Mémento pour les requêtes SQL

I - Quelques rappels

Les données constituant la base sont organisées en *tables*. Une table contient des *enregistrements* composés de différents *champs* (ou *attributs*). Le type associé au champ définit le *domaine* dans lequel le champ pourra prendre ses valeurs : date, entier, flottant, chaîne de caractères... Les chaînes s'écrivent entre 'simples' ou "doubles" quotes.
Chaque table est désignée de manière unique par un *identificateur* (son nom) au sein de la base de données, de même que chaque champ au sein d'une table.
Usuellement on représente une table par un tableau dont les colonnes correspondent aux champs et les lignes aux enregistrements.
Une *clé primaire* d'une table est un champ (ou un ensemble de champs) qui identifie de manière unique chaque enregistrement dans la table. Chaque table devrait avoir une clé primaire.
Une *clé étrangère* dans une table est un champ (ou un ensemble de champs) qui fait référence à un champ (ou un ensemble de champs) d'une autre table (généralement la clé primaire).
Les *requêtes SQL* permettent d'interroger une base de données (il existe aussi des commandes SQL pour créer, modifier une base de données, mais cela est hors programme).

II - Les requêtes

1) Définitions

a) Valeur `NULL`

Un champ qui n'est pas renseigné, donc vide, contient la valeur `NULL`. Cette valeur est différente de zéro et représente l'absence de valeur.

b) Expressions

Les expressions valides mettent en jeu des noms de champs, le mot clé « `*` » (qui signifie « tous les champs »), des constantes, des fonctions et des opérateurs classiques. Il existe des fonctions logiques, mathématiques, de manipulation de chaînes, de dates et des *fonctions d'agrégation*.
Les fonctions d'agrégation permettent de calculer un résultat numérique (un entier ou un flottant) à partir d'un ensemble de valeurs. Les fonctions d'agrégation sont au nombre de 5 : `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`. Les quatre dernières s'appliquent à un champ ; par exemple `AVG(nom)` renvoie la moyenne des valeurs contenues dans le champ `nom`.
Le calcul sera effectué groupe par groupe en présence d'une clause `GROUP BY`.

c) Quelques précisions sur `COUNT`

`COUNT(*)` compte le nombre total d'enregistrements dans une table, après restriction éventuelle par un `WHERE` et groupe par groupe en présence d'une clause `GROUP BY`.
`COUNT(nom)` compte le nombre d'enregistrements pour lesquels le champ `nom` contient une valeur autre que `NULL`.
`COUNT(DISTINCT nom)` compte le nombre de valeurs **distinctes** autres que `NULL` présentes dans le champ `nom`.

2) Interrogation d'une base de données

Dans les descriptions qui suivent, les expressions entre crochets sont facultatives.
Lorsque plusieurs opérateurs sont utilisables à un endroit donné, ils sont séparés par des barres verticales (il faut en mettre un seul dans la requête !).

La syntaxe ordinaire est la suivante (mais voir les *jointures* au **3)**) :

```
SELECT... FROM... [WHERE...] [GROUP BY... [HAVING...]] [ORDER BY...] [LIMIT... [OFFSET...]]
```

a) SELECT...

`SELECT [DISTINCT] expr1 [[AS] nom1], expr2 [[AS] nom2]...`
réalise une *projection*. Le mot clé `DISTINCT` permet de supprimer les doublons.
`expr1`, `expr2`... indiquent quelles expressions devront être renvoyées, par exemple des noms de champs. S'il y a une ambiguïté (cas de deux tables contenant des champs de même nom), il est nécessaire de préfixer le nom du champ par celui de la table suivi d'un point.

Le mot clé `AS` permet le *renommage* : `nom1`, `nom2`... sont des *alias* qui fournissent un identificateur abrégé pour chaque champ concerné, pour la suite de la requête. En outre, un tel alias est utilisé comme titre de la colonne correspondante dans le résultat de la requête (noter que le mot clé `AS` est facultatif, mais il améliore la lisibilité).

b) FROM...

`FROM table1 [[AS] alias1], table2 [[AS] alias2]...`
donne la liste des tables participant à l'interrogation. `alias1`, `alias2`... sont des alias attribués aux tables pour le temps de la requête. Quand une table se voit attribuer un alias, elle n'est plus reconnue sous son nom d'origine dans la requête. On peut attribuer plusieurs alias à une même table.

c) WHERE...

`WHERE prédicat`
permet d'effectuer une *restriction*, c'est-à-dire de spécifier quels enregistrements sélectionner dans une table ou un produit cartésien de tables.

1) *Prédicats simples*

Un prédicat simple est la comparaison de plusieurs expressions au moyen d'un opérateur logique :

`WHERE expr1 = | != | < | > | <= | >= expr2` (opérateurs classiques)

`WHERE expr1 [NOT] LIKE expr2` (où `expr2` est un *masque*, chaîne de caractères contenant au moins un joker : `_` pour un unique caractère ou `%` pour un nombre quelconque de caractères ; un masque comme `'%7%'` permet de filtrer les chaînes, **mais aussi les nombres** contenant un 7)

`WHERE expr1 IS [NOT] NULL` (valeur spéciale `NULL` pour un champ vide, cf. **II-1)a)**)

`WHERE expr1 [NOT] IN (expr2,expr3...)` (appartenance à la liste d'expressions)

`WHERE [NOT] EXISTS (expr1)` (teste si la sous-requête `expr1` renvoie au moins un résultat)

2) *Prédicats composés*

Les opérateurs logiques `AND` et `OR` permettent de combiner plusieurs prédicats. `AND` est prioritaire par rapport à `OR` mais l'utilisation de parenthèses permet de modifier l'ordre d'évaluation.

3) *Sous-requêtes*

Le critère de recherche employé dans une clause `WHERE` (l'expression à droite d'un opérateur de comparaison) peut être le résultat d'un `SELECT`.
Dans le cas des opérateurs classiques, la sous–interrogation ne doit ramener qu'une ligne et une colonne. Elle peut ramener plusieurs lignes à la suite des opérateurs `[NOT] IN`, `[NOT] EXISTS`.

d) GROUP BY... [HAVING...]

`GROUP BY expr1, expr2...`
permet de subdiviser la table en *groupes*, chaque groupe étant l'ensemble des enregistrements ayant une valeur commune pour les expressions spécifiées. Les champs de la clause `SELECT` doivent alors être des fonctions d'agrégation ou des expressions figurant dans la clause `GROUP BY`. Comme son nom l'indique, une fonction d'agrégation s'applique à chaque groupe d'enregistrements créé par la clause `GROUP BY`.

`HAVING prédicat`
sert à sélectionner une partie seulement des groupes formés par `GROUP BY`. Le prédicat ne peut porter que sur des fonctions d'agrégation ou des expressions figurant dans la clause `GROUP BY`.

e) ORDER BY...

`ORDER BY expr1 [ASC | DESC], expr2 [ASC | DESC]...`
classe les enregistrements retournés selon les valeurs de `expr1` puis `expr2`... L'ordre par défaut est ***croissant***, la clause `ASC` est donc vraiment facultative ! La clause `DESC` implique un tri par ordre décroissant. Pour préciser lors d'un tri sur quelle expression va porter le tri, il est possible de donner sa position dans la liste des expressions de la clause `SELECT` ou encore l'alias qu'on lui a attribué.

f) LIMIT... [OFFSET...]

`LIMIT nombre [OFFSET décalage]`
limite à `nombre` le nombre d'enregistrement retournés. Cette clause est souvent utilisée après un tri. L'option `OFFSET` permet de zapper les premiers résultats ; par exemple, `LIMIT 5 OFFSET 10` donnera les enregistrements de 11 à 16.

3) Jointure vs produit cartésien

Avec une base de données complexe, on a souvent besoin de rassembler des données réparties dans plusieurs tables, typiquement concaténer les enregistrements provenant de deux tables, `table1` et `table2`, et vérifiant `table1.champ1 = table2.champ2` (l'un des deux `champ1` ou `champ2` est souvent une clé primaire dans sa table).

Pour ce faire, une solution (coûteuse) consiste à former le *produit cartésien* des deux tables, puis à sélectionner parmi le (trop) grand nombre d'enregistrements obtenus :

```
SELECT expr1, expr2...  FROM table1, table2  WHERE table1.champ1 = table2.champ2
```

Il faut préférer à ce type de requête l'utilisation d'une *jointure* (partie du produit cartésien formée des enregistrements vérifiant la condition). Une telle jointure est crée par les logiciels de gestion de bases de données de façon plus efficace que la sélection dans le produit cartésien. La syntaxe est la suivante :

```
SELECT expr1, expr2...  FROM table1 JOIN table2 ON table1.champ1 = table2.champ2
```

Jointure de plusieurs tables : le résultat de `JOIN... ON...` étant une nouvelle table, on peut enchaîner plusieurs clauses `JOIN` en affectant si besoin plusieurs alias à une même table si elle doit intervenir à plusieurs endroits.

4) Les opérateurs ensemblistes

`requête1 UNION | INTERSECT | MINUS requête2 ...`
permettent de réaliser des opérations ensemblistes sur les résultats de plusieurs interrogations.

Les champs renvoyés par les requêtes impliquées doivent être identiques. Les opérations sont évaluées de gauche à droite mais l'utilisation de parenthèses permet de modifier cet ordre.

III - Python et SQLite

Il existe pour Python le module `sqlite3` qui permet de se connecter à une base de données SQLite et de la manipuler en exécutant des commandes SQL.

L'exécution d'une requête renvoie un itérateur d'un type spécial (`Cursor`), que l'on peut utiliser directement à l'aide d'une boucle `for...in`, qui permet de parcourir les lignes du résultat de la requête. Ces lignes se présentent sous forme de *tuples*. On peut aussi convertir le résultat en "liste de lignes" grâce à la méthode `fetchall()`. On peut enfin obtenir une ligne après l'autre avec la méthode `fetchone()`. Mais attention ! Ces trois procédés détruisent les lignes utilisées !! Si l'on souhaite les utiliser plusieurs fois, il est nécessaire de les stocker dans un tableau par une instruction du type `tableau=resultat.fetchall()`.

Voici un exemple qui extrait d'une base de données géographique les cinq communes de France ayant les points culminants les plus élevés :

```
import sqlite3
base = sqlite3.connect("geographie.sqlite")
resultat = base.execute("SELECT nom, zmax FROM communes ORDER BY zmax DESC LIMIT 5")
for ligne in resultat :
    print(ligne)
tableau=resultat.fetchall()
print(tableau)
```

Comme indiqué ci-dessus, on récupère un `tableau` vide !! Les enregistrements sélectionnés par la requête ont été détruits durant la boucle `for`. En revanche, si l'on supprime ladite boucle `for`, alors `tableau` contiendra bien le résultat de la requête, à savoir cette liste de 5 couples :
`[('Chamonix-Mont-Blanc', 4807), ('Saint-Gervais-les-Bains', 4807), ('Houches', 4280), ('Pelvoux', 4099), ('Saint-Christophe-en-Oisans', 4008)]`.