	Argenta	Kanto	Indigo
5	Onix	Pierre	Pierre	Argenta	2	Bourg Palette	Kanto	Indigo

En SQL la commande est

```
SELECT ... FROM Relation 1 ,Relation 2 , Relation 3....;
```

ou

```
SELECT ... CROSS JOIN Relation 1 ,Relation 2 , Relation 3....;
```

On obtient donc beaucoup d'enregistrements dont une grande proportion n'est pas pertinente. Pour ne garder que les enregistrements pertinents il faut s'assurer que le lieu de capture correspond au lieu de la relation LIEU. C'est ce qu'on appelle faire une **jointure**. On pourrait utiliser WHERE pour ne garder que les enregistrements pertinents. Cette méthode était autrefois utilisée mais elle pose le problème suivant : on ne fait plus la distinction entre ce qui relève de la sélection (et qui dépend de la requête) et ce qui relève de la jointure (dépendant plutôt de la structure logique du découpage en relations de la base de données). Pour pallier à ce problème on utilise les mots-clés JOIN.... ON :

```
SELECT espece,region FROM Centre JOIN Lieu ON lieuc = lieu;
```

Si on veut faire plus d'une jointure la requête s'écrit :

```
SELECT ...
FROM Relation 1
JOIN Relation 2 ON condition
JOIN Relation 3 ON condition;
```

Exercice 7

1. *Ecrire une requête qui renvoie la liste des Pokémon qui ont été soignés au centre et qui ont été capturés à Kanto.*
2. *Ecrire une requête qui renvoie le nombre de Pokémon de chaque type ainsi que le type associé qui ont été soignés au centre et qui ont été capturés à Kanto.*
3. *Ecrire une requête qui renvoie la liste des Pokémon qui sont venus au centre et qui ont été capturés dans une région où le maître de la ligue est Peter.*

3.9.2 Autojointure.

On peut aussi faire joindre une relation avec elle-même dans ce cas on parle **d'autojointure**. Cette possibilité est d'ailleurs très appréciée des sujets de concours... Dans ce cas il faut absolument renommer au moins une des deux relations ! Par exemple récupérons tous les couples possibles de Pokémon de Sacha qui ont été soignés au centre.

```
SELECT c1.espece, c2.espece
```

```
FROM Centre as c1 JOIN Centre AS c2 ON c1.dresseur=c2.dresseur
WHERE c1.dresseur='Sacha' AND c1.id<c2.id;
```

La dernière condition sert à éviter d'avoir des doublons et d'avoir un couple formé avec un pokémon et lui-même !

3.9.3 Opérations ensemblistes.

Il existe d'autres opérations ensemblistes que le produit cartésien.

3.9.4 UNION et UNION ALL.

Le mot-clé UNION de SQL permet de mettre bout-à-bout les résultats de plusieurs requêtes utilisant elles-même la commande SELECT. Pour l'utiliser il est nécessaire que chacune des requêtes à concaténer retournes le même nombre d'attributs, avec les mêmes types de données et dans le même ordre. Les doublons ne sont pas répétés. Si l'on souhaite obtenir les doublons on utilisera UNION ALL. On a donc en SQL une requête du type :

```
SELECT* FROM Relation1 UNION (ALL) SELECT * FROM Relation 2;
```

3.9.5 Intersection.

Le mot-clé INTERSECT permet d'obtenir l'intersection des résultats de deux requêtes. Cela peut s'avérer utile lorsqu'il faut trouver s'il y a des données similaires sur deux relations distinctes. Pour l'utiliser convenablement il faut que les deux requêtes retournent le même nombre d'attributs, avec les mêmes types et dans le même ordre. La syntaxe est la suivante :

```
SELECT* FROM Relation1 INTERSECT SELECT * FROM Relation 2;
```

Pour savoir si Sacha et Ondine ont soigné des pokémons de la même espèce au centre :

```
SELECT Especе FROM Centre WHERE Dresseur='Sacha'
INTERSECT
SELECT Espce FROM Centre WHERE Dresseur='Ondine';
```

3.9.6 Privée de .

Le mot-clé EXCEPT permet d'obtenir les résultats de la première requête qui ne sont pas dans la deuxième requête. Pour l'utiliser convenablement il faut que les deux requêtes retournent le même nombre d'attributs, avec les mêmes types et dans le même ordre. La syntaxe est la suivante :

```
SELECT* FROM Relation1 EXCEPT SELECT * FROM Relation 2
```

Pour savoir si Sacha a des espèces de pokémon qu' Ondine n'a pas (du point de vue du centre) :

```
SELECT Especе FROM Centre WHERE Dresseur='Sacha'
EXCEPT
SELECT Especе FROM Centre WHERE Dresseur='Ondine'
```

3.9.7 La division ensembliste : HORS PROGRAMME.

Définition 7 (*division ensembliste*)

Soit les relations $R(a_1 : A_1, ..., a_p : A_p, a_{p+1} : A_{p+1}, ..., a_n : A_n)$, $R2(a_{p+1} : A_{p+1}, ..., a_n : A_n)$. *Le quotient de la relation R par la relation R2 est défini par :*

$$R \div R2 = \{(t_1, ..., t_p) \in A_1 \times ... \times A_p : \forall (t_{p+1}, ..., t_n) \in R2, (t_1, ..., t_p, t_{p+1}, ..., t_n) \in R\}$$

Théorème 1

En notant π *les projections (analogue de select en algèbre relationnelle) : Si* $R1 = \pi_{A1,A2...Ap}(R)$ *et* $R1' = \pi_{A1,A2...Ap}((R1 \times R2) - R)$ *alors*

$$R \div R2 = R1 - R1'$$

Exemple 2

R :

NomEleve	*Sport*
Jacques	*Athlétisme*
Pierre	*Athlétisme*
Paul	*Cyclisme*
Pierre	*Cyclisme*
Paul	*Football*
Pierre	*Football*

$R2$:

Sport
Athlétisme
Cyclisme
Football

On a : $R1$:

NomEleve
Jacques
Pierre
Paul

,

$R1 \times R2$:

NomEleve	*Sport*
Jacques	*Athlétisme*
Jacques	*Cyclisme*
Jacques	*Football*
Pierre	*Athlétisme*
Pierre	*Cyclisme*
Pierre	*Football*
Paul	*Athlétisme*
Paul	*Cyclisme*
Paul	*Football*

, $(R1 \times R2) - R$:

NomEleve	*Sport*
Jacques	*Cyclisme*
Jacques	*Football*
Paul	*Athlétisme*

Puis : $R1'$

NomEleve
Jacques
Paul

, $R1 - R1'$:

NomEleve
Pierre

$R \div R2 =$ | *Pierre* | *est la réponse à la requête "Quels élèves pratiquent tous les sports ?"*

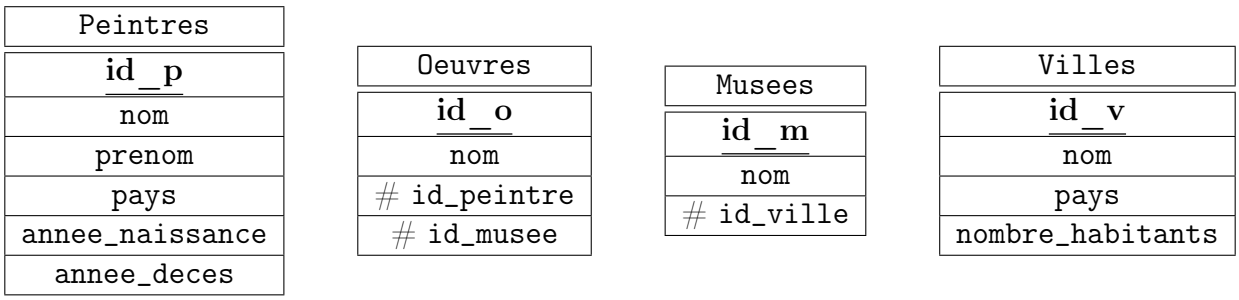
4 A vous de jouer !

Ci-joint le schéma relationnel d'une base PEINTURE :

- PEINTRES (`id_p` : *int* ; `nom` : *string* ; `prenom` : *string* ; `pays` : *text* ; `annee_naissance` : *int* ; `annee_deces` : *int*)
- MUSEES (`id_m` : *int* ; `nom` : *text* ; # `id_ville` : *int*)
- OEUVRES (`id_o` : *int* ; `nom` : *varchar(20)* ; # `id_peintre` : *int* ; # `id_musee` : *int*)

- VILLES (id_v : *int* ; nom : *text* ; pays : *text* ; nombre_habitants : *int*)

Ou noté de façon équivalente :



1. Donner le nom et la durée de vie des peintres en se limitant aux lignes 11 à 58 incluses de la table.
2. Lister les noms des villes par ordre de population décroissante puis par ordre alphabétique de leur nom de pays.
3. Lister tous les informations de la table œuvres par ordre des identifiants des peintres.
4. Lister au plus 50 identifiants de musées dont l'identifiant de ville est écrit avec un A en troisième position et B en dernière position.
5. Lister les noms des peintres nés au quinzième siècle.
6. Donner les identifiants des peintres dont aucune oeuvre n'est exposée dans un musée.
7. Donner tous les couples formés par un nom de peintre et un nom de musée.
8. Donner les identifiants des villes italiennes contenant au moins un musée.
9. Lister les noms des peintres exposés dans un musée.
10. Lister, pour le musée du Louvre, le nom de toutes les oeuvres ainsi que le nom du peintre qui en est l'auteur. (on admet qu'un seul musée porte ce nom).
11. Lister les noms des musées qui exposent une toile de Raphael.
12. Lister les noms des peintres originaires du même pays que Botticelli.
13. Lister le nom des oeuvres de Léonard de Vinci.
14. Donner le nombre total d'oeuvres exposées.
15. Donner le nom et le nombre d'habitants des pays possédant au moins un million d'habitants.
16. Donner le nom d'un musée ayant le moins d'oeuvres.
17. Donner les noms des musées ayant plus d'oeuvres que la moyenne.
18. Donner les identifiants des peintres dont aucune oeuvre n'est exposée.
19. Donner les identifiants des peintres exposés dans tous les musées.
20. Donner les noms des peintres exposés dans tous les musées parisiens.

    Ce sont les peintres pour lesquels il n'existe pas de musée parisien qui ne les expose pas.

    Pour un peintre pe donné :

    On sélectionne A l'ensemble des identifiants musées parisiens.

    On sélectionne $B(pe)$ l'ensemble des identifiants des musées (parisiens) qui expose au moins une oeuvre du peintre pe.

    On retourne les noms des peintres pe pour lesquels $A - B(pe) = \emptyset$