Informatique

1

Bases de données

Cours

Bases de données

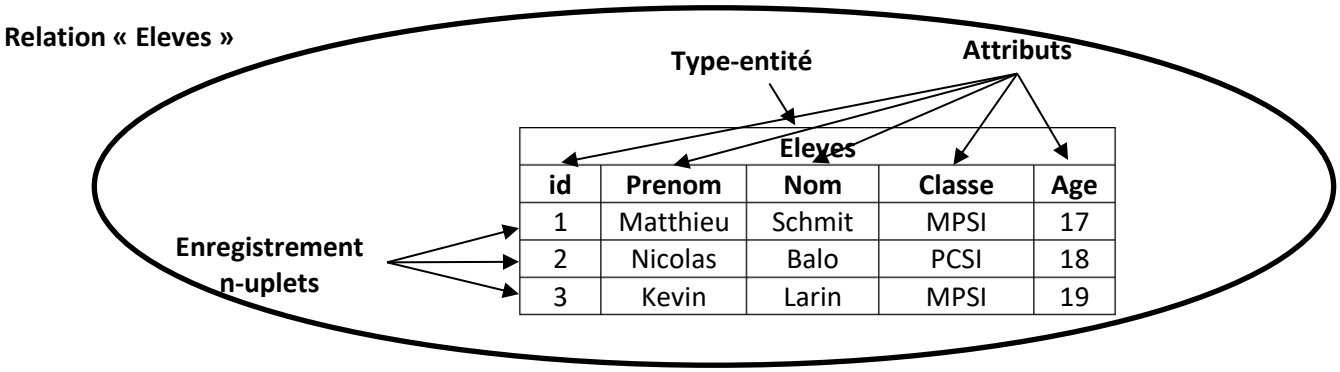
1.I. Définition

Une base de données (BDD) est, comme son nom l'indique, un outil de stockage de données. Elle permet la collecte, le stockage et l'utilisation des données associées.

Le système de gestion des bases de données s'appelle le « SGBD » pour « Système de Gestion de Base de Données ». Il permet d'accéder aux données de la base de données par l'intermédiaire de requêtes simples qui cachent une manipulation complexe des données.

1.II. Vocabulaire

Le programme d'informatique propose d'aborder le concept de bases de données relationnelles, c'est-à-dire dans laquelle l'information est organisée dans des tableaux à deux dimensions appelés des **relations** ou des **tables**. Les **lignes** de ces tables sont appelées **n-uplets** (tuples en anglais) ou **enregistrements**, et les **colonnes** sont associées à des **attributs**.



Un **Type-entité** définit la caractéristique de la relation, dans notre cas ce sont des « ELEVES ». Une **entité** est une occurrence de son type-entité. A chaque enregistrement d'une relation correspond donc une entité du type-entité, ici nos 3 élèves Matthieu, Nicolas et Kevin.

Le **domaine** d'un attribut, fini ou infini, est l'ensemble des valeurs qu'il peut prendre. Le domaine de l'attribut « Age » est l'ensemble des entiers de 0 à 130 (actuellement) par exemple. Le domaine de « Nom » est une chaine de caractères de taille à priori non connu. Le domaine de « classe » est ambigu. Si on sait qu'il n'existe aucune autre classe que « MPSI », « PCSI », alors le domaine est « MPSI » et « PCSI ». Sinon, c'est une chaine de caractères de taille à priori non définie…

Le **degré** d'une relation est son nombre d'attributs, 5 dans notre exemple (id, Prenom, Nom, Classe, Age).

Un **schéma de relation** précise le nom d'une relation ainsi que la liste des attributs avec leur domaine : Eleves(id : Entier, Prenom : Chaîne , Nom : Chaîne , Classe : Chaîne , Age : Entier). On appelle **Schéma relationnel** l'ensemble des schémas de relation. On aura évidemment un attribut en commun entre chacune des relations, reliant toutes les données entre elles.

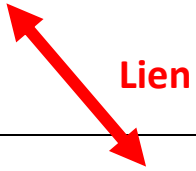
On appelle « **clé primaire** » une ou plusieurs données permettent d'identifier de manière unique un enregistrement dans une table. On ne peut avoir deux fois le même enregistrement, il existe donc toujours une clé primaire. On la souligne dans le schéma de relation: Eleves(id : Entier, Prenom : Chaîne , Nom : Chaîne , Classe : Chaîne , Age : Entier). On ajoute souvent un identifiant « id » en première colonne, entier unique qui permet d'être clé primaire. Dans une relation contenant des numéros de téléphone de personnes différentes, l'attribut « Numéro » pourrait être la clé primaire.

Une **base de données relationnelle** est constituée par l'ensemble des n-uplets des différentes relations du schéma relationnel.

1.III. Exemple support du cours

Soient les deux tables suivantes :

Eleves					
id	**Prenom**	**Nom**	**Classe**	**Age**	**Ville**
1	Steph	ANE	MPSI	17	Paris
2	Marc	IMBUT	PCSI	18	Marseille
3	Jo	NID	MPSI	19	Montpelier
4	Rayan	AIR	MP	19	Paris
5	Tom	DESAVOIE	PCSI	18	Nice
6	Jerry	CANE	PSI	19	Paris
7	Paul	AINE	MP	19	Marseille
8	Jean	NAIMAR	PCSI	18	Toulouse
9	Jack	OUILLE	MPSI	18	Montpelier
10	Sam	OURAIL	PC	20	Paris



Profs					
id	**Titre**	**Prenom**	**Nom**	**Classe**	**Salle**
1	Mr	Denis	DEFAUCHY	MPSI	D21
2	Mr	Paul	YMERE	PCSI	A05
3	Mme	Sylvie	DEFOU	MP	B18
4	Mr	Georges	AITTE	PC	C15
5	Mme	Marie	AGE	PSI	E8

La base de données relationnelle est donc :

- Eleves(id : Entier, Prenom : Chaîne , Nom : Chaîne , Classe : Chaîne , Age : Entier,Ville : Chaîne)
- Profs(id : Entier, Titre : Chaîne, Prenom : Chaîne , Nom : Chaîne , Classe : Chaîne , Salle : Chaîne)

Voici le lien vers la base de données en « .db » pour travailler dessus si cela vous intéresse : LIEN

Définissons une **clé étrangère**. C'un attribut (ou plusieurs) d'une table qui sert de clé primaire d'une autre table. La clé étrangère permet donc de lier les deux tables. Dans l'exemple ci-dessus, considérant que la table Profs a bien une unique classe par ligne, l'attribut Classe de la table Eleves est une clé étrangère. On dit que Eleves.Classe référence la clé primaire Profs.Classe.

1.IV. Généralités

Nous serons ici amenés à détailler le langage SQL (Structured Query Language) pour interroger les bases de données, c'est-à-dire pour réaliser des **Requêtes** sur une base de données.

1.IV.1 Manipulations et requêtes

Les bases de données se trouvent sous différentes extensions (csv, sql, xml, xls, bd…). On en trouve par exemple beaucoup concernant la France ici : https://www.data.gouv.fr/fr/

La gestion des bases de données est totalement indépendante du logiciel Python. En effet, le langage SQL peut être directement utilisé dans des logiciels comme « Sqlite browser » gratuit et disponible ici : http://sqlitebrowser.org/

Je vous recommande d'importer un fichier .csv en faisant :

- Création d'une nouvelle bdd et enregistrement
- Ne pas remplir de tables (Cancel)
- Fichier – Importer – Table vers un fichier csv – Afficher toutes les extensions – Ouvrir le fichier
- Choisir les bonnes options d'importation en visualisant le résultat

Ecriture des requêtes dans l'onglet « Exécuter le SQL »

Nous allons toutefois voir dans ce cours comment traiter les bases de données sous Python, même s'il est possible d'en être totalement indépendant.

1.IV.2 Généralités de langage

Les instructions SQL dans un code doivent impérativement être écrites en lettres majuscules et se finir par un point-virgule « ; ».

Quelques symboles peuvent vous être utiles pour effectuer des recherches particulières :

- « * » permet de dire que l'on recherche tous les attributs
- « % » désigne une chaîne de caractères quelconque – « A% » désignera un mot commençant par A – « %a% » un mot contenant a…
- « _ » désigne un caractère quelconque
- « '…' » désigne un champ au format texte

1.V. Requêtes SQL au programme

1.V.1.a Intérêt de l'utilisation du SQL

Supposons que l'on veuille récupérer pour chaque élève de la relation « Eleves » le nom de son prof.

Il est parfaitement possible de travailler avec des listes, en programmant des boucles for qui vont, pour chaque élève, trouver sa classe, puis chercher dans la relation « Profs » la classe afin d'attribuer le nom de l'élève. Supposons que les tables aient les noms Liste_Eleves et Liste_Profs :

```
Liste_Resultat = []
for Eleve in Liste_Eleves:
    Prenom_Eleve = Eleve[1]
    Nom_Eleve = Eleve[2]
    Classe_Eleve = Eleve[3]
    for Prof in Liste_Profs:
        Nom_Prof = Prof[3]
        Classe_Prof = Prof[4]
        if Classe_Eleve == Classe_Prof:
            Liste_Resultat.append([Prenom_Eleve,Nom_Eleve,Nom_Prof])
print(Liste_Resultat)
```

Cet algorithme est préprogrammé, de manière optimisée, dans le langage SQL ☺

Il suffit d'écrire :

```
SELECT Eleves.Prenom, Eleves.Nom, Profs.Nom FROM Eleves JOIN Profs ON
Eleves.Classe = Profs.Classe ;
```

Voilà tout l'intérêt d'utiliser ce langage, tout est déjà prêt. Il faudra simplement apprendre les requêtes.

1.V.1.b Opérateurs de base

1.V.1.b.i Projection : SELECT FROM

- **Principe**

La projection permet de n'afficher qu'une partie des attributs (colonnes) d'une relation.

Sur une table :

Objectif	Requête
Obtenir toutes les informations d'un tableau	`SELECT * FROM Eleves ;` *Toutes les informations concernant les élèves*
Sélectionner certains attributs dans un tableau	`SELECT Nom,Prenom FROM Profs ;` *Noms et prénons des profs*
Obtenir toutes les occurrences d'un attribut en supprimant les doublons	`SELECT DISTINCT Age FROM Eleves ;` *Tous les âges des élèves*

- **Options**

Renommer une table : On peut renommer une table avec « **AS** » afin de simplifier son utilisation ensuite, par exemple : `SELECT Nom,Prenom FROM Eleves as E WHERE E.Age = 18`

Classement par ordre alphabétique, numérique, dates… : « **ORDER BY Att** ». Classement par ordre inverse : ORDER BY Att DESC (ou ASC pour croissant, mais c'est déjà ce que fait ORDER BY). Classement sur plusieurs attributs (ex : ORDER BY Att1 DESC,Att2 DESC), pour par exemple, avoir une liste triée par ordre décroissant de dates, puis d'heures pour les mêmes dates

Limiter le nombre de résultats : Si l'on ne souhaite que récupérer les premiers résultats d'une requête, on ajoute **LIMIT i**, i étant le nombre de résultats à garder. Par exemple : `SELECT Nom,Prenom FROM Eleves ORDER BY Age DESC LIMIT 1` renverra le premier élève dans l'ordre alphabétique décroissant (élève le plus âgé). Une autre version existe selon les logiciels : `FETCH FIRST 1 ROWS ONLY`. En complément de LIMIT, on peut ajouter **OFFSET i** afin de ne pas récupérer les i premiers résultats renvoyés par le LIMIT : `SELECT Nom,Prenom FROM Eleves ORDER BY Age DESC LIMIT 2 OFFSET 1` renvoie les 2° et 3° les plus âgés.

- **Remarques**

On peut réaliser une jointure sur plusieurs tables : `SELECT * FROM Eleves,Profs` ou `SELECT * FROM Eleves JOIN Profs`. Le résultat de cette requête est un tableau peu utile en soi. Chaque élève est répété autant de fois qu'il y a de profs, et devant chaque ligne de cet élève est mise l'une des lignes du tableau prof. La dimension de ce tableau est donc Nombre élèves X Nombre profs et les données qui y sont inscrites n'ont pas beaucoup de sens à la lecture humaine… Associée à une restriction « ON » qui sera vue plus tard dans ce cours, elle prendra beaucoup de sens.

Lorsque l'on souhaite sélectionner plusieurs colonnes dans le but de trouver des couples d'attributs distincts, DISTINCT ne fonctionne pas sur un SELECT Distinct (Att1,Att2), on devra utiliser un GROUP BY vu plus tard dans ce cours.

1.V.1.b.ii Sélection (ou restriction) : WHERE

La sélection permet de n'afficher qu'une partie des enregistrement (lignes) d'une relation. On utilise alors le mot **WHERE**.

Les opérateurs de sélection utilisables sont :

- « = », « == » ou « != » utilisables avec tous types de données
- « > », « < », « >= », « <= » utilisables uniquement avec des données numériques
- LIKE, BETWEEN, IN, AND, OR, NOT

Exemples :

Objectif	Requête

Récupérer les attributs des enregistrements possédant un attribut spécifique	`SELECT Nom,Prenom FROM Eleves WHERE…`
	`- Classe = "MPSI" ;`
	`- Classe LIKE "MPSI" ;`
	Noms et prénoms des élèves de MPSI

	`SELECT Nom,Prenom FROM Eleves WHERE…`
	`- Classe != "MPSI" ;`
	`- NOT Classe = "MPSI" ;`
	`- Classe NOT LIKE "MPSI" ;`
	Noms et prénoms des élèves pas en MPSI

	`SELECT DISTINCT Age FROM Eleves WHERE Classe = "MPSI" OR Classe = "PCSI" ;` *Tous les âges des élèves de première année sans doublons*
	`SELECT Nom FROM Eleves WHERE Age BETWEEN 18 AND 19 ;` *Tous les noms des élèves ayant de 18 à 19 ans*

1.V.1.b.iii Jointure : … JOIN ON …

- **Principe**

La jointure consiste à trouver des données en lien avec plusieurs tables. Par exemple :

- « Dans quelle salle doivent aller les élèves pour voir leur prof ? ».
- « Dans quelle salle doivent aller les élèves de MPSI pour voir leur prof ? ».

Il faut identifier les élèves de la classe MPSI dans la table « Eleves », chercher le prof de la classe MPSI dans la table « Profs », et associer la salle aux élèves concernés.

Le programme d'IPT propose de ne se limiter qu'aux jointures simples utilisant Join…ON…=….

Voici comment écrire les requêtes proposées :

```
SELECT Eleves.Prenom, Eleves.Nom, Profs.Salle FROM Eleves JOIN Profs ON
Eleves.Classe = Profs.Classe ;
SELECT DISTINCT Profs.Salle FROM Eleves JOIN Profs ON Eleves.Classe =
Profs.Classe WHERE Eleves.Classe = "MPSI" ;
```

Comme il faut préciser ce que l'on sélectionne parmi les deux relations, on précise les attributs en ajoutant le nom de la table et un point avant chacun d'entre eux.

Quelques explications :

La jointure **(Profs JOIN Eleves ON)** crée une nouvelle table de la basée où pour chaque entrée de la table Eleves sont associés chacun des profs de la table PROFS. Les attributs sont renommés en fonction de leur table de provenance (Eleves.Nom, Profs.Nom etc) sauf s'ils sont différents. Elle possède donc 10*5=50 lignes, sous la forme suivante :

Profs						Eleves					
Profs.id	Profs.Titre	Profs.Prenom	Profs.Nom	Profs.Classe	Profs.Salle	Eleves.id	Eleves.Prenom	Eleves.Nom	Eleves.Classe	Eleves.Age	Eleves.Ville
1	Mr	Denis	DEFAUCHY	MPSI	D21	1	Steph	ANE	MPSI	17	Paris
1	Mr	Denis	DEFAUCHY	MPSI	D21	2	Marc	IMBUT	PCSI	18	Marseille
1	Mr	Denis	DEFAUCHY	MPSI	D21	3	Jo	NID	MPSI	19	Montpelier
1	Mr	Denis	DEFAUCHY	MPSI	D21	4	Rayan	AIR	MP	19	Paris
1	Mr	Denis	DEFAUCHY	MPSI	D21	5	Tom	DESAVOIE	PCSI	18	Nice
1	Mr	Denis	DEFAUCHY	MPSI	D21	6	Jerry	CANE	PSI	19	Paris
1	Mr	Denis	DEFAUCHY	MPSI	D21	7	Paul	AINE	MP	19	Marseille
1	Mr	Denis	DEFAUCHY	MPSI	D21	8	Jean	NAIMAR	PCSI	18	Toulouse
1	Mr	Denis	DEFAUCHY	MPSI	D21	9	Jack	OUILLE	MPSI	18	Montpelier
1	Mr	Denis	DEFAUCHY	MPSI	D21	10	Sam	OURAIL	PC	20	Paris
2	Mr	Paul	YMERE	PCSI	A05	1	Steph	ANE	MPSI	17	Paris
…	…	…	…	…	…	…	…	…	…	…	…

La jointure **(Profs JOIN Eleves ON Eleves.Classe = Profs.Classe)** sélectionne dans cette table les lignes où la classe du prof et la classe de l'élève sont identiques.

Profs						Eleves					
id	**Titre**	**Prenom**	**Nom**	**Classe**	**Salle**	**id**	**Prenom**	**Nom**	**Classe**	**Age**	**Ville**
1	Mr	Denis	DEFAUCHY	MPSI	D21	1	Steph	ANE	MPSI	17	Paris
1	Mr	Denis	DEFAUCHY	MPSI	D21	3	Jo	NID	MPSI	19	Montpelier
1	Mr	Denis	DEFAUCHY	MPSI	D21	9	Jack	OUILLE	MPSI	18	Montpelier
2	Mr	Paul	YMERE	PCSI	A05	2	Marc	IMBUT	PCSI	18	Marseille
2	Mr	Paul	YMERE	PCSI	A05	5	Tom	DESAVOIE	PCSI	18	Nice
2	Mr	Paul	YMERE	PCSI	A05	8	Jean	NAIMAR	PCSI	18	Toulouse
3	Mme	Sylvie	DEFOU	MP	B18	4	Rayan	AIR	MP	19	Paris
3	Mme	Sylvie	DEFOU	MP	B18	7	Paul	AINE	MP	19	Marseille
4	Mr	Georges	AITTE	PC	C15	10	Sam	OURAIL	PC	20	Paris
5	Mme	Marie	AGE	PSI	E8	6	Jerry	CANE	PSI	19	Paris

Il reste alors comme d'habitude à écrire : SELECT … FROM **(table de jointure)** WHERE …

- **Remarques**

Le programme d'informatique se limite aux jointures naturelles/équi-jointures, c'est-à-dire se basant sur un critère d'égalité « ON T1.Att = T2.Att », ce que nous avons donc vu dans ce paragraphe.

ON et WHERE fonctionnent indépendamment l'un de l'autre, mais préférer ON pour la condition de jointure créant une table « logique », puis WHERE pour restreindre les résultats

Vous remarquerez que rien ne change si on intervertit les 2 tables droite/gauche. Ce qui peut changer si l'instruction est écrite selon Eleves JOIN Profs ou Profs JOIN Eleves, c'est l'ordre de l'organisation des classes, mais les requêtes sur cette table seront inchangées.

Ecrire **Eleves JOIN Profs** sans condition ON revient à créer une table dans laquelle pour chaque n-uplet d'une des tables initiales, on associe tous les n-uplets de l'autre table, ce qui revient à faire un produit cartésien CROSS JOIN que nous verrons dans un prochain paragraphe

Il est parfaitement possible de réaliser une jointure sur plus de 2 tables, par exemple : SELECT * FROM Table1 JOIN Table2 JOIN Table3 ON Table1.Att = Table2.Att AND Table1.Att = Table3.Att. Exemple :

Passages

id_passage	badge	zone	date
Filtre	Filtre	Filtre	Filtre
1	3	2	2022-04-06
2	1	1	2022-04-06
3	4	3	2022-04-07
4	2	1	2022-04-08
5	2	4	2022-04-08
6	3	3	2022-04-08
7	1	2	2022-04-09
8	4	4	2022-04-12
9	1	1	2022-04-12
10	3	1	2022-04-12

Personnes

id_personne	nom	prenom	badge
Filtre	Filtre	Filtre	Filtre
1	Ette	George	1
2	Ant	Laure	2
3	Belle	Isa	3
4	Pold	Léo	4

Zones

id_zone	type
Filtre	Filtre
1	Batiment D
2	Salle polyvalente
3	Cantine
4	Bibliothèque

Résultat de la requête

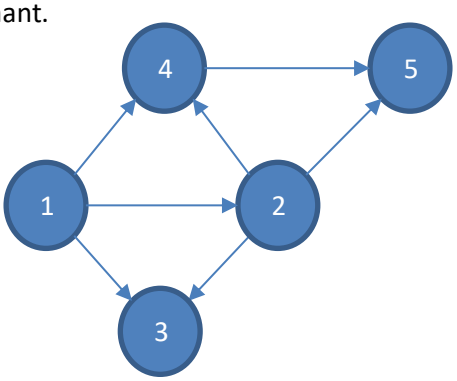
Belle	Isa	Salle polyvalente
Ette	George	Batiment D
Pold	Léo	Cantine
Ant	Laure	Batiment D
Ant	Laure	Bibliothèque
Belle	Isa	Cantine
Ette	George	Salle polyvalente
Pold	Léo	Bibliothèque
Ette	George	Batiment D
Belle	Isa	Batiment D

SELECT Pe.nom,Pe.prenom,Z.type FROM Passages as Pa JOIN Personnes as Pe JOIN Zones as Z ON Pa.badge = Pe.badge AND Pa.zone = Z.id_zone

On renvoie la liste de (noms/prénoms/type de zone) de tous les passages

Il est possible de réaliser une « **auto-jointure** » en faisant une jointure sur une même table « Table JOIN Table ». Attention, pour que l'auto-jointure soit possible, il faut distinguer les deux tables, par exemple en les renommant.

Soit le graphe suivant :



id	depart	arrivee
Filtre	Filtre	Filtre
1	1	2
2	1	3
3	1	4
4	2	3
5	2	4
6	4	5
7	2	5

La requête suivant renvoie tous les chemins à 2 tronçons dans le graphe :

```
SELECT T1.depart,T2.arrivee FROM Chemins as T1 JOIN Chemins as T2 ON T1.arrivee = T2.depart
```

depart	arrivee
1	3
1	4
1	5
1	5
2	5

La suivante permet de trouver tous les départs permettant d'arriver à 5 par 2 chemins successifs :

à 2 tronçons arrivant à 4 :

```
SELECT T1.depart FROM Chemins as T1 JOIN Chemins as T2 ON T1.arrivee = T2.depart WHERE T2.arrivee = 5
```

depart
1
2
1

On a donc ceux chemins partant de 1 et un partant de 2 pour arriver à 5 en 2 tronçons 😊

1.V.1.c Opérateurs ensemblistes

1.V.1.c.i Union – Intersection - Différence

Les trois opérateurs usuels en bases de données sont l'union, l'intersection et la différence. Ces opérateurs s'utilisent entre deux tables (relations) de même structure.

Le principe de ces opérateurs consiste à ajouter un mot (UNION, EXCEPT, INTERSECT) entre deux requêtes SELECT.

Exemple : Soient les deux tables suivantes contenant la référence et le prix d'objets d'un magasin : BDD

T1	
Ref	**Prix**
A27	12
C32	17
E15	18
Z7	20

T2	
Ref	**Prix**
C32	17
A27	15
B20	20
A12	21

Obtention d'une table de l'ensemble des produits Rq : UNION ne garde pas les doublons	`SELECT * FROM T1 UNION SELECT * FROM T2 ;` `[('A12', 21), ('A27', 12), ('A27', 15), ('B20', 20), ('C32', 17), ('E15', 18), ('Z7', 20)]`
Enlever les produits de T1 présents dans T2 (même ref & prix)	`SELECT * FROM T1 EXCEPT SELECT * FROM T2 ;` `[('A27', 12), ('E15', 18), ('Z7', 20)]`
Trouver les produits présents dans les deux tables (même ref & prix)	`SELECT * FROM T1 INTERSECT SELECT * FROM T2 ;` `[('C32', 17)]`

Remarques :

- On pourra bien-sûr ajouter des restrictions dans chaque sous-requête (WHERE) pour affiner les recherches.
- **EXCEPT renvoie des résultats distincts et ces résultats font forcément partie de l'attribut étudié**. Par exemple, cherchons dans la liste des classes des élèves, les classes que M. DEFAUCHY n'a pas : `SELECT Classe FROM Eleves EXCEPT SELECT Classe FROM Profs WHERE Nom = "DEFAUCHY"` – Le résultat est une **liste distincte de classes** (MP, PC, PCSI et PSI)

1.V.1.c.ii IN et NOT IN

Il existe la fonction « WHERE … NOT IN () » qui permet deux choses en plus que EXCEPT.

Enlever les produits de T1 dont le prix est dans T2	`SELECT Ref FROM T1 WHERE Prix NOT IN (SELECT Prix FROM T2) ;` Résultat : A27 C32 E15
Récupérer les produits de T1 dont le prix est dans T2	`SELECT Ref FROM T1 WHERE Prix IN (SELECT Prix FROM T2) ;` Résultat : C32 Z7

Reprenons l'exemple des classes. Avec EXCEPT, on écrivait :

```
SELECT Classe FROM Eleves EXCEPT SELECT Classe FROM Profs WHERE Nom =
"DEFAUCHY"
```

On peut obtenir le même résultat ainsi :

```
SELECT DISTINCT Classe FROM Eleves WHERE Classe NOT IN (SELECT Classe
FROM Profs WHERE Nom = "DEFAUCHY")
```

La fonction NOT IN présente deux grandes différences avec EXCEPT :

- **Ne pas renvoyer de résultats distincts** :
  ```
  SELECT Classe FROM Eleves WHERE Classe NOT IN (SELECT Classe FROM
  Profs WHERE Nom = "DEFAUCHY"
  ```
  renvoie PCSI, MP, PCSI, PSI, MP, PCSI, PC
- **Pouvoir sélectionner un autre attribut que celui qui trie,** autrement dit ne pas devoir enlever le même attribut (SELECT **Classe** … EXCEPT SELECT **Classe**) renvoyé par les deux requêtes. Par exemple, récupérer les noms des élèves de M. DEFAUCHY :
  ```
  SELECT Nom FROM Eleves WHERE Classe NOT IN (SELECT Classe FROM Profs
  WHERE Nom != "DEFAUCHY")
  ```

1.V.1.d Fonctions d'agrégation

Voyons dans le tableau suivant quelques exemples de ce que l'on peut faire :

Minimum MIN()	Age min des élèves : `SELECT MIN(Age) FROM Eleves ;`
Maximum MAX()	Age max des élèves : `SELECT MAX(Age) FROM Eleves ;`
Somme SUM()	Somme des âges des élèves : `SELECT SUM(Age) FROM Eleves ;`
Moyenne AVG()	Moyenne des âges des élèves : `SELECT AVG(Age) FROM Eleves ;`

Comptage
COUNT
	Nombre d'élèves :
	`SELECT COUNT(*) FROM Eleves ;`
	Nombre délèves de 17 ans :
	`SELECT COUNT(*) FROM Eleves WHERE Age = 17 ;`

Il est possible d'obtenir un résultat restreint à l'aide d'une fonction d'agrégation. Voici quelques exemples :

- Récupérer le nom et l'âge de l'élève le plus jeune lorsqu'il est seul
  `SELECT Nom,MIN(Age) FROM Eleves`
  Le résultat ne sera pas une colonne des noms avec partout l'âge mini, mais l'un des noms dont l'âge est minimum s'il y en a plusieurs avec l'âge mini associé.
- Récupérer tous les noms et âges des élèves ayant l'âge minimum :
  `SELECT Nom,Age FROM Eleves WHERE Age = (SELECT MIN(Age) FROM Eleves)`

1.V.1.d.i Produit cartésien : … CROSS JOIN …

Soient les deux tables suivantes contenant les groupes sanguins d'une population de 4 mâles et 4 femelles d'un laboratoire :

MALES	
Groupe	**Resus**
A	-
AB	+
B	+
B	-

FEMELLES	
Groupe	**Resus**
B	-
B	-
AB	+
A	-

On souhaite créer une base de données de tous les mixages possibles afin d'étudier les groupes possibles des enfants issus de tous les croisements possibles.

Il suffit d'utiliser l'opérateur CROSS JOIN :

```
SELECT * FROM MALES CROSS JOIN FEMELLES ;
```

Voici les premières lignes du résultat obtenu :

A	-	B	-
A	-	B	-
A	-	AB	+
A	-	A	-
AB	+	B	-
AB	+	B	-
…	…	…	…

Remarque : comme vu pour la jointure, on obtient le même résultat avec :

```
SELECT * FROM MALES JOIN FEMELLES ;
```

La jointure permettant d'ajouter la condition ON.

1.V.1.d.ii Division cartésienne : GROUP BY … HAVING …

- **Exemple 1**

Soient les deux tables suivantes (BDD) désignant pour l'une l'ensemble des lâchés machines des pilotes du club (Nom pilote, Identification Avion) et l'autre regroupant l'ensemble des avions du club :

Lachers	
Nom	**Avion**
DEFAUCHY	XH
SMITH	RL
DEFAUCHY	RL
DEFAUCHY	PQ
OMAR	XH
OLAF	PQ
SMITH	PQ
SMITH	PQ

Avions	
Avion	**Type**
XH	DR420
RL	DR480
PQ	Sportstar

Cela n'a pas beaucoup de sens, mais SMITH a été 2 fois enregistré pour PQ (ça sera utile pour l'exemple).

Voyons quelques exemples de requêtes utilisant GROUP BY.

Trouvons l'ensemble des noms des pilotes de la table « Lachers » :

```
SELECT Nom FROM Lachers GROUP BY Nom ;
```

Cela revient au même qu'écrire « `SELECT DISTINCT(Nom) FROM Lachers` » à l'ordre des résultats près. **Toutefois, des choses restent « cachées » derrière ce résultat. En effet, derrière chaque nom, il est possible d'aller chercher des informations sur les attributs qui lui sont rattachés…**

Ainsi, on souhaite maintenant obtenir la liste des pilotes ainsi que le nombre d'avions sur lesquels ils sont lâchés :

```
SELECT Nom,COUNT(*) FROM Lachers GROUP BY Nom ;
```

Nom	COUNT(*)
DEFAUCHY	3
OLAF	1
OMAR	1
SMITH	3

On arrive à récupérer, derrière chaque regroupement, le nombre de lignes qui lui sont rattachées. Tout se passe comme si une table était créée pour chaque regroupement, sur laquelle on va pouvoir réaliser des opérations, spécifiques à chacun d'entre eux.

Ainsi, puisque ce qui nous intéresse est le nombre d'avions différents sur lesquels ils sont lâchés, on écrit :

```
SELECT Nom,COUNT(DISTINCT(Avion)) FROM Lachers GROUP BY Nom ;
```

Nom	COUNT(DISTINCT(Avion))
DEFAUCHY	3
OLAF	1
OMAR	1
SMITH	2

Il va maintenant être possible d'ajouter une condition à cette sélection en écrivant : `HAVING` ...

On cherche finalement les noms des pilotes étant lâchés sur tous les avions.

```
SELECT Nom FROM Lachers GROUP BY Nom HAVING COUNT(DISTINCT(Avion)) =
(SELECT COUNT(*) FROM Avions) ;
```

On cherche dans la table « Lachers » regroupée par noms de pilotes ceux qui ont un nombre distinct d'avions égal au nombre d'avions de la table « Avions ».

Que se passe-t-il ? Lorsque nous essayons de répondre à cette question sans ordinateur, on compte le nombre d'avions `(SELECT COUNT(*) FROM Avions)` qui vaut 3 dans l'exemple. On regroupe ensuite les pilotes en le nombre d'avions distincts sur lesquels ils sont lâchés :

SELECT Nom FROM Lachers GROUP BY Nom	HAVING COUNT(DISTINCT(Avion))	(SELECT COUNT(*) FROM Avions)
DEFAUCHY	3	3
SMITH	2	3
OMAR	1	3
OLAF	1	3

Le résultat est donc DEFAUCHY.

Notez bien que la table `SELECT Nom FROM Lachers GROUP BY Nom` est une table contenant autant de lignes que de noms, sachant que pour chaque nom, c'est l'une des lignes associées à ce Nom qui est prise dans la table initiale. On ne sait pas a priori laquelle sera récupérée. Et le résultat final renvoyé sera l'une des lignes de ce tableau. Si on demande `SELECT *` :

```
SELECT * FROM Lachers GROUP BY Nom HAVING
COUNT(DISTINCT(Avion)) = (SELECT COUNT(*) FROM Avions) ;
```

Nom	Avion
DEFAUCHY	PQ

Attention : HAVING nécessite un GROUP BY. Il ne remplace donc pas un WHERE et va permettre de faire une sélection dans les regroupements obtenus par le GROUP BY.

- **Exemple 2**

Vous avez donc maintenant compris que derrière chaque regroupement, des informations sont stockées. Voici donc un nouvel exemple (BDD) faisant appel à des fonctions d'agrégations :

Achats

id	Nom	Dépense
	...	Filtre
5	Nicolas	17
4	Thierry	15
3	Denis	4
2	Nicolas	102
1	Denis	50

Somme et moyenne des dépenses de chacun	`SELECT Nom,SUM(Dépense),AVG(Dépense) FROM Achats GROUP BY Nom`

Nom	SUM(Dépense)	AVG(Dépense)
Denis	54	27.0
Nicolas	119	59.5
Thierry	15	15.0

- **Exemple 3**

Il est possible d'utiliser GROUP BY sur plusieurs attributs afin de différencier, par exemple, différentes personnes ayant le même nom de famille :

```
SELECT nom,prenom,COUNT(*) FROM Activité
GROUP BY nom,prenom
```

nom	prenom	COUNT(*)
DEFAUCHY	Chloé	2
DEFAUCHY	Denis	2
DEFAUCHY	Léa	3
DEFAUCHY	Virgine	1
DUBOIS	Antoine	1
DUBOIS	Hugo	2
DUBOIS	Maxime	1
DUBOIS	Victor	1

id	nom	prenom	inscription	annee
Filtre	Filtre	Filtre	Filtre	Filtre
1	DEFAUCHY	Denis	Avion	2021
2	DEFAUCHY	Léa	Danse	2019
3	DEFAUCHY	Denis	ULM	2021
4	DEFAUCHY	Léa	Karaté	2022
5	DEFAUCHY	Chloé	Gym	2021
6	DEFAUCHY	Léa	Gym	2021
7	DEFAUCHY	Virgine	Vélo	2020
8	DUBOIS	Hugo	Vélo	2020
9	DUBOIS	Maxime	Gym	2020
10	DUBOIS	Hugo	Foot	2021
11	DUBOIS	Victor	Karaté	2020
12	DUBOIS	Antoine	Vélo	2019
13	DEFAUCHY	Chloé	Gym	2022

On obtient bien le nombre d'inscriptions de chaque personne de la table initiale.

Allons plus loin :

```
SELECT nom,prenom,COUNT(DISTiNCT(inscription)) FROM Activité GROUP BY
nom,prenom
```

nom	prenom	COUNT(DISTiNCT(inscription))
DEFAUCHY	Chloé	1
DEFAUCHY	Denis	2
DEFAUCHY	Léa	3
DEFAUCHY	Virgine	1
DUBOIS	Antoine	1
DUBOIS	Hugo	2
DUBOIS	Maxime	1
DUBOIS	Victor	1

On obtient le nombre d'inscriptions à un sport **différent** pour chaque personne.

1.VI. Cardinalité : les relations 1-1, 1-*, *-*

Pour illustrer ce paragraphe, reprenons l'exemple utilisé précédemment des 3 tables suivantes :

Passages

id_passage	badge	zone	date
Filtre	Filtre	Filtre	Filtre
1	3	2	2022-04-06
2	1	1	2022-04-06
3	4	3	2022-04-07
4	2	1	2022-04-08
5	2	4	2022-04-08
6	3	3	2022-04-08
7	1	2	2022-04-09
8	4	4	2022-04-12
9	1	1	2022-04-12
10	3	1	2022-04-12

Personnes

id_personne	nom	prenom	badge
Filtre	Filtre	Filtre	Filtre
1	Ette	George	1
2	Ant	Laure	2
3	Belle	Isa	3
4	Pold	Léo	4

Zones

id_zone	type
Filtre	Filtre
1	Batiment D
2	Salle polyvalente
3	Cantine
4	Bibliothèque

1.VI.1 Vocabulaire

Une **entité** est une chose qui existe et est distinguable des autres entités. Les personnes, les zones et les passages sont des entités.

L'occurrence d'une entité est un élément particulier correspondant à l'entité. Chaque entité possède des propriétés (ou attributs) qui la décrivent. Un passage possède ses propres attributs (id, badge, zone, date). Chaque attribut est associé à un domaine de valeur.

Une **occurrence** possède des valeurs pour chacun de ses attributs dans le domaine correspondant.

Une **association** représente un lien quelconque entre différentes entités, par exemple le lien entre un passage et une personne ou une zone. On parle **d'association binaire** lorsque seules deux entités sont concernées par l'association (chaque passage est associé à une zone, chaque passage est associé à une personne).

1.VI.2 Cardinalités

Les cardinalités sont des couples de valeur MIN-MAX (ex : 1-1, 1-*, *-1 citées au programme) que l'on trouve entre chaque entité et ses associations liées.

Dans notre exemple, supposons que pour être présente dans les passages, une personne doit en avoir au moins réalisé un. On représente les choses ainsi :

Voilà comment interpréter les cardinalités proposées :

- Une personne réalise de 1 à plusieurs passages (1-*). On retrouve donc plusieurs fois la même personne dans la table Passages.
- Un passage est réalisé par une et une seule personne (1-1). Chaque passage de la table Passage n'a donc qu'un seul attribut badge.
- Un passage se fait dans une à plusieurs zones (1-*). On retrouve donc plusieurs fois le même badge dans la table Passages.
- Une zone reçoit de 0 à plusieurs passages (0-*). On retrouve donc de 0 à plusieurs fois la même zone dans la table Passages.

L'utilisation d'une clé étrangère permet de représenter une relation 1-* (ou 0-*) entre deux tables. Par exemple :

A chaque personne, on associe un à plusieurs passages 1-* A chaque passage, on associe une seule personne 1-1	SELECT * FROM Passages JOIN Personnes ON Passages.badge = Personnes.badge *Passages.badge : Clé étrangère* *Personnes.badge : Clé primaire*
A chaque passage, on associe une à plusieurs zones 1-* A chaque zone, on associe 0 à plusieurs passages 0-*	SELECT * FROM Passages JOIN Zones ON Passages.zone = Zones.id_zone *Passages.zone : Clé étrangère* *Personnes.badge : Clé primaire*

Prenons maintenant un exemple de relation*-*. On souhaite créer un historique des zones visitées par chaque personne. Une personne peut visiter plusieurs zones, une zone peut avoir été visitée par plusieurs personnes. C'est donc une relation *-*. Pour représenter cette relation, on utilise deux associations 0-* et une table (la table Passages) appelée table d'association ou table de liaison. Cette table est constituée de deux clés étrangères Passages.badge (clé primaire de Personnes) et Passages.zone (clé primaire de Zones). On a donc :

On pourra obtenir la relation en réalisant la jointure suivante :

SELECT Pa.zone,Pe.nom,PE.prenom FROM Passages as Pa JOIN Zones as Z JOIN Personnes as Pe ON Pa.zone = Z.id_zone AND Pa.badge = Pe.badge

1.VII. Gestion des BDD sous Python

Attention, on attend de vous de connaître les requêtes. Ce paragraphe vous sera utile pour manipuler les BDD sous Python, mais n'a pas à être connu par cœur.

1.VII.1 Structure du code

La gestion d'une base de données sous Python s'organise ainsi :

```
import sqlite3
BDD = sqlite3.connect('BDD.db')
cursor = BDD.cursor()
cursor.execute("""…""")
BDD.commit()
BDD.close()
```

C'est normal qu'à ce stade, vous ne compreniez pas grand-chose à ce code ! En quelques mots, et nous développerons tout cela dans les prochains paragraphes :

- On importe la librairie de bases de données SQL sous Python
- On ouvre ou crée la base de données BDD.db
- On crée un outil « cursor » qui va manipuler la base de données
- On exécute alors différentes requêtes, et les « … » sont des requêtes que vous apprendrez à manipuler par la suite
- On applique les modifications au fichier BDD.db
- On ferme la base de données

Remarque : Dans `cursor.execute("""…""")`, il faut un texte sous forme de string, on peut donc utiliser `",'` ou `'''` ou `"""`. J'ai choisi ici les `"""` (équivalent à `'''`) pour les raisons suivantes :

- Pas de problèmes pour utiliser une seule Guillement pour définir un texte dans le texte : `"""("A","B")"""`
- La couleur du texte sous Python est bleue et permet d'identifier les codes SQL facilement

1.VII.2 Premiers pas – Création - Ouverture

1.VII.2.a Librairie sous Python

Nous allons manipuler les bases de données sous Python avec la librairie sqlite3. Il faut donc écrire :

```
import sqlite3
```

1.VII.2.b Création/ouverture d'une base de données

Que la base de données existe ou non, une commande permet de l'ouvrir ou de la créer. Il faut écrire :

```
BDD = sqlite3.connect('BDD.db')
```

Où `'BDD.db'` est la base de données à ouvrir ou à créer, dans le répertoire de travail (on peut préciser un chemin absolu) comme pour les fichiers textes.

Concrètement, on demande à Python de se « connecter » à la base de données `'BDD.db'` qu'il ouvre ou crée si elle n'existe pas.

1.VII.2.c Création de l'outil de manipulation de la base de données

Pour manipuler la base de données BDD, il est nécessaire de créer un outil sous Python qui va permettre de travailler dessus. Il faut écrire :

```
cursor = BDD.cursor()
```

Ne l'oubliez pas !

1.VII.2.d Création/suppression de relations/tables

1.VII.2.d.i Création d'une relation

Nous pouvons maintenant créer des relations à l'aide d'une requête « CREATE TABLE ». Créons les deux tables de l'exemple proposé précédemment :

```
cursor.execute("""CREATE TABLE Eleves(id INTEGER PRIMARY KEY,Prenom
TEXT,Nom TEXT,Classe TEXT,Age INTERGER,Ville TEXT) ;""")
cursor.execute("""CREATE TABLE Profs(id INTEGER PRIMARY KEY,Titre
TEXT,Prenom TEXT,Nom TEXT,Classe TEXT, Salle TEXT) ;""")
```

Quelques explications :

- Pour exécuter une commande, il faut écrire `cursor.execute("""  """)` avec entre les guillemets les commandes du langage SQL.
- Pour créer une table, on écrit : `CREATE TABLE` suivi de son nom, puis entre parenthèses, on précise le nom de chaque attribut suivi de son type en majuscules (TEXT pour un texte, INTEGER pour un entier…).
- La mention `PRIMARY KEY` est optionnelle pour nos applications simples – Elle impose en tout cas que le champ soit unique pour chaque n-uplet et il sera donc impossible d'ajouter un n-uplet ayant le même id dans notre exemple

Ça y est, les tables sont initialisées, encore vides, et pas encore enregistrées dans le fichier associé à la base de données.

Remarque : si une table existe, il est possible de préciser de créer la table si elle n'existe pas afin d'éviter une erreur si elle existe en précisant : `CREATE TABLE IF NOT EXISTS`

1.VII.2.d.ii Lister les relations d'une base de données (pour info)

```
cursor.execute("""SELECT name FROM sqlite_master WHERE type='table';""")
print(cursor.fetchall())
```

A écrire tel quel sans rien changer !

1.VII.2.d.iii Lister les attributs d'une relation (pour info)

```
cursor.execute("""PRAGMA table_info(Eleves);""")
print(cursor.fetchall())
```

1.VII.2.d.iv Suppression d'une relation

Il suffit d'écrire :

```
cursor.execute("""DROP TABLE Eleves; """)
```

On supprime ainsi la table « Eleves ».

1.VII.2.e Ajout/modification/suppression d'enregistrements

Il existe plusieurs manières de procéder, voyons les deux plus classiques.

1.VII.2.e.i Ajout ligne par ligne

On peut ajouter les donnes ligne par ligne, voici deux possibilités :

```
cursor.execute("""INSERT INTO Eleves VALUES
(1,"Steph","ANE","MPSI",17,"Paris");""")
```

Cette méthode très simple ajoute à la table « Eleves » le nuplet `(1,"Steph","ANE","MPSI",17);"""`. Attention, il doit avoir la même taille que lors de la définition de la relation (CREATE TABLE).

```
cursor.execute("""INSERT INTO Eleves VALUES (?,?,?,?,?,?)
;""",(1,"Steph","ANE","MPSI",17,"Paris"))
```

Cette méthode un peu moins évidente au premier abord se comprend ainsi : Ajouter à la table « Eleves » les 6 valeurs (?, ?, ?, ?,?, ?) précisées dans le nuplet un peu plus loin… Ce fonctionne tant qu'il y a au moins 2 éléments par n-uplet. Je la précise surtout pour que vous compreniez l'écriture de l'ajout d'une liste au paragraphe suivant.

1.VII.2.e.ii Ajout d'une liste de lignes

On crée une liste de nuplets puis on l'ajoute à la table :

```
Liste_Eleves =
[(1,"Steph","ANE","MPSI",17,"Paris"),(2,"Marc","IMBUT","PCSI",18,"Marseil
le")]
cursor.executemany("""INSERT INTO Eleves VALUES
(?,?,?,?,?,?);""",Liste_Eleves)
```

On notera qu'il faut alors utiliser la commande « executemany » et non plus « execute ». Comme précisé précédemment, il faut au moins 2 attributs pour utiliser la méthode (?, ?…).

1.VII.2.e.iii Modification d'un enregistrement : UPDATE SET

```
cursor.execute('''UPDATE Eleves SET Nom = New_Nom, Prenom = New_Prenom
WHERE id = 2 ; ''')
```

On met à jour le nom et le prénom de l'élève d'id 2.

1.VII.2.e.iv Suppression d'un enregistrement : DELETE

```
cursor.execute('''DELETE FROM Eleves WHERE id = 3 ;''')
```

On supprime l'élève d'id 3.

1.VII.2.f Gestion des résultats d'une requête

Maintenant que la base de données est créée, on peut effectuer des requêtes. Voici quelques exemples :

- Combien d'élèves viennent de paris
- Quel âge moyen ont les élèves en 2° année
- En quelle salle doivent aller les élèves de MPSI pour voir leur prof principal

La structure des requêtes est toujours à peu près la même :

```
cursor.execute("""SELECT … FROM … WHERE …;""")
```

Remarque : Il y aura quelques variantes que nous allons voir avec des exemples.

On pourra alors récupérer les résultats de la requête à l'aide de l'une des commandes suivantes :

```
Resultats = cursor.fetchall()
Resultat = cursor.fetchone()
```

Remarque : cursor contient le résultat de la requête précédente uniquement.

La variable Resultat(s) contiendra alors l'ensemble des résultats de la requête avec :

- Pour fetchone, le premier résultat
- Pour fetchall, tous les résultats

Il restera à utiliser Python classiquement pour traiter ces résultats, par exemple :

```
Resultats = cursor.fetchall()
for Resultat in Resultats:
    print("L'élève s'appelle %s %s, il est en classe de %s" %
(Resultat[2], Resultat[3], Resultat[4]))
```

1.VII.2.g Validation des modifications dans le fichier 'BDD.db'

Toutes les modifications qui sont faites sur la base de données sont en réalité réalisées dans la mémoire de Python, et non dans la base de données elle-même. Il faut donc, pour que le fichier de la base de données soit modifié, exécuter une commande « d'enregistrement » en exécutant la ligne :

```
BDD.commit()
```

1.VII.2.h Fermeture de la base de données

Comme pour les fichiers textes, il faut obligatoirement refermer la base de données après utilisation sous Python.

```
BDD.close()
```

1.VII.3 Création d'une nouvelle bdd avec les résultats d'une requête sous python

Voici comment créer une nouvelle relation basée sur le résultat de la requête réalisée. Voici le code permettant de créer une nouvelle relation basée sur les résultats de la requête réalisée (on pourra utiliser ce code pour toutes les requêtes et pas uniquement dans le cadre des unions, intersections et différence) :

```
# Opération d'union, différence, intersection
cursor.execute("""SELECT … FROM … WHERE … UNION SELECT … FROM … WHERE …
;""")
Res_OP = cursor.fetchall()
# Nouvelle base de données
cursor.execute("""CREATE TABLE T_OP(Ref Text,Prix Integer);""")
cursor.executemany("""INSERT INTO T_OP VALUES (?,?);""",Res_OP)
# Affichage du résultat
cursor.execute("""SELECT * FROM T_OP;""")
print(cursor.fetchall())
```