Chapitre 2
Bases de données

Table des matières

Dans ce chapitre, nous allons étudier une manière efficace stocker des données, et surtout de les exploiter, qui est plus performante qu'une création de listes dans Python. On va donc présenter brièvement ce qu'est une *base de données*, et comment accéder aux informations pertinentes par des *requêtes SQL*.

1 Présentation des bases de données

1.1 Stockage de données

Prenons pour exemple le carnet de commande d'un entrepreneur. Il pourrait ressembler à ceci :

Commande	Client	Produit	Prix	Quantité	Adresse de livraison
AF472	Lycée Eiffel	Feutres	2	100	Avenue Champollion
BD312	Mairie Dijon	Papier	10	10	Palais ducs de Bourgogne
TX524	Lycée Eiffel	Papier	10	25	Avenue Champollion
DR964	DFCO	Ballon	50	4	Gaston Gérard
TZ268	Université	Feutres	2	100	Boulevard Gabriel
FG317	GRETA	Papier	10	5	Avenue Champollion

On pourrait imaginer stocker ces informations en Pyhton comme une liste de listes du type :

```
commandes = [['AF472' , 'Lycée Eiffel' , 'Feutres' , 2 , 100 , 'Avenue Champollion'],
['BD312' , 'Mairie Dijon' , 'Papier' , 10 , 10 , 'Palais ducs de Bourgogne'],
['TX524' , 'Lycée Eiffel' , 'Papier' , 10 , 25 , 'Avenue Champollion' ],
['DR964' , 'DFCO' , 'Ballons' , 50 , 4 ,  'Gaston Gérard' ],
['TZ268' , 'Université' , 'Feutres' , 2 , 100 , 'Boulevard Gabriel'],
['FG317' & 'GRETA' & 'Papier' & 10 & 5 & 'Avenue Champollion']
```

On pourrait alors retrouver quels numéros de commande correspondent à des commandes de feutres en faisant :

```
1  feutres = []
2  for i in range (len(commandes)) :
3    if commandes[i][2] = 'Feutres' : feutres.append(commandes[i][0])
4  print feutres
```

On voit déjà que ce n'est pas bien facile, mais on pourrait imaginer vouloir faire des choses plus compliquées (par exemple trier les commandes par ordre décroissant de prix des articles, ou par lieu de livraison, etc). Le problème est que l'écriture des commandes devient bien compliqué.

A la place, on utilise une architecture de base de données, adapté au stockage, à la manipulation et à la recherche des informations.

On souhaite en particulier avoir :

- une structure de données efficace ;
- une rapidité d'accès ;
- éviter à l'utilisateur d'avoir à s'interroger sur la manière dont sont stockées les données ;
- une sauvegarde des modifications ;
- une gestion des pannes ;
- une gestion des conflits si plusieurs utilisateurs modifient la base en même temps.

Dans ce cours, nous ne nous intéresserons pas à la manière dont les bases de données sont réalisées en pratique. Il est toutefois intéressant de savoir que toutes les bases de données suivent un même standard *SQL* pour *Structured Query Language*, qui est celui dont les bases présentées ici sont exigibles aux concours.

Pour le carnet de commandes précédent, une structure en base de données pourrait ressembler à :

Table commandes			
Commande	id_client	id_produit	Quantité
AF472	11	3	100
BD312	12	1	10
TX524	11	1	25
DR964	13	2	4
TZ268	14	3	100
FG317	15	1	5

Table clients		
id_client	Client	Adresse
11	Lycée Eiffel	Avenue Champollion
12	Mairie de Dijon	Palais de ducs de Bourgogne
13	DFCO	Gaston Gérard
14	Université	Boulevard Gabriel
15	GRETA	Avenue Champollion

Table produits		
id_produit	Produit	Prix
1	Papier	10
2	Ballons	50
3	Feutres	2

Nous allons partir de cette exemple afin de définir les différents termes que nous utiliserons, ainsi que la manière d'effectuer des *requêtes* afin d'obtenir les informations souhaitées.

1.2 Vocabulaire

La base de données précédente se compose de 3 tables, composées d'attributs et de tuples (ou p-uplets).

Par exemple, la table commandes comporte 4 *attributs* ou *colonnes*. On peut "facilement" changer le nombre d'attributs d'une table (par exemple ajouter l'attribut "Date de commande" à cette table). A chaque attribut est associé un *domaine*, l'ensemble des valeurs que celui peut prendre relié au type de données (ici, id_commande est une chaine de 5 caractères, les autres attributs de cette table sont des entiers, l'attribut Prix de la table produits est un flottant, etc).

Chaque ligne correspond à un élément unique de la table et est appelé un *enregistrement*.

Lors de la création de la table il est nécessaire de réfléchir à la définition d'une *clé primaire* pour cette table. Une super-clé est un ensemble d'attributs tel que si deux enregistrements sont égaux sur tous ces attributs ils sont égaux partout. Par définition, l'ensemble de tous les attributs est une super clé.

Une clé candidate pour une table est une super clé minimale (donc un ensemble d'attributs qui est une super clé, mais qui ne l'est plus si on enlève un attribut).

Par exemple, pour la table commandes, {id_client, id_produit} est une clé candidate, de même que {id_client, Quantité} alors que ni {id_client, id_produit,Quantité} (elle n'est pas minimale) ni {id_produit, Quantité} (ce n'est pas une super clé, voir les lignes 1 et 5). Toutefois, ces clés candidates ne le sont que par accident, on ne va donc pas les utiliser. Par contre, la super clé constitué du seul attribut Commande} est une clé candidate (comme toute super clé à un seul attribut) que l'on peut choisir comme *clé primaire*. C'est ce que l'on a fait ici, et on note la clé primaire d'une table en la soulignant.

En pratique, on crée facilement une clé primaire en ajoutant un attribut qui est indexé par un entier que l'on incrémente à chaque nouvelle entrée dans la table. Par exemple, on aurait pu écrire la table commandes comme :

Table commandes				
id_commande	Code commande	id_client	id_produit	Quantité
1	AF472	11	3	100
2	BD312	12	1	10
3	TX524	11	1	25
4	DR964	13	2	4
5	TZ268	14	3	100
6	FG317	15	1	5

Les clés primaires sont notées à part car ce sont souvent elles qui permettent de facilement faire le lien entre plusieurs tables (par exemple si l'on veut croiser les informations de la table commandes et de la table client, afin de savoir combien de commandes sont à livrer Avenue Champollion). Ainsi les clés primaires id_client et id_produit des tables clients et produits sont présentes dans la table commandes : on parle dans ce cas de *clés étrangères*. C'est alors un attribut qui doit respecter une contrainte supplémentaire (en plus du domaine) : pour assurer l'intégrité de la base de données, il faut que la valeur de l'enregistrement pour l'attribut id_client de la table commandes corresponde à un enregistrement valide dans la table clients.

2 Requêtes SQL

Une fois la base de données créée (ce qui n'est pas dans les exigences du programme), il est alors possible de l'interroger. Le standard utilisé est le standard *SQL* dont nous allons voir la manière d'écrire les requêtes usuelles.

2.1 Requêtes simples : sélection, projection, renommage

Les premières requêtes que nous allons voir sont les opérations de sélection, de projection ainsi que le renommage. Le résultat d'une requête est une nouvelle table.

Par exemple, la requête :

```
SELECT Code commande FROM Commandes ;
```

renverra la table :

Code commande
AF472
BD312
TX524
DR964
FG317

Il s'agit d'une opération de *projection* (on peut la voir comme la projection sur l'espace *Codecommande* de la table entière, qui est un espace plus grand). Au niveau du vocabulaire, le résultat d'une requête n'est pas nécessairement une table, car après projection, il peut y avoir des doublons (par exemple, si on projète la table commandes selon le seul attribut id_produit), mais par abus de langage, on gardera ce terme.

On peut par ailleurs noter :

- il y a une syntaxe particulière, avec les commandes SQL habituellement notées en lettres majuscules ;
- la requête commence toujours avec `SELECT` ;
- elle se termine toujours avec un point-virgule.

Toutefois, toutes les lignes (tous les enregistrements) ne sont pas nécessairement utiles à l'utilisateur qui écrit la requête. Il est donc souvent nécessaire de préciser quelles sont les lignes utiles : on réalise alors une opération de *sélection*.

Par exemple la requête :

```
SELECT Code commande FROM Commandes WHERE id_produit = 3 ;
```

renverra la table constituée du seul attribut Code commande avec les enregistrements $AF472, TZ268$, qui correspond à la liste des Code commande pour lesquelles le produit est celui identifié par l'id_produit 3 (donc les feutres si on lit la table produits).

On remarque que la sélection se fait une fois la table choisie (logique), et que la condition après le `WHERE` doit être un booléen. On a donc de manière générale les opérateurs `=` et `<>` pour les comparaisons (égalité ou différence) et les connecteurs logiques `AND`, `OR` et `NOT` pour des conditions composées. Si le domaine de l'attribut le permet on peut aussi utiliser `>`, `<`, `>=` et `<=`, et faire des opérations arithmétiques de base `+,-,*,/`.

Il est enfin possible (et souvent recommandé) de renommer les attributs ou les tables avec la commande `AS`. Par exemple, la requête :

```
SELECT Code commande AS code FROM Commandes WHERE id_produit = 3 ;
```

produit la table constituée du seul attribut code avec les enregistrements $AF472, TZ268$. C'est particulièrement utile lorsque l'on croise des tables différentes avec des attributs qui ont le même nom.

2.2 Présentation des résultats

Dans une base de données, ni les attributs, ni les enregistrements ne sont numérotés (il n'existe pas d'attribut numéro 2 par exemple). Le résultat d'une requête produit une table, donc on ne peut pas connaitre *a priori* la manière dont sont ordonnés ses enregistrements. Pour une exploitation pratique, il est donc souvent nécessaire d'ordonner les résultats.

On utilise alors les commandes `ORDER BY` pour classer les éléments, `OFFSET` m pour enlever les m premiers enregistrements, et `LIMIT` n pour limiter les résultats à n lignes. Evidemment, puisqu'on ne connait pas l'ordre dans lequel vont être manipulés les enregistrements, les commandes OFFSET et LIMIT n'ont pas vraiment d'intérêt si les résultats ne sont pas ordonnés avant.

Par exemple la requête :

```
SELECT * FROM Commandes ORDER BY Quantité DESC, id_client ASC;
```

renverra la table :

id_commande	Code commande	id_client	id_produit	Quantité
1	AF472	11	3	100
5	TZ268	14	3	100
3	TX524	11	1	25
2	BD312	12	1	10
6	FG317	15	1	5
4	DR964	13	2	4

Le signe * signifie que tous les attributs de la table sont sélectionnés (donc on fait une copie de la table), que l'on ordonne par ordre décroissant de l'attribut Quantité, puis par ordre croissant de l'id_client en cas d'égalité.

La requête :

```
SELECT * FROM Commandes ORDER BY Quantité DESC, id_client ASC OFFSET 1 LIMIT 2;
```

renverra la table :

id_commande	Code commande	id_client	id_produit	Quantité
5	TZ268	14	3	100
3	TX524	11	1	25

Enfin, il peut être nécessaire d'enlever les doublons d'une table obtenue par projection (en fait, il ne doit pas y avoir de doublons dans une table, donc une requête SQL ne renvoie pas nécessairement une table), ce qui se fait avec la commende `Distinct`

Par exemple :

```
SELECT id_client FROM Commandes ORDER BY Quantité DESC, id_client ASC;
```

renverra une "table" avec les enregistrements {11,14,11,12,15,13} alors que

```
SELECT DISTINCT id_client FROM Commandes ORDER BY Quantité DESC, id_client ASC;
```

renverra bien une table avec les enregistrements {11,14,12,15,13}.

2.3 Fonctions d'agrégation

L'étape suivante consiste souvent à faire des opérations sur les résultats de requête à des fins de statistiques. Parmi les opérations disponibles de base en SQL vous devez connaitre :

- `COUNT` qui donne le nombre de lignes ;
- `MIN` et `MAX` qui donne respectivement l'élément minimal ou maximal d'une colonne ;
- `SUM` qui donne la somme des éléments d'une colonne ;
- `AVG` qui donne la moyenne des éléments d'une colonne.

Le résultat est alors une table avec une unique ligne et une unique colonne. On peut alors utiliser cette valeur pour d'autres opérations (par exemple en groupant les enregistrements sur lesquels doit se faire l'opération).

Ainsi la requête :

```
SELECT COUNT(Code commande) FROM Commandes;
```

renverra la valeur 6.

2.3.1 Groupement des résultats

En groupant les commandes par client selon la requête :

```
SELECT COUNT(Code commande) FROM Commandes GROUP BY id_client ;
```

on obtient la table à une seule colonne :

COUNT(Code commande)
2
1
1
1
1

qui n'est pas très pratique pour savoir quels clients sont concernés. On fera donc plutôt :

```
SELECT id_client, COUNT(Code commande) AS Nombre de commandes FROM Commandes GROUP BY id_client ;
```

pour avoir la table :

id_client	Nombre de commandes
11	2
12	1
13	1
14	1
15	1

2.3.2 Sélection après agrégation

Il peut alors être nécessaire de faire une sélection après l'agrégation des résultats. En pratique, on préfèrerait faire la sélection en premier (afin de manipuler une table la plus petite possible et nécessitant moins d'espace mémoire) mais ce n'est pas toujours possible.

On fait les sélections après agrégation avec la fonction `HAVING`.

Ainsi, si on souhaite savoir quels clients ont fait plusieurs commandes on peut faire la requête :

```
SELECT id_client, COUNT(Code commande) AS Nombre de commandes FROM Commandes GROUP BY id_client HAVING Nom
```

pour avoir la table :

id_client	Nombre de commandes
11	2

2.4 Opérations ensemblistes

Il est ensuite possible de réaliser des opérations ensemblistes sur les requêtes afin de grouper les résultats de plusieurs requêtes différentes.

On utilise ainsi :

- "REQUETE 1 " `UNION` "REQUETE 2" pour obtenir les enregistrements qui correspondent à l'une ou l'autre des deux requêtes ;
- "REQUETE 1 " `INTERSECT` "REQUETE 2" pour obtenir les enregistrements qui correspondent aux deux requêtes ;
- "REQUETE 1 " `EXCEPT` "REQUETE 2" pour obtenir les enregistrements qui correspondent à la première requête mais pas à la deuxième.

On comprend donc bien qu'il faut que les tables réponses des deux requêtes aient les mêmes attributs.

Par exemple, si l'on souhaite avoir la liste des identifiants des clients qui ont fait plusieurs commandes ou dont la quantité commandée est la quantité maximale, on peut faire :

```
(SELECT id_client FROM Commandes GROUP BY id_client HAVING COUNT(Code commande) > 1)
UNION
(SELECT id_client FROM Commandes HAVING Quantité = MAX(Quantité))
```

pour obtenir les identifiants 11 et 14 alors que

```
(SELECT id_client FROM Commandes GROUP BY id_client HAVING COUNT(Code commande) > 1)
INTERSECT
(SELECT id_client FROM Commandes HAVING Quantité = MAX(Quantité))
```

donnerait l'identifiant 11 et

```
(SELECT id_client FROM Commandes HAVING Quantité = MAX(Quantité))
EXCEPT
(SELECT id_client FROM Commandes GROUP BY id_client HAVING COUNT(Code commande) > 1)
```

donnerait l'identifiant 14.

En pratique, sur cet exemple, il est préférable de travailler sur la sélection en amont puisqu'on ne manipule que la table commandes.

2.5 Jointures

2.5.1 Produit cartésien

Enfin, la dernière partie, et la plus intéressante en pratique est celle qui croise les informations de table différentes. Dans notre exemple, on a besoin pour savoir (par exemple) où livrer chaque commande.

Ici, il serait nécessaire de croiser la table commandes (qui contient l'information sur le numéro de commande) et la table clients (qui contient les adresses).

Une manière simple d'obtenir ces informations est alors de faire le produit cartésien des deux tables, ce qui s'obtient très simplement en SQL en séparant les tables à utiliser par une virgule après le FROM.

On peut par exemple faire la requête :

```
SELECT Code commande, Adresse FROM Commandes, Clients
```

On obtient alors la table :

Code commande	Adresse
AF472	Avenue Champollion
AF472	Palais de ducs de Bourgogne
AF472	Gaston Gérard
AF472	Boulevard Gabriel
AF472	Avenue Champollion
BD312	Avenue Champollion
BD312	Palais de ducs de Bourgogne
⋮	⋮
TX524	Boulevard Gabriel
⋮	⋮

On voit ainsi que le produit cartésien d'une table à n lignes avec une table à m lignes produit une table à $n * m$ lignes, donc une table très grande en pratique.

De plus, on voit sur cet exemple que plein de lignes ne sont pas pertinentes (il est évident que la commande AF472 ne doit pas être livré à toutes les adresses...). Pour éviter de créer des tables trop grandes comptant des informations non pertinentes, il est mieux de faire des jointures de tables.

2.5.2 Jointure

Sur notre exemple, il est évident qu'il ne faut sélectionner dans le produit cartésien que les lignes où les identifiants clients correspondent aux deux tables. On peut alors faire une des requêtes :

```
SELECT Code commande, Adresse FROM Commandes AS CO, Clients AS CL WHERE CL.id_client = CO.id_client ;
SELECT Code commande, Adresse FROM Commandes AS CO JOIN Clients AS CL ON CL.id_client = CO.id_client ;
SELECT Code commande, Adresse FROM Commandes NATURAL JOIN Clients ;
```

On remarque alors :

- que SQL s'assure qu'il n'y a pas d'attributs en double en nommant l'attribut "id_client" de la table "Commandes" comme Commandes.id_client (d'où l'intérêt du renommage !) ;
- que l'opération de jointure, contrairement au produit cartésien ne crée pas la table contenant les lignes inutiles, ce qui permet de gagner du temps et de l'espace ;
- si les attributs portent le même nom dans les deux tables, la jointure naturelle est plus simple (mais ce n'est pas forcément le cas).

Pour joindre plusieurs tables, il n'y a pas d'ordre : on peut soit faire T1 JOIN T2 ON Φ_1 JOIN T3 ON Φ_2 JOIN T4 ON Φ_3 ... ou bien T1 JOIN T2 JOIN T3 JOIN T4 ... ON Φ_1 ON Φ_2 ON Φ_3 ...

Il est à noter que si l'attribut n'existe que dans une seule table, il n'est pas nécessaire d'utiliser la notation table.attribut .

3 Bilan

En groupant tout ceci, une requête SQL sera toujours de la forme :

```
SELECT (agrégation) attribut
FROM table1 JOIN table2 ON condition de jointure JOIN ....
WHERE condition de sélection
GROUP BY condition de regroupement
HAVING condition de sélection après agrégation
ORDER BY quantité d'ordonnancement
LIMIT n OFFSET m
```