12

SQL : Création d'une BDD et requêtes

Dans ce chapitre, nous voyons comment créer et interroger une base de données. Le langage qui s'est imposé pour cela est le langage SQL (Structured Query Language). La plupart des SGBD actuels utilisent ce langage, avec éventuellement quelques variantes. Ceux ne l'utilisant pas (ACCESS par exemple) ont en général une interface de traduction permettant à l'utilisateur d'entrer ses requêtes en SQL.

Le langage SQL est utilisé tout aussi bien pour la création, le rmplissage et l'interrogation de la base. En ce sens, les 3 instructions principales sont :

- `CREATE` pour créer une table
- `INSERT` pour ajouter des données
- `SELECT ... FROM ... WHERE ...` pour interroger la base (requête)

À ces trois instructions, on peut ajouter `ALTER` permettant des modifications sur la structure de la base de donnée, et `UPDATE` permettant des modifications sur les enregistrements déjà présents (mises à jour).

Nous utiliserons Python pour créer et manipuler les bases de données. Pour cela, nous utilisons la bibliothèque `sqlite3`. Le principe est ensuite de créer une connection sur une BDD avec la fonction `connect`, puis de créer un « curseur » sur cette connection. Grâce à ce curseur, on peut ensuite exécuter sur la base des instructions SQL, grâce à la méthode `execute()` applicable au curseur créé. On passe en paramètre de cette méthode l'instruction SQL sous forme d'une chaîne de caractères (par exemple raw string pour éviter tout problème d'utilisation des guillemets, apostrophes et retours à la ligne).
La connection doit être fermée en fin de programme.
Par exemple, pour créer une table `compositeur` dans une base de données, l'instruction SQL est :

```
CREATE TABLE IF NOT EXISTS compositeur (
    idcomp varchar(6) NOT NULL,
    nom varchar(30) NOT NULL,
    prenom varchar(30) NOT NULL,
    naissance int(4),
    mort int(4),
    PRIMARY KEY (idcomp)
    );
```

Si on veut créer cette base dans un SGBD muni d'une interface graphique conviviale, il est possible de rentrer directement cette instruction telle quelle, voire de façon encore plus simple, suivant l'interface. Pour créer cette table via Python, on écrit :

```
import sqlite3
```

```
connection = sqlite3.connect('musique.db')

cur = connection.cursor()

cur.execute("""CREATE TABLE IF NOT EXISTS compositeur (
    idcomp varchar(6) NOT NULL,
    nom varchar(30) NOT NULL,
    prenom varchar(30) NOT NULL,
    naissance int(4),
    mort int(4),
    PRIMARY KEY (idcomp)
    );""")

connection.commit()
connection.close()
```

De même, chaque requête doit ensuite être envoyée via l'instruction `cur.execute("""requête""")`.
Le résultat parvient sous forme d'un objet itérable, les objets (sous format `tuple`) pouvant être énumérés à l'aide de `for`. Par exemple, la fonction suivante créer l'affichage des objets de l'itérable `cur`, à employer à l'issue d'une requête.

```
def affiche(curseur):
    for L in curseur:
        print(L)
```

Chaque ligne `L` du résultat de la requête est affiché sous forme d'un tuple.

Un des intérêts de l'utilisation d'un langage de programmation (par exemple Python) pour l'interrogation des bases de données est de pouvoir facilement utiliser, et donc traiter informatiquement, le résultat de la requête.

I Création d'une base de données

I.1 Création des tables

Nous ne nous étendrons pas trop sur la création d'une base de données. La création d'une table se fait avec l'instruction `CREATE`.
Si on veut recréer une base déjà existante (par exemple obtenue lors d'une tentative précédente qui ne nous satisfait pas), on peut commencer par effacer les tables existantes, à l'aide de `DROP TABLE`. Pour ne pas faire d'erreur, on peut ajouter une option `IF EXISTS` :

```
DROP TABLE IF EXISTS compositeur;
```

On créer une nouvelle table avec `CREATE TABLE`. L'option `IF NOT EXISTS` permet de ne pas retourner d'erreur au cas où la table existe déjà (dans ce cas, il n'y a pas de nouvelle table créée)
Lorsqu'on crée une table, il faut préciser :

1. son nom
2. pour chacun des attributs : son nom, son format et d'éventuelles contraintes sur lesquelles nous ne nous étendons pas. La seule que nous mentionnons (en plus des contraintes liées aux clés) est la contrainte `NOT NULL`, imposant qu'une valeur soit nécessairement donnée à cet attribut lors d'un enregitrement d'une donnée.
3. la donnée de la clé primaire, et eventuellement des clés étangères.

Par exemple, pour créer une table `oeuvre` (la table `compositeur` étant définie comme plus haut) :

```
CREATE TABLE IF NOT EXISTS oeuvre (
    idoeuvre varchar(10) NOT NULL,
    compositeur varchar(6) NOT NULL,
    oeuvre varchar(128) NOT NULL,
    datecomp int(4) NOT NULL,
    type varchar(20) NOT NULL,
    FOREIGN KEY (compositeur) REFERENCES compositeur,
    PRIMARY KEY (idoeuvre));
```

La clé étangère renvoie à la table `compositeur`. L'identification se fait sur la clé primaire de cette table. Une autre syntaxe possible pour la déclaration des clés est :

```
CREATE TABLE IF NOT EXISTS oeuvre (
    idoeuvre varchar(10) NOT NULL PRIMARY KEY,
    compositeur varchar(6) NOT NULL REFERENCES compositeur,
    oeuvre varchar(128) NOT NULL,
    datecomp int(4) NOT NULL,
    type varchar(20) NOT NULL);
```

Les formats classiques des attributs sont `varchar(n)`, `int(n)`, `decimal(n,m)`, pour des chaînes de caractères, des entiers ou des décimaux. Le paramètre n indique le nombre maximal de caractères, et m indique le nombre de chiffres après la virgule. D'autres types existent (par exemple `date`). À découvrir soi-même.

On peut définir des contraintes beaucoup plus précises sur les différents attributs (notamment des contraintes d'unicité). Ceci dépasse les objectifs de ce chapitre d'introduction.

Il existe également des instructions permettant de modifier des tables existantes, en particulier `ALTER TABLE`. On peut à l'aide de cette instruction modifier le nom d'un attribut, les contraintes, ajouter ou enlever des clés... Là encore, cela va au-delà des objectifs de ce chapitre.

I.2 Entrée des données

Entrer une ligne dans une table se fait avec l'instruction `INSERT INTO ... VALUES ...` , suivant la syntaxe suivante :

```
INSERT INTO compositeur (idcomp, nom, prenom, naissance, mort) VALUES
    ('BAC 1', 'Bach', 'Johann Sebastian', 1685, 1750),
    ('MOZ 1', 'Mozart', 'Wolfgang Amadeus', 1756, 1791),
    ('BEE 1', 'Beethoven', 'Ludwig (van)', 1770, 1827),
    ('BOU 1', 'Boulez', 'Pierre', 1925, 2016);
```

On indique donc le nom de la table, les attributs qu'on souhaite remplir, dans un ordre déterminé, et les n-uplets que l'on souhaite définir, séparés par une virgule. On termine la liste par un point-virgule. Comme le montre la dernière ligne entrée, certains attributs peuvent rester non définis, ce qui revient à leur attribuer la valeur `NULL`. Ceci est possible si l'attribut n'a pas été défini avec la contrainte `NOT NULL` lors de la création de la table.

En n'indiquant qu'un sous-ensemble des attributs de la table, on peut ne remplir que quelques colonnes. Les autres seront alors complétées par la valeur `NULL` :

```
INSERT INTO compositeur (idcomp, nom, prenom) VALUES
    ('RAC 1', 'Rachmaninov', 'Serge'),
    ('DVO 1', 'Dvorak', 'Antonin');
```

On peut aussi modifier des données déjà rentrées avec `UPDATE ... SET ...` :

```
UPDATE compositeur SET naissance = 1873, mort = 1943 WHERE idcomp = 'RAC 1'
```

II Interrogation d'une BDD (Requêtes)

Nous abordons ici l'aspect principal de l'utilisation d'une base de données, à savoir la rédaction de requêtes en langage SQL afin d'extraire d'une base de données les informations qui nous intéressent.

Pour rédiger une requête, **il est indispensable de connaître précisément la structure de la base** (les noms des tables, les noms des attributs, les clés primaires et étrangères). Ainsi, la première étape est souvent de rechercher et comprendre cette structure.

Ensuite, les requêtes sont toutes effectuées à l'aide de l'instruction `SELECT ... FROM ... WHERE ...`, et de constructions algébriques effectuées sur les tables, en particulier de jointures (`JOIN ON`).

Une requête consiste en une demande d'extraction d'un ensemble d'attributs, vérifiant une certaine propriété.

- Les attributs demandés peuvent être pris dans différentes tables (nécessite des jointures ou produits cartésiens)
- Les conditions peuvent porter sur les attributs demandés, ou sur d'autres attributs, des mêmes tables, ou non.
- La réponse consiste en l'ensemble des données des attributs selectionnés, tels que les entrées (complètes) associées vérifient les conditions souhaitées.

L'ensemble des requêtes données en exemple ici porte sur une base de donnée musicale respectant la structure de BDD exposée dans le chapitre précédent, à l'exception des formations musicales, laissées de côté. Ainsi, en plus des deux tables `compositeur` et `oeuvre` déjà créées dans les exemples précédents, nous avons aussi les tables suivantes à créer :

```
CREATE TABLE IF NOT EXISTS nationalité (
    idnat int(3) NOT NULL,
    compositeur varchar(6) NOT NULL,
    pays varchar(30) NOT NULL,
    PRIMARY KEY (idnat),
    FOREIGN KEY (compositeur) REFERENCES compositeur );

CREATE TABLE IF NOT EXISTS soliste (
    idsol int(5) NOT NULL,
    oeuvre varchar(10) NOT NULL,
    instrument varchar(30) NOT NULL,
    PRIMARY KEY (idsol),
    FOREIGN KEY (instrument) REFERENCES instrument,
    FOREIGN KEY (oeuvre) REFERENCES oeuvre );

CREATE TABLE IF NOT EXISTS instrument (
    instrument varchar(30) NOT NULL,
    famille varchar(30) NOT NULL,
     PRIMARY KEY (instrument) );
```

Il est important de noter que la syntaxe utilisée ici est celle définie dans la bibliothèque SQLITE de Python, qui s'écarte parfois de la syntaxe standard. De manière générale, d'une version à l'autre de SQL, il existe souvent des petites variantes dans la syntaxe.

II.1 Requêtes simples

Définition 12.2.1 (Requête simple)

Une requête simple est une requête portant sur l'extraction d'attributs d'une même table, la condition d'extraction s'exprimant uniquement à l'aide des attributs de cette même table.

Ainsi, tout se passe dans une unique table. L'instruction incontournable est :

```
SELECT Att1, Att2, ...
    FROM nom_table
    WHERE conditions
```

Remarque 12.2.2

La sélection `SELECT` est à voir comme une projection sur un sous-ensemble des colonnes (on ne conserve que certaines colonnes), alors que la condition `WHERE` est à voir comme une projection sur un sous-ensemble de lignes (on ne garde que les lignes vérifiant les conditions requises)

La clause `WHERE` est optionnelle. Si on ne l'utilise pas, il s'agit de l'extraction des colonnes souhaitées, sans restriction :

```
SELECT instrument
    FROM instrument
```

retourne l'ensemble des instruments solistes présents dans la base, à savoir :

```
('luth',)
('violoncelle',)
('violon',)
('viole de gambe',)
('clavecin',)
('piano',)
('orgue',)
('flûte',)
('hautbois',)
('baryton',)
('soprano',)
('clarinette',)
('cor',)
('basson',)
('haute-contre',)
```

Formellement, il s'agit d'une projection sur l'ensemble des coordonnées selectionnées. Si le tableau s'appelle `TAB`, et les attributs selectionnés $A_{i_1}, \ldots, A_{i_k}$, on notera formellement $\mathrm{TAB}[A_{i_1}, \ldots, A_{i_k}]$.
L'instruction suivante :

```
SELECT instrument
    FROM instrument
    WHERE famille = 'claviers'
```

renvoie quand à elle uniquement les instruments à clavier :

```
('clavecin',)
('piano',)
('orgue',)
```

Remarquez que pour obtenir l'ensemble des familles d'instruments, on formule assez logiquement la requête :

```
SELECT famille
       FROM instrument
```

On reçoit la réponse suivante, peu satisfaisante :

```
('cordes pincées',)
('cordes frottées',)
('cordes frottées',)
('claviers',)
('claviers',)
('claviers',)
('bois',)
('bois',)
('voix',)
('voix',)
('bois',)
('cuivre',)
('bois',)
('voix',)
```

Ainsi, `SELECT` ne supprime pas les redondances dans les réponses. On peut le forcer à le faire en ajoutant l'option `DISTINCT` :

```
--Requête:
SELECT DISTINCT famille
       FROM instrument

--Réponse:
('cordes pincées',)
('cordes frottées',)
('claviers',)
('bois',)
('voix',)
('cuivre',)
```

Enfin, on peut remplacer la liste des attributs par * si on veut selectionner l'ensemble des attributs de la table :

```
-- Que fait la requête suivante?
SELECT *
   FROM compositeur
   WHERE naissance = 1685
```

Voici une liste non exhaustive de tests possibles pour exprimer les conditions :

```
-- TESTS POSSIBLES POUR LA CONDITION WHERE...

= > < <> <= >= -- comparaisons sur des nombres ou chaînes de caractères.
               -- Attention, 'Z'<'a'
IS NULL        -- teste si la valeur n'a pas été attribuée
IN (A1,A2,...) -- teste l'appartenance à une liste
BETWEEN a AND b -- appartenance à un intervalle (nombre ou chaîne)
```

```
LIKE '_1'       -- compare à une chaîne de caractères où _ représente
                -- un caractère quelconque
LIKE '%1'       -- compare à une chaîne de caractères où % représente
                -- une chaîne (éventuellement vide) quelconque
AND OR NOT      -- opérations booléennes usuelles, à parenthéser bien
```

Par ailleurs, on peut utiliser des opérations sur les attributs, soit pour exprimer des tests, soit pour former de nouvelles colonnes. Former de nouvelles colonnes se fait en indiquant dans la clause SELECT les opérations à faire à partir des autres attributs pour créer cette colonne.

```
-- OPÉRATIONS SUR LES ATTRIBUTS

+ * - /            -- Opérations arithmétiques usuelles
LENGTH(t)      -- longueur de la chaîne de caractères t
ch1 || ch2         -- concaténation
REPLACE(attribut,ch1,ch2) -- remplace dans l'attribut les sous-chaînes
                      -- ch1 par ch2
LOWER(ch)          -- met en minuscules
UPPER(ch)          -- met en majuscules
SUBSTR(ch,a,b)     -- extrait la sous-chaîne des indices a à b
                   -- SUBSTRING dans de nombreuses versions
```

Voici quelques exemples :

```
-- Que font les requêtes suivantes?

   SELECT nom
        FROM compositeur
        WHERE LENGTH(nom) = 6

   SELECT prenom || ' ' || nom
        FROM compositeur
        WHERE nom LIKE 'B%'

   SELECT UPPER(nom), LOWER(prenom)
        FROM compositeur
        WHERE mort BETWEEN 1820 AND 1830

   SELECT UPPER(SUBSTR(nom,1,3))
        FROM compositeur
        WHERE naissance < 1600
```

Certaines opérations mathématiques portant sur l'ensemble des valeurs d'un attribut sont possibles (fonctions agrégatives) :

```
-- FONCTIONS AGRÉGATIVES

COUNT(*)     -- nombre de lignes
COUNT(ATTR)  -- nombre de lignes remplies pour cet attribut
AVG(ATTR)    -- moyenne
SUM(ATTR)    -- somme
MIN(ATTR)    -- minimum
MAX(ATTR)    -- maximum
```

Les fonctions agréatives servent à définir de nouvelles colonnes, mais ne peuvent pas être utilisées dans une condition. Un exemple :

```
-- Requête: âge du compositeur mort le plus vieux

   SELECT MAX(mort - naissance)
       FROM compositeur
```

Les fonctions agrégatives peuvent s'utiliser avec l'instruction `GROUP BY Att HAVING Cond` permettant de calculer les fonctions agrégatives sur des paquets de données prenant la même valeur pour l'attribut Att, en ne gardant que les lignes vérifiant la condition Cond.

```
-- Date de la première sonate pour des compositeurs avant N dans
-- l'ordre alphabétique.

   SELECT compositeur, MIN(datecomp)
       FROM oeuvre
       GROUP BY compositeur, type
       HAVING (compositeur < 'N') AND (type = 'sonate')
```

Remarque 12.2.3

- L'instruction `HAVING` effectue une selection sur la table obtenue après calcul de la fonction agrégative.
- On peut aussi utiliser l'instruction `WHERE`, se plaçant alors avant `GROUP BY`. Ceci permet d'effectuer une selection avant calcul de la fonction agrégative.
- Attention, les champs sélectionnés doivent en théorie respecter le groupement dans le sens où la valeur du champ doit être la même pour toutes les entrées groupées dans une part. Dans notre exemple, c'est le cas puisqu'on selectionne un champ groupé `compositeur`. Toute autre sélection est incorrecte en SQL strict, mais tolérée dans certaines variantes (SQLite utilisé sous Python par exemple). On peut parfois s'en sortir, notamment pour le maximum, en utilisant une structure `ORDER BY ... LIMIT ...` explicitée ci-dessous.

Dans notre exemple, la selection peut s'effectuer avant ou après calcul de la fonction agrégative (puisque la selection se fait sur des paquets d'enregistrement de même compositeur et même type). Ainsi, on aurait pu écrire :

```
   SELECT compositeur, MIN(datecomp)
       FROM oeuvre
       WHERE (compositeur < 'N') AND (type = 'sonate')
       GROUP BY compositeur, type
```

Dans le cas où la selection peut se faire avant comme ici, c'est préférable, car plus économe en calcul (la fonction agrégative n'est calculée que sur la selection faite)
Certaines selections ne peuvent être effectuées qu'avant (lorsqu'elles ne sont pas faites sur des paquets respectant le groupement). Par exemple, extraire de la base la moyenne d'âge des compositeurs nés entre 1800 et 1900 (ici, il n'y a pas de groupement à faire, mais on pourrait imaginer par exemple un groupement par nationalité, en cherchant cette information dans une autre table, avec la syntaxe qu'on verra plus loin) :

```
   SELECT AVG(mort - naissance)
       FROM compositeur
       WHERE naissance BETWEEN 1800 AND 1900
```

Certaines selections ne peuvent être effectuées qu'après, notamment celles qui utilisent le résultat du calcul. Pour accéder à la valeur de la fonction agrégative pour pouvoir l'utiliser dans un test, il faut au passage renommer la colonne correspondante, ce qui se fait avec `AS`. Par exemple, pour récupérer la première sonate des compositeurs avant N dans l'ordre alphabétique, mais seulement si celle-ci est écrite après 1800, on peut écrire :

```
SELECT compositeur, MIN(datecomp) AS dt
    FROM oeuvre
    WHERE (compositeur < 'N') AND (type = 'sonate')
    GROUP BY compositeur, type
    HAVING dt >= 1800
```

En revanche, la syntaxe suivante renvoie une erreur :

```
SELECT compositeur, MIN(datecomp) AS dt
    FROM oeuvre
    WHERE (compositeur < 'N') AND (type = 'sonate') AND (dt >= 1800)
    GROUP BY compositeur, type
```

En effet, la colonne `dt` n'a pas encore été calculée au moment du test qui l'utilise.
On aurait pu penser écrire :

```
SELECT compositeur, MIN(datecomp) AS dt
    FROM oeuvre
    WHERE (compositeur < 'N') AND (type = 'sonate') AND (datecomp >= 1800)
    GROUP BY compositeur, type
```

mais l'effet de cette selection est un peu différent.

Exercice 20

Comprendre la différence entre cette requête et la précédente.

Enfin, notons qu'il est possible d'ordonner les résultats par ordre croissant (numérique ou alphabétique) grâce à ORDER BY :

```
-- Compositeurs nés entre 1870 et 1890, par ordre de naissance

SELECT nom, naissance, mort
    FROM compositeur
    WHERE naissance BETWEEN 1870 AND 1890
    ORDER BY naissance
```

La structure `ORDER BY` peut s'utiliser avec une clause `LIMIT n` où n est un entier. Dans ce cas, on récupère uniquement les n premiers résultats après classement. Dans l'exemple ci-dessus, on peut modifier la requête de sorte à récupérer les 2 premières lignes uniquement :

```
-- 2 premiers compositeurs nés entre 1870 et 1890, par ordre de naissance

SELECT nom, naissance, mort
    FROM compositeur
    WHERE naissance BETWEEN 1870 AND 1890
    ORDER BY naissance
    LIMIT 2
```

Cette structure peut être intéressant pour obtenir une ligne réalisant le maximum d'un de ses attributs (avec `LIMIT 1`). Mais cela se retourne qu'une des éventuelles plusieurs lignes réalisant le maximum de cet attribut.

II.2 Sous-requêtes

Définition 12.2.4 (Sous-requête)

Une sous-requête est une requête portant sur l'extraction d'attributs d'une même table, la condition s'exprimant à l'aide d'attributs d'une autre table. Plus précisément, on extrait les attributs $A_1, \cdots A_k$, et on exprime sur A_1 une condition portant sur les attributs de la table T_1, telle que (A_1, T_1) soit une clé étrangère. On peut combiner grâce aux opérations booléennes, plusieurs tests, pour les différents A_i, impliquant plusieurs tables T_i, en suivant les clés étrangères.

La syntaxe à suivre est la suivante :

```
SELECT A1, A2, ..., Ak
    FROM T
    WHERE A1 IN (SELECT B1
                    FROM T1
                    WHERE condition)
```

Ici, B_1 est la clé primaire de T_1, donc l'attribut vers lequel réfère la clé étrangère (A_1, T_1). On exprime dons la condition sur B_1, en extrayant de T_1 l'attribut B_1, lorsque la condition requise est satisfaite. On teste ensuite l'appartenance de A_1 à la liste des admissibles ainsi obtenue.

Remarque 12.2.5

- Il est important de remarquer que le résultat d'une requête SELECT a le même format qu'une table, et peut donc être réutilisé dans une autre requête SELECT comme toute table. On peut donc imbriquer des instructions SELECT les unes dans les autres.
- On peut aussi remonter les clés étrangères : dans ce cas, la clé étrangère est (B_1, T), et A_1 est la clé primaire de T.

Un exemple :

```
-- Les compositeurs de la base ayant écrit au moins une musique de film:
   SELECT nom, prenom
        FROM compositeur
        WHERE idcomp in (SELECT DISTINCT compositeur
                        FROM oeuvre
                        WHERE type = 'film')
```

Exercice 21

Extraire de la base les oeuvres (compositeur, titre) utilisant un hautbois solo.

On peut enchaîner de la sorte des sous-requêtes en suivant les clés étrangères, si la condition porte sur une table accessible en plusieurs étapes.

Exercice 22

Trouver les oeuvres de compositeur français utilisant un instrument soliste dans la famille des cordes frottées

Le principe de la sous-requête permet aussi d'utiliser le résultats de fonctions agrégatives dans un test :

```
-- Compositeurs morts 20 ans avant l'âge moyen des compositeurs de la
-- base
   SELECT nom, prenom, naissance, mort
      FROM compositeur
      WHERE mort - naissance +20 < (SELECT AVG(mort-naissance)
                     FROM compositeur)
```

II.3 Constructions ensemblistes

Lorsqu'on veut extraire des attributs de plusieurs tables, il faut utiliser des constructions ensemblistes permettant de combiner plusieurs tables. Nous introduisons ici l'union, l'intersection et le produit cartésien. C'est à partir de cette dernière construction qu'on construira des « jointures » permettant des requêtes complexes sur plusieurs tables. Nous isolons cette dernière construction dans le paragraphe suivant.

Définition 12.2.6 (Intersection)
L'intersection de deux extractions de deux tables peut se faire à condition que les attributs extraits des deux tables soient de même format. Généralement, elle se fait sur les attributs reliés par une clé (l'un d'eux étant une clé primaire). Le résultat est alors l'ensemble des valeurs de cet (ou ces) attribut(s) commun aux deux tables. Chacune des deux tables peut elle-même être le résultat d'une extraction précédente.

La syntaxe utilise `INTERSECT`. Sur un exemple :

```
-- Identifiants de compositeurs nés entre 1700 et 1800 et ayant écrit
-- une sonate.

  SELECT idcomp FROM compositeur
               WHERE naissance BETWEEN 1700 AND 1800
  INTERSECT
  SELECT DISTINCT compositeur FROM oeuvre
                    WHERE type = 'sonate'
```

Une intersection peut souvent se reexprimer plus simplement avec une sous-requête et une opération booléenne `AND`. Cela peut être efficace lorsqu'on veut croiser deux tables complètes (par exemple deux tables de clients de deux filiales) :

```
SELECT * FROM table1
INTERSECT
SELECT * FROM table2
```

Définition 12.2.7 (Union)
L'union de deux tables est possible si les attributs sont en même nombre et de même type. Il s'agit des enregistrements présents dans l'une ou l'autre de ces tables. L'union ne conserve que les attributs distincts.

Par exemple pour fusionner deux tables :

```
SELECT * FROM table1
UNION
SELECT * FROM table2
```

Ici encore, il est souvent possible de remplacer avantageusement une union par une opération booléenne OR.
On peut aussi faire des exceptions $A \setminus B$, sur des tables de même type :

```
SELECT instrument FROM instrument
                    WHERE famille = 'bois'
EXCEPT
SELECT instrument FROM soliste
    WHERE oeuvre in
       (SELECT idoeuvre FROM oeuvre WHERE compositeur IN
            (SELECT idcomp FROM compositeur
             WHERE idcomp in
                      (SELECT compositeur FROM nationalité
                       WHERE pays = 'Allemagne')))
```

Exercice 23

Que fait la requête ci-dessus ?

La dernière opération ensembliste est le produit cartésien.

Définition 12.2.8 (Produit cartésien de deux tables)

Le produit cartésien de deux tables $\mathcal{R}_1$ et $\mathcal{R}_2$ est l'ensemble des $n+p$-uplets $(x_1, \ldots, x_n, y_1, \ldots, y_p)$, pour toutes les $(x_1, \ldots, x_n) \in \mathcal{R}_1$ et $(y_1, \ldots, y_p) \in \mathcal{R}_2$, sans préoccupation de concordance des attributs de $\mathcal{R}_1$ et $\mathcal{R}_2$ qui pourraient être reliés.

Le produit cartésien se fait en extrayant simultanément les colonnes des deux tables :

```
SELECT * FROM tab1, tab2
```

On peut faire le produit cartésien d'extractions de deux tables. Dans ce cas, on liste les attributs à garder après `SELECT`, en précisant dans quelle table l'attribut se trouve par une notation suffixe. Si l'attribut (sous le même nom) n'apparaît pas dans les deux tables, on peut omettre la notation suffixe (il n'y a pas d'ambiguïté) :

```
SELECT tab1.Att1, Att2, tab2.Att3 FROM tab1, tab2
```

Dans cet exemple, l'attribut Att2 est supposé n'exister que dans l'une des deux tables.
Pour éviter des lourdeurs d'écriture (dans les conditions), et pouvoir référer plus simplement aux différents attributs (c'est utile aussi lorsqu'un attribut est obtenu par un calcul et non directement), on peut donner un alias aux attributs selectionnés

```
SELECT tab1.Att1 AS B1, Att2*Att4 AS B2 , tab2.Att3 AS B3 FROM tab1, tab2
```

Ici, on suppose que Att2 et Att4 sont deux attributs numériques. On pourra utiliser les valeurs des attributs de la nouvelle table via les alias B1, B2, et B3.
Le produit cartésien est une opération coûteuse et peu pertinente en pratique si elle n'est pas utilisée en parallèle avec une opération de sélection des lignes. En effet, on associe le plus souvent des lignes n'ayant rien à voir, par exemple une ligne consacrée à une oeuvre de Mozart et une ligne consacrée aux données personnelles de Stravinsky. Mais cette opération est le point de départ de la notion de jointure.

II.4 Jointure

La technique de la jointure permet de former une table à l'aide d'une selection d'attributs provenant de deux tables différentes :

Définition 12.2.9 (Jointure)
Une jointure de deux tables consiste à considérer le produit cartésien de ces deux tables (ou d'extractions), en identifiant deux colonnes (une de chaque table, typiquement d'un côté une clé étrangère vers l'autre table, de l'autre la clé primaire), de sorte à ne garder dans le produit cartésien que les n-uplets tels que les valeurs soient les mêmes pour ces deux attributs.

En d'autre terme, une jointure revient à une instruction de sélection sur le produit cartésien. Par exemple, pour sélectionner des attributs de deux tables en faisant une jointure en identifiant l'attribut `Att4` de la table 2 et l'attribut `Att1` de la table 1 :

```
SELECT T1.Att1, T1.Att2, T2.Att1
FROM T1, T2
WHERE T1.Att1 = T2.Att4
```

Un exemple concret :

```
SELECT nom, prenom, titre FROM compositeur, oeuvre
    WHERE idcomp = compositeur
```

Une autre syntaxe parfois plus commode (notamment pour itérer le procéder en opérant des jointures successives) se fait avec `INNER JOIN ... ON ...` (le terme `INNER` étant facultatif) :

```
SELECT nom, prenom, titre
FROM compositeur JOIN oeuvre ON idcomp = compositeur
```

On peut combiner cette jointure avec une sélection conditionnelle `WHERE` :

```
-- Que fait cette requête?

SELECT nom, prenom, titre
FROM compositeur JOIN oeuvre ON idcomp = compositeur
WHERE datecomp - naissance <= 18
ORDER BY nom
```

On peut faire des jointures successives si on veut des attributs de tableaux plus éloignés, ou si on veut extraire des données de plus de deux tables :

```
-- Que fait cette requête?

SELECT nom, prenom, titre, soliste.instrument
FROM compositeur
JOIN oeuvre ON idcomp = compositeur
JOIN soliste ON oeuvre = idoeuvre
JOIN instrument ON instrument.instrument = soliste.instrument
WHERE famille = 'bois'
ORDER BY nom
```

II.5 Algèbre relationnelle

Il s'agit de la formalisation algébrique des constructions étudiées précédemment. On rappelle qu'une relation $\mathcal{R}$ (c'est-à-dire une table) est la donnée d'un ensemble de k-uplets, vérifiant certaines contraintes liées à un schéma relationnel $\mathcal{S}$. On définit alors formellement les constructions étudiées dans les paragraphes précédents.

Définition 12.2.10 (Union, intersection, exception, produit cartésien)

Les constructions d'union, intersection, exception correspondent précisément aux constructions algébriques usuelles sur les relations : $\mathcal{R}_1 \cup \mathcal{R}_2$, $\mathcal{R}_1 \cap \mathcal{R}_2$, $(\mathcal{R}_1 \cup \mathcal{R}_2) \setminus \mathcal{R}_2$ et $\mathcal{R}_1 \times \mathcal{R}_2$.

Définition 12.2.11 (Projection)

L'opération de projection de la relation $\mathcal{R}$ sur les attributs $A_{i_1}, \ldots, A_{i_k}$ consiste en l'extraction sans répétition des attributs $A_{i_1}, \ldots, A_{i_k}$ de la relation $\mathcal{R}$. Il s'agit donc de l'image de $\mathcal{R}$ par la projection

$$\pi_{(A_{i_1},\ldots,A_{i_k})} : \mathrm{dom}(A_1) \times \cdots \times \mathrm{dom}(A_n) \longrightarrow \mathrm{dom}(A_{i_1}) \times \cdots \times \mathrm{dom}(A_{i_k}).$$

La nouvelle relation obtenue par projection est donc notée à juste titre $\pi_{(A_{i_1},\ldots,A_{i_k})}(\mathcal{R})$.

En SQL, la projection correspond donc à l'instruction :

```
SELECT DISTINCT Ai1, ..., Aik FROM R
```

Définition 12.2.12 (Sélection)

Soit $\mathcal{R}$ une relation, et E une expression logique exprimant une condition sur les attributs de $\mathcal{R}$. La selection, notée $\sigma_E(\mathcal{R})$ est la nouvelle relation obtenue de $\mathcal{R}$ en ne conservant que les enregistrements vérifiant la condition E. Ainsi, en notant $E(x)$ l'expression booléenne égale à `True` si et seulement si l'enregistrement x vérifie la condition E, la selection est définie par :

$$\sigma_E(\mathcal{R}) = \{x \in \mathcal{R} \mid E(x)\}$$

En SQL, il s'agit donc de l'instruction :

```
SELECT * FROM R WHERE E
```

Définition 12.2.13 (Renommage)

Le renommage consiste à donner un nouveau nom à un attribut d'une table. Il ne s'agit pas nécessairement d'une modification définitive de la base (possible avec `ALTER` dans certains SGBD), mais d'une modification locale, résultat d'une requête. Le résultat du renommage de l'attribut A en B se note $\rho_{A \leftarrow B}(\mathcal{R})$.

En SQL, on effectue un renommage avec `AS` :

```
SELECT A1, ..., Ak AS B, ... An FROM R
```

Définition 12.2.14 (Jointure)

Soient $\mathcal{R}_1$ et $\mathcal{R}_2$ deux tables, et A et B deux attributs, l'un de $\mathcal{R}_1$, l'autre de $\mathcal{R}_2$. La jointure de $\mathcal{R}_1$ et $\mathcal{R}_2$ sur l'expression $A = B$ est la table obtenue en conservant tous les attributs de $\mathcal{R}_1$ et de $\mathcal{R}_2$, recollées sur les attributs A et B : ainsi, les attributs A et B sont confondus (ne forment qu'un attribut de la nouvelle table), et les enregistrements sont formés de la concaténation des valeurs des attributs de $\mathcal{R}_1$ et de $\mathcal{R}_2$, respectant la condition $A = B$. La jointure est notée $\mathcal{R}_1 \bowtie_{A=B} \mathcal{R}_2$.
Formellement, elle peut être décrite par :

$$\mathcal{R}_1 \bowtie_{A=B} \mathcal{R}_2 = \pi_{(A_1,\ldots,A_n,B_1,\ldots,\hat{B},\ldots,B_n)}(\sigma_{A=B}(\mathcal{R}_1 \times \mathcal{R}_2)),$$

où la notation $\hat{B}$ signifie qu'on a ôté l'attribut B de la liste.

En SQL, la jointure $\mathcal{R}_1 \bowtie_{A=B} \mathcal{R}_2$ s'effectue de la façon suivante :

```
SELECT * FROM R1 JOIN R2 ON A=B
```

Exercice 24

Exprimer les résultats de toutes les requêtes de ce chapitre n'utilisant pas de fonctions agrégatives de façon formelle, à l'aide des opérations de l'algèbre relationnelle.