Requêtes SQL et mise à jour

Nous avons vu au chapitre précédent comment créer des tables et les remplir. Nous avons maintenant une base de données, c'est-à-dire un ensemble de tables, contenant des données *cohérentes* vis à vis de nos contraintes d'intégrité. Nous allons maintenant voir deux autres utilisations d'un SGBD :

- la sélection de données ;
- la mise à jour des données.

La sélection va consister en l'écriture de *requêtes SQL* permettant de trouver toutes les données de la base vérifiant un certain critère. Le premier rôle du programmeur de bases de données va donc être celui qui consiste à traduire des questions que l'on se pose sur les données du langage naturel au langage SQL, afin que le SGBD puisse y répondre.

Reprenons encore une fois le fil conducteur de notre introduction aux bases de données, à savoir celui de la médiathèque municipale. Quelles questions peut-on poser à la base de données ? Il semble naturel que le SGBD puisse répondre aux questions suivantes, nécessaire au bon fonctionnement de la médiathèque.

- Etant donné un code barre, quels sont les livres empruntés par l'utilisateur correspondant ?
- Étant donné un ISBN, le livre correspondant est-il emprunté ?
- Quels sont les utilisateurs en retard, c'est-à-dire ceux dont la date de retour est inférieure à une date donnée ?
- Quels sont tous les livres écrits par Voltaire qui ne sont pas empruntés ?
- Quel est le nombre total de livres empruntés ?

Une autre fonction importante du SGBD est la mise à jour des données. Elle peut consister en une modification d'une ligne existante (par exemple, pour changer l'adresse d'un utilisateur ayant déménagé, sans modifier son code barre, son nom ou son e-mail) ou une suppression (par exemple lorsqu'un utilisateur rend un livre, il faut supprimer la ligne correspondante dans la table emprunt).

Afin de rendre les exemples plus parlants, nous utilisons dans la suite le SGBD libre PostgreSQL, avec une base de données de médiathèque fictive.

A. Sélection de données

Requête sur une table

Commençons par une requête simple. On considère la table livre, créée par l'ordre suivant :

```
CREATE TABLE livre (titre VARCHAR(300) NOT NULL,
                    éditeur VARCHAR(90) NOT NULL,
                    annee INT NOT NULL,
                    isbn CHAR(14) PRIMARY KEY) ;
```

On souhaite trouver les titres de tous les livres publiés après 1990 dans la base de données de la médiathèque. Une telle requête peut s'écrire en SQL :

```
SELECT titre FROM livre WHERE annee >= 1990;
```

titre
Les Aventures de Huckleberry Finn
Fondation et Empire
Akira
Les Robots
Astérix chez les Pietés
Les Monades urbaines
Les Voyages de Gulliver
Lolita
La Nuit des temps
Ravage
⋮ (112 résultats)

Dans cette requête, la partie « `FROM livre` » indique la table sur laquelle porte la requête. La partie « `WHERE ...` » indique que l'on ne sélectionne que les lignes de la table livre pour lesquelles la valeur de 1'attribut année est plus grande que 1990. Enfin, la partie `SELECT` titre indique qu'on ne veut renvoyer que les valeurs de l'attribut titre des lignes trouvées. On remarque que le résultat d'une requête « `SELECT ...` » est une table, ici possédant une unique colonne titre.

Clause `WHERE`. L'expression se trouvant dans la partie `WHERE` doit être une expression booléenne. Elle peut être construite à partir d'opérateurs de comparaison (<, <=, >, >=, = et <>), d'opérateurs arithmétiques (+, -, *, /, %), de constantes, de noms d'attributs, d'opérateurs logiques (`AND`, `OR` et `NOT`) et d'opérateurs spéciaux tels que l'opérateur de comparaison de textes `LIKE`.

Par exemple, si l'on souhaite afficher les titres de tous les livres publiés par Dargaud entre 1970 et 1980, on pourra écrire :

```
SELECT titre FROM livre WHERE année >= 1970 AND
                              année <= 1980 AND
                              éditeur = 'Dargaud';
```

titre
Astérix chez les Belges

Si l'utilisation de l'égalité « = » est appropriée ici, on pourrait vouloir faire une requête approchée. Par exemple, trouver les titres des livres qui contiennent le mot « Astérix » dans le titre. Une telle requête s'écrira

```
SELECT titre FROM livre WHERE titre LIKE '%Astérix%';
```

titre
Astérix et les Normands
Le Tour de Gaule d'Astérix
Astérix chez les Pictes
Astérix chez les Belges
Astérix et Cléopâtre
Astérix légionnaire
Astérix et la Transitalique
Astérix chez les Bretons
Astérix en Corse
L'Odyssée d'Astérix

La chaîne de caractères « `'%Astérix%'` » est un motif. L'opération *s* `LIKE` *m* s'évalue à vrai si et seulement si la chaîne de caractères *s* correspond au motif *m*. Dans un motif, le symbole « `%` » est un joker et peut être substitué par n'importe quelle chaîne. Le symbole « _ » quant à lui représente n'importe

quel caractère. Ainsi, le motif « `'%Astérix%'` » correspond à n'importe quelle chaîne dans laquelle les caractères Astérix sont précédés ou suivis de caractères quelconques.

Clause `SELECT`. La clause `SELECT` peut prendre trois formes (dans notre présentation simplifiée). La première est celle où l'on liste explicitement les attributs que l'on désire renvoyer. Si on veut renvoyer les titres et l'ISBN des livres publiés après 1990, on écrira ceci :

```
SELECT titre, isbn FROM livre WHERE annee >= 1990;
```

titre	isbn
Les Aventures de Huckleberry Finn	978-2081509511
Fondation et Empire	978-2207249123
Akira	978-2723428262
Les Robots	978-2745989857
Astérix chez les Pictes	978-2864972662
Les Monades urbaines	978-2221197691
Les Voyages de Gulliver	978-2335008586
Lolita	978-0141391601
La Nuit des temps	978-2258116429
Ravage	978-2072534911
⋮ (112 résultats)	

Le résultat est de nouveau une table, mais cette fois avec les colonnes (ou attributs) `titre` et `isbn`. Il est possible de renommer les colonnes au moyen du mot clé `AS`.

```
SELECT titre AS le_titre, isbn AS num_serie FROM livre WHERE année >= 1990;
```

le_titre	num_serie
Les Aventures de Huckleberry Finn	978-2081509511
Fondation et Empire	978-2207249123
Akira	978-2723428262
⋮ (112 résultats)	

Ainsi, si la clause `WHERE` d'une requête permet de restreindre les lignes de la table que l'on renvoie (en ne gardant que celle vérifiant la condition), la clause `SELECT` permet de restreindre la liste des colonnes.

La seconde forme de la clause `SELECT` est celle que l'on utilise lorsqu'on veut conserver toutes les colonnes. En effet, il serait fastidieux de récrire toutes les colonnes d'une table. On peut utiliser à cette fin le symbole « * » qui signifie « toutes les colonnes de la table ».

```
SELECT * FROM livre WHERE année >= 1990;
```

titre	editeur	annee	isbn
Les Aventures de H…	Flammarion	2020	978-2081509511
Fondation et Empire	Editions Denoël	1999	978-2207249123
Akira	Glénat	2000	978-2723428262
Les Robots	Éditions Milan	2017	978-2745989857
Astérix chez les Pictes	Éditions Alb…	2013	978-2864972662
Les Monades urbaines	Robert Laffont	2016	978-2221197691
Les Voyages de Gulliver	Primento	2015	978-2335008586
Lolita	Penguin UK	2012	978-0141391601
La Nuit des temps	Presses de la Cité	2014	978-2258116429
Ravage	Éditions Gallimard	2014	978-2072534911
⋮ (112 résultats)			

La troisième utilisation de la clause SELECT est celle permettant d'appeler des *fonctions d'agrégation*. Ces dernières permettent d'appliquer une fonction à l'ensemble des valeurs d'une colonne et de renvoyer le résultat comme une table ayant une seule case (une ligne et une colonne). Nous présentons quelques unes de ces fonctions : COUNT qui permet d'obtenir le nombre de résultats, AVG (pour l'anglais average) qui permet de calculer la moyenne d'une colonne, SUM qui permet d'en faire la somme, MIN et MAX qui permettent de trouver respectivement le minimum et maximum d'une colonne. Si on souhaite savoir combien de livres contiennent la chaîne « Astérix » dans leur titre (plutôt que de renvoyer ces titres), on écrira la requête suivante :

```
SELECT COUNT(titre) AS total FROM livre WHERE titre LIKE '%Astérix%' ;
```

total
10

Notons que nous avons choisi de renommer la colonne. En effet, le résultat n'étant pas directement une colonne d'une table existante, les SGBD choisissent un nom arbitraire, souvent peu parlant. La fonction « COUNT » ne faisant que compter la taille de la colonne, elle peut s'appliquer à n'importe quel nom de colonne et même au symbole « * ». Il est donc courant d'écrire une requête de la forme

```
SELECT COUNT(*) AS total FROM livre WHERE titre LIKE '%Astérix%';
```

qui donnera le même résultat que précédemment. Les fonctions AVG et SUM ne peuvent s'appliquer qu'à des colonnes dont le domaine est un nombre. On peut écrire par exemple

```
SELECT SUM(annee) AS somme FROM livre;
SELECT AVG(annee) AS moyenne FROM livre;
```

somme	moyenne
256701	2005.4765625

Ici, en l'absence de clause WHERE, toutes les lignes sont sélectionnées. La somme et la moyenne des années sont calculées et renvoyées comme une table. Enfin, les fonctions MIN() et MAX() peuvent s'appliquer sur n'importe quelle colonne et la comparaison pour son type sera utilisée pour déterminer le plus petit ou le plus grand élément.

```
SELECT MIN(année) AS inf FROM livre;
SELECT MAX(annee) AS sup FROM livre;
```

inf	sup
1933	2020

Tri et suppression des doublons. Comme nous avons pu l'observer lors de nos premières requêtes, les résultats sont affichés par le SGBD dans un ordre a priori quelconque. La situation est même plus complexe. En effet, en fonction de certains paramètres, le SGBD peut choisir entre différentes façons de calculer la requête. L'ordre peut donc être modifié entre deux exécutions de la même requête. Si l'on désire obtenir les résultats dans un ordre particulier, on peut utiliser la clause ORDER BY en fin de requête.

```
SELECT titre FROM livre WHERE annee >= 1990 ORDER BY titre ASC;
```

titre
Akira
Algorithms
Anna Karénine
Astérix chez les Bretons
Astérix chez les Pictes
Astérix en Corse
Astérix et Cléopâtre
Astérix et la Transitalique
Astérix et les Normands
Astérix légionnaire
⋮ (112 résultats)

Ici, on demande au SGBD de trier les résultats par titre croissants `ASC` (pour l'anglais ascending). Si on souhaite trier par valeurs décroissantes il suffit d'utiliser le mot clé `DESC` (pour l'anglais descending) à la place de `ASC`. Supposons maintenant que l'on souhaite connaître toutes les années dans lesquelles un livre a été publié. La requête

```
SELECT annee FROM livre;
```

nous donne toutes les années, mais la même année peut apparaître plusieurs fois. Si on souhaite retirer les doublons d'un résultat, le mot clé `DISTINCT` peut être ajouté à la clause `SELECT`.

```
SELECT DISTINCT annee FROM livre;
```

Attention cependant, chaque ligne entière de résultat est considérée lors de la comparaison. C'est pourquoi, la requête

```
SELECT DISTINCT annee, isbn FROM livre;
```

continuera d'afficher plusieurs fois la même année. En effet, comme l'isbn est unique pour chaque ligne, tous les couples annee, isbn de la table sont différents deux à deux (ils diffèrent par leur isbn même s ils ont la même année). Le mot clé `DISTINCT` n'aura donc ici aucun effet.

Jointure

Les requêtes que nous avons vues nous permettent assez facilement de déterminer les livres qui ont été empruntés. Ces derniers sont simplement ceux dont l'ISBN est présent dans la table emprunt.

```
SELECT * FROM emprunt;
```

code_barre	isbn	retour
421921003090881	978-2081358881	2020-04-28
421921003090881	978-2207249123	2020-04-28
421921003090881	978-2824709420	2020-04-28
137332830764072	978-2352879183	2020-02-20
137332830764072	978-2335008586	2020-02-20
137332830764072	978-2013230827	2020-02-20
533299198788609	978-2253174561	2020-02-28
533299198788609	978-2251013039	2020-02-28
917547585216771	978-2290105504	2020-04-07
654834075188732	978-2864973270	2020-02-17
654834075188732	978-2070406340	2020-02-17
654834075188732	978-2806231697	2020-02-17
934701281931582	978-2260019183	2020-01-01
934701281931582	978-2371240087	2020-01-01
035184062854281	978-2745989857	2020-02-18
035184062854281	978-2072762093	2020-02-18
035184062854281	978-2742744824	2020-02-18

Cette réponse n'est cependant pas très satisfaisante. En effet, il serait plus naturel de pouvoir afficher les titres de ces livres plutôt que leur ISBN. Le problème est que les titres des livres sont présents uniquement dans la table livre. L'opération de jointure de deux tables apporte une réponse à ce problème. Étant données deux tables A et B, la jointure consiste à créer toutes combinaisons de lignes de A et de B ayant un attribut de même valeur. Ici, on souhaiterait obtenir une « grande table » dont les colonnes sont celles de la table emprunt et celle de la table livre, en réunissant les lignes ayant le même ISBN. Cela peut être fait au moyen de la directive `JOIN`.

```
SELECT * FROM emprunt JOIN livre ON emprunt.isbn = livre.isbn;
```

Cette requête crée la jointure des deux tables, représentée table ci-après. Comme on peut le voir, toutes les colonnes des deux tables ont été recopiées dans la sortie. Chaque ligne est le résultat de la fusion de deux lignes ayant le même ISBN. Le choix de ces lignes est donné par la condition de jointure indiquée par le mot clé ON. La condition indique au SGBD dans quel cas deux lignes doivent être fusionnées. Ici, on joint les lignes pour lesquelles les ISBN sont égaux. On écrit donc l'expression booléenne « `emprunt.isbn = livre.isbn` ». La notation « `nom_de_table.attribut` » permet de différencier deux attributs portant le même nom selon leur provenance. La jointure peut être combinée avec les clauses `SELECT` et `WHERE`. Par exemple, si on souhaite afficher uniquement les titres et les dates des livres empruntés qui sont à rendre avant le 1er février 2020, on peut écrire la requête suivante :

```
SELECT livre.titre, emprunt.retour FROM emprunt
JOIN livre ON emprunt.isbn = livre.isbn
WHERE emprunt.retour < '2020-02-01';
```

titre	retour
La Planète des singes	2020-01-01
Anna Karénine	2020-01-01

Même s'il n'y a pas d'ambiguïté ici, une bonne pratique consiste à préfixer les noms d'attributs par leur table dès que l'on utilise plus d'une table dans la requête. On n'est évidemment pas limité à une seule jointure. Si on souhaite afficher les noms et prénoms des utilisateurs ayant emprunté ces livres, il suffit de joindre la table usager, en rajoutant une nouvelle clause `JOIN ON`, cette fois sur le `code_barre` de l'usager.

```
SELECT usager.nom, usager.prénom, livre.titre, emprunt.retour FROM emprunt
JOIN livre ON emprunt.isbn = livre.isbn
JOIN usager ON usager.code_barre = emprunt.code_barre
WHERE emprunt.retour < '2020-02-01';
```

nom	prénom	titre	retour
PETIT	SÉBASTIEN	La Planète des singes	2020-01-01
PETIT	SÉBASTIEN	Anna Karénine	2020-01-01

La requête ci-dessus fonctionne parfaitement mais est un peu fastidieuse à écrire. Il est possible de créer dans une requête un alias pour un nom de table au moyen du mot clé `AS`, comme pour le renommage de colonne. La requête peut donc être réécrite de la manière suivante :

```
SELECT u.nom, u.prénom, l.titre, e.retour
FROM emprunt AS e
JOIN livre AS l ON e.isbn = l.isbn
JOIN usager AS u ON u.code_barre = e.code_barre
WHERE e.retour < '2020-02-01';
```

La jointure est une opération fondamentale des bases de données relationnelles. En effet, comme nous l'avons vu au chapitre **Modèle relationnel**, la modélisation relationnelle des données impose parfois un découpage des données. Les relations entre ces dernières sont maintenues par des contraintes, notamment les contraintes de référence. La jointure permet de reconstituer ce lien, en construisant « à la volée » de grandes tables contenant toutes les informations liées.

B. Modification des données

Les données stockées dans un SGBD ne sont a priori pas figées et peuvent être modifiées au cours du temps. Nous allons montrer deux types de modifications pouvant être faites sur les tables : la suppression d'un ensemble de lignes et la mise à jour de certains attributs d'un ensemble de lignes.

Suppression de lignes

L'ordre `DELETE FROM` *t* `WHERE` *c* permet de supprimer de la table *t* toutes les lignes vérifiant la condition *c*. Dans l'exemple de notre médiathèque, supposons que l'utilisateur Sébastien Petit, dont le `code_barre` est `'934701281931582'`, ait rendu ses livres. Il faut supprimer de la table emprunt toutes les lignes pour lesquelles le `code_barre` vaut `'934701281931582'`, ce qui donne l'ordre suivant :

```
DELETE FROM emprunt WHERE code_barre = '934701281931582';
```

Après exécution de cet ordre, la recherche dans la table emprunt ne donne plus de résultats :

```
SELECT COUNT(*) AS total
FROM emprunt
WHERE code_barre = '934701281931582';
```

total
0

Attention, au même titre qu'une requête `SELECT` sans clause `WHERE` sélectionne toutes les lignes, un ordre `DELETE` sans clause `WHERE` efface toutes les lignes de la table. Il ne faut pas confondre « `DELETE FROM` *t* » et « `DROP TABLE` *t* ». La première opération vide une table de son contenu, mais ne supprime pas la table. Il est donc possible d'y ajouter de nouveau des données au moyen de l'instruction `INSERT`. La seconde opération détruit la table (et ses données). La table ne peut donc plus être référencée.

Comme nous l'avons dit précédemment, les contraintes sont vérifiées à chaque mise à jour. Essayons de supprimer le livre *Hacker's delight* de la table `livre`, sachant que l'ISBN de ce dernier est `'978-0201914658'`.

```
# DELETE FROM livre WHERE isbn = '978-0201914658';
ERROR: update or delete on table "livre" violates foreign
       key constraint "auteur_de_isbn_fkey" on table
       "auteur_de"
```

Ici, le SGBD nous indique que supprimer ce livre (et donc supprimer sa clé primaire de la table livre) violerait la contrainte de clé étrangère dans la table `auteur_de`. Comme pour la destruction d'une table, il faut donc supprimer en premier les lignes dont les attributs sont déclarés comme clés étrangères avant de supprimer celles contenant les clés primaires correspondantes.

Du point de vue de leur exécution, les ordres de modification de table sont soit entièrement exécutés, soit entièrement annulés. Considérons la requête suivante :

```
DELETE FROM usager WHERE cp = '75001' OR cp = '75002' ;
```

qui efface de la table usager toutes les personnes dont le code postal est 75001 ou 75002. Si aucune de ces personnes n'apparaît dans la table emprunt, alors les suppressions peuvent être effectuées sans erreur. Supposons maintenant que certaines de ces personnes ont emprunté un livre. Même si le SGDB rencontre en premier des personnes sans emprunt et les supprime, il lèvera une erreur dès qu'il rencontrera un usager référencé dans la table emprunt. Dans ce cas, toutes les modifications déjà faites seront annulées et la table se trouvera dans l'état qu'elle avait avant la tentative d'exécution. Les exécutions sont donc de type « tout ou rien ».

Mise à jour

Le second type de modification est la mise à jour. Elle consiste à remplacer certains attributs d'un ensemble de lignes par de nouvelles valeurs. La syntaxe est la suivante :

$$\texttt{UPDATE } t \texttt{ SET } a_i = e_i, \ldots, \texttt{SET } a_n = e_n \texttt{ WHERE } c$$

Cette dernière signifie « sélectionne dans la table t toutes les lignes vérifiant la condition c et, pour chacune de ces lignes, remplace la valeur courante de l'attribut a_i par la valeur de l'expression e_i ». Par exemple, si l'utilisateur Sébastien Petit souhaite mettre à jour son adresse email, on écrit ceci :

```
UPDATE utilisateur SET email = 'spetit@hmail.com'
WHERE code.barre = '934701281931582';
```

Les expressions de mise à jour peuvent mentionner des noms d'attributs. Ces derniers sont alors remplacés par la valeur courante (avant mise à jour) de ces attributs. Supposons par exemple que la médiathèque soit fermée au mois d'avril. On souhaite que tous les emprunts dont la date de rendu était en avril soient prolongés de 30 jours.

```
UPDATE emprunt SET retour = retour + 30
WHERE retour >= '2020-04-01';
```

Dans la mise à jour précédente, la clause « `SET retour = retour + 30` » est similaire à la modification d'une variable dans un langage de programmation comme Python, c'est-à-dire prendre la valeur courante de `retour`, y ajouter 30 et écrire la nouvelle valeur dans `retour`.

Copie de table

Toute modification (création de table, suppression de table, mise à jour, suppression de ligne, insertion de ligne) qui ne viole pas de contrainte est définitive. En cas de suppression ou de mise à jour, les anciennes données sont perdues. Il faut donc être particulièrement vigilant lors de la conception d'un programme effectuant des mises à jour dans une base de données. Les bonnes pratiques recommandent l'utilisation de plusieurs « copies » de la base de données. Une copie de test, utilisée pour le développement et une copie de « production » utilisée pour faire fonctionner le logiciel, une fois que ce dernier a été testé rigoureusement. La base de production doit par ailleurs être sauvegardée fréquemment pour éviter les risques liés à de mauvaises manipulations ou à des défaillances logicielles ou matérielles.

Comme nous l'avons vu, l'ordre « `SELECT ... FROM` » renvoie une table comme résultat. Il est possible de nommer cette dernière grâce au mot-clé `INTO` :

```
SELECT * INTO usager_paris FROM usager WHERE cp LIKE '75%';
```

Cet ordre crée une nouvelle table nommée `usager_paris` contenant le résultat de la requête. Cette dernière a le même schéma que la table usager car toutes les colonnes ont été copiées. En revanche, elle ne contiendra que les lignes pour lesquelles le code postal commence par 75. La table `usager_paris` est ensuite utilisable comme n'importe quelle autre table. La clause `WHERE` permet de choisir les lignes à conserver. Par exemple,

```
SELECT * INTO usager2 FROM usager;
```

crée une copie conforme de `usager` alors que

```
SELECT * INTO usager2 FROM usager WHERE 1 = 0;
```

crée une table vide ayant le même schéma que `usager` car la condition « 1 = 0 » est toujours fausse.

Attention cependant, l'opération `SELECT ... INTO` ne copie pas les contraintes. Ainsi, dans la table usager2 ci-dessus, la colonne `code_barre`, bien qu'elle existe, n'est pas déclarée en tant que clé primaire. L'opération `SELECT INTO` servira donc plutôt à sauvegarder un résultat temporaire de requête plutôt que créer une véritable copie.

Créer une copie conforme d'une table peut se faire en utilisant deux variations sur des opérations que nous connaissons bien. La première est l'opération `CREATE`, utilsée comme ceci :

```
CREATE TABLE usager3 (LIKE usager INCLUDING ALL);
```

Cette syntaxe indique de créer la table usager3 comme une table de même schéma que usager, en incluant les contraintes (clés primaires, étrangères, CHECK, NOT NULL, ...).

L'inconvénient est que la table usager3 est vide. On peut cependant la remplir en utilisant une variation de l'order INSERT :

```
INSERT INTO usager3 (SELECT * FROM usager);
```

Ici, on dit d'insérer dans usager3 toutes les lignes renvoyées par la requête mise entre parenthèses. Cette dernière peut être arbitraire, mais doit renvoyer des lignes du même schéma que celles attendues pour la table usager3 (en particulier on aurait pu utiliser une clause WHERE);

C. Requêtes imbriquées

L'opération « SELECT ... INTO » permet de sauver le résultat d'une requête sous un certain nom de table. Il est donc possible d'effectuer sur ce résultat une nouvelle requête. Cependant, cette opération va occuper de l'espace de stockage. Il ne faudra donc pas oublier de supprimer la table ainsi créée. Il est possible de créer une table de manière temporaire et d'exécuter une requête sur cette table en imbriquant la première requête dans la clause « FROM » de la seconde ou dans une clause « JOIN ... ON » :

```
SELECT * FROM (SELECT * FROM livre
                       WHERE année >= 1990) AS tmp
WHERE tmp.année <= 2000;
```

La requête ci-dessus calcule d'abord une table intermédiaire nommée tmp qui liste les livres publiés après 1990. Suite à quoi, la table tmp est refiltrée pour ne garder que les livres pour lesquels l'année est inférieure à 2000. Attention, il ne s'agit ici que d'une explication de « haut niveau ». En pratique, n'importe quel SGBD moderne évaluera cette deux requêtes imbriquées comme la requête équivalente :

```
SELECT * FROM livre WHERE annee >= 1990 AND annee <= 2000;
```

Une autre manière d'imbriquer les requêtes consiste à utiliser une sous-requête dans la clause WHERE. En effet, le langage SQL identifie les valeurs scalaires et les tables à une seule « case » telles que celles renvoyées par les fonctions d'agrégation. Par exemple, si on souhaite afficher les titres des livres dont l'année est la plus ancienne dans la base, on pourra écrire :

```
SELECT titre FROM livre WHERE annee = (SELECT MIN(annee) FROM livre);
```

Ici, la sous-requête calcule l'année minimum de la table livre (1933 dans notre base), puis affiche tous les titres de livres dont l'année vaut 1933. Attention, la sous-requête ne doit pas nécessairement comporter une fonction d'agrégation. Il suffit qu'elle renvoie une table contenant une seule valeur.

Ainsi, si nous voulons afficher les titres des livres publiés la même année que *Moby Dick* (sans connaître cette année), nous pouvons écrire :

```
SELECT titre FROM livre WHERE annee =
       (SELECT annee FROM livre WHERE titre = 'Moby Dick');
```

titre
Moby Dick
L'île des morts
Le Devin
Le Berceau du chat
Les Enfants de minuit
À la recherche du temps perdu

Mais attention, si la sous-requête renvoie plusieurs résultats, le SGBD renverra une erreur :

```
# SELECT titre FROM livre WHERE annee =
         (SELECT année FROM livre WHERE titre LIKE '%Astérix%');

ERROR: more than one row returned by a subquery
       used as an expression
```

Un opérateur utilisant la puissance des requêtes imbriquées est l'opérateur `IN`. L'expression e `IN` (q) renvoie vrai si et seulement, si la valeur résultant de l'évaluation de e est l'une des lignes renvoyées par la requête q. Ainsi, pour exprimer la requête « afficher les titres des livres qui ont été publiés la même année qu'un livre dont le titre contient Astérix », on pourra écrire

```
SELECT titre FROM livre WHERE annee IN
       (SELECT annee FROM livre WHERE titre LIKE '%Astérix%')
```