Bases de données – Le language SQL

I) Fonctionnalités attendues d'un SGBD

Un SGBD doit permettre la création, la modification et l'accès aux données de manière sécurisée.
Plus précisément :

- Création de la base de données
- Création des relations (tables) de la base
    - Avec respect des attributs et contraintes d'intégrité
    - Avec respect des contraintes de cohérence
- Suppression de données dans les tables, ou de tables en entier
- Mise à jour des données
- Interrogation des données avec un langage spécifique de requêtes, SQL étant le plus connu
- Sécurité des données :
    - tous les utilisateurs n'ont pas accès à toutes les données
    - respect des règles de cohérence, notamment lorsque plusieurs requêtes sont effectuées à la suite
    - pas de modification simultanée des mêmes données par plusieurs utilisateurs
    - conservation des données en cas de panne matérielle
- Tout ceci en étant transparent vis-à-vis de la structure des données. L'utilisateur – ou le programme - faisant une requête n'a pas besoin de savoir comment sont organisées les données en mémoire ;

Propriétés ACID

Source Wikipédia
En informatique, les propriétés **ACID** (atomicité, cohérence, isolation et durabilité) sont un ensemble de propriétés qui garantissent qu'une transaction informatique, comme une opération sur les données d'une base de données, est exécutée de façon fiable.

- **Atomicité** : cette propriété assure qu'une transaction se fait au complet ou pas du tout ;
- **Cohérence** : cette propriété assure que chaque transaction amènera le système d'un état valide à un autre état valide. En particulier les contraintes d'intégrité doivent être vérifiées (mais pas seulement) ;
- **Isolation** : les transactions s'exécutent comme si elles étaient seules sur le système. Si par exemple une transaction complexe T1 s'exécute en même temps qu'une autre transaction complexe T2, alors T1 ne peut pas accéder à un état intermédiaire de T2. T1 et T2 doivent produire le même résultat qu'elles soient exécutées simultanément ou successivement.
- **Durabilité** : les transactions sont « gravées dans le marbre ». Une fois qu'une transaction a eu lieu, la base de données reste dans l'état modifié, même suite à une panne d'électricité…

II) Le langage SQL

SQL (*Structured Query Language*) est un langage permettant d'exploiter les bases de données relationnelles.
Le langage SQL permet de rechercher, d'ajouter, de modifier ou de supprimer des données dans les bases de données relationnelles.
1974 : première version de SQL.
En juin 1970, *Edgar Frank Codd* publia un article sur un référentiel de données relationnel, rapidement reconnu comme un modèle théorique intéressant pour l'interrogation des bases de données, et qui inspira le développement, au début des années 70, du langage *Structured English QUEry Language* (SEQUEL), renommé **SQL** en 1975.
En 1979, *Relational Software, Inc.* (actuellement *Oracle Corporation*) présenta la première version commercialement disponible de SQL, rapidement imité par d'autres fournisseurs.
En 1986, SQL a été normalisé par l'Institut américain (ANSI), puis en 1987 par l'organisme internationale ISO.

SQL un langage **déclaratif**, c'est-à-dire qu'il permet de décrire le résultat escompté sans décrire la manière de l'obtenir.

L'interaction avec un SGBD se fait par l'envoi d'une suite d'instructions SQL.
Celles-ci s'écrivent d'une manière qui ressemble à celle de phrases ordinaires en anglais. Cette ressemblance voulue vise à faciliter l'apprentissage et la lecture.

Exemple :

```
CREATE TABLE Ville (
  ID INTEGER PRIMARY KEY AUTOINCREMENT,
  Nom CHAR(35) NOT NULL DEFAULT '',
  CodePays CHAR(3) NOT NULL DEFAULT '',
  Circonscription CHAR(25) NOT NULL DEFAULT '',
  Population INTEGER NOT NULL DEFAULT '0'
);
```

On distingue en général :

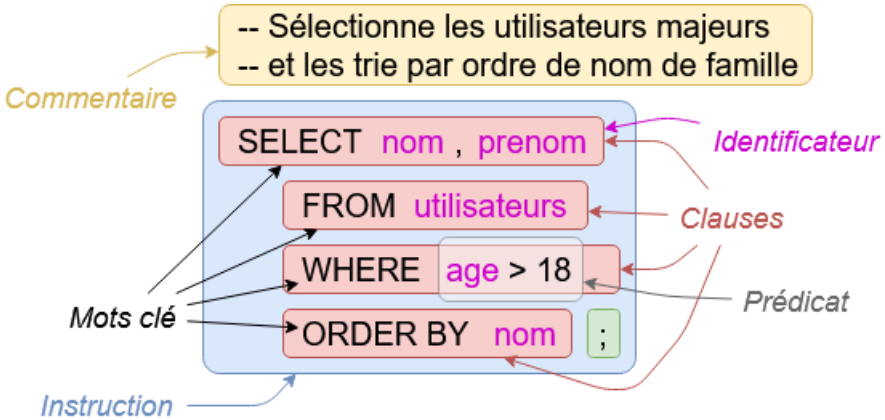
- Les **instructions**, qui peuvent avoir un effet persistant sur les schémas et les données, permettent de contrôler les transactions, le déroulement du programme, les connexions, les sessions ou les diagnostics ;
- Les **requêtes**, qui sont des instructions qui récupèrent les données sur la base de critères spécifiques.

Caractéristiques des instructions et requêtes :

Une instruction SQL peut s'étendre sur plusieurs lignes et se termine par le caractère « ; ».
Les espaces, tabulations, et saut de ligne ne sont pas significatifs.
SQL est insensible à la casse, mais il est d'usage d'écrire les mots clés du langage en majuscules et d'écrire les noms des attributs en minuscules (les noms des attributs ne pouvant pas contenir d'espaces, on peut utiliser le caractère de soulignement _).

Les instructions SQL sont constituées ainsi :

- Des **clauses**, qui sont des éléments constitutifs des déclarations et des requêtes (dans certains cas, elles sont facultatives).
    - des **mots clés** sont des mots qui sont définis dans le langage SQL.
    - des **identificateurs** sont des noms sur les objets de la base de données, comme les *tables*, les *colonnes* et les *schémas*.
      Un identificateur ne peut pas être égal à un mot-clé réservé, sauf s'il est placé entre guillemets (identificateur délimité).
    - des **expressions**, qui peuvent produire soit des valeurs scalaires, soit des tableaux composés de colonnes et de lignes de données
    - des **prédicats**, qui spécifient les *conditions* pouvant être évaluées selon la **logique à trois valeurs** (*3VL – three-valued logic*) (vrai/faux/inconnu) ou bien la **logique booléenne**
      Ils sont utilisés pour limiter les effets des énoncés et des requêtes, ou pour modifier le déroulement du programme.
- Un **terminateur** d'instruction, le point-virgule « ; ».



1) Création d'une table

Strictement parlant, une relation du modèle relationnel et une table SQL ne sont pas des concepts équivalents.
En particulier une table SQL peut contenir des doublons et il n'est pas obligatoire de spécifier une clé primaire lors de sa création.
En SQL, les attributs d'une table sont appelés des *colonnes* et les entités des *lignes*.
On commencera par lister les différents types de données possibles pour les colonnes d'une table.

Types de données en SQL

Types numériques

Nom du type	Exact/approché	Description
SMALLINT	exact	entier 16 bits signé
INTEGER	exact	entier 32 bits signé
INT	exact	alias pour INTEGER
BIGINT	exact	entier 64 bits signé
DECIMAL(t, f)	exact	décimal signé de t chiffres dont f décimales
REAL	approché	flottant 32 bits
DOUBLE PRECISION	approché	flottant 64 bits

On notera le type DECIMAL(t, f) qui permet de représenter de manière exacte un nombre à virgule flottante de taille donnée. Ce type est important pour représenter des attributs portant sur des valeurs monétaires.
Par exemple DECIMAL$(5,2)$ peut être utilisé pour stocker des valeurs décimales comprises entre $-999{,}99$ et $999{,}99$.

Type booléen

Le type BOOLEAN est inégalement supporté par les différentes implémentations du langage SQL. Le standard SQL laisse ce type comme optionnel.
Une alternative possible est d'utiliser CHAR(1) avec les valeurs 'T' et 'F' ou encore SMALLINT avec 1 pour True et 0 pour False.

Types textes

Nom du type	Description
CHAR(n)	Chaîne d'exactement n caractères (complétée à droite par les espaces le cas échéant)
VARCHAR(n)	Chaîne d'au plus n caractères
TEXT	Chaîne de taille quelconque, sans taille maximale $a\ priori$

Le type CHAR(n) est approprié si l'on souhaite stocker des chaînes de taille fixe et connue. Dans les requêtes SQL, les chaînes de caractères seront délimitées par des guillemets simple. Le caractère guillemet peut être échappé en le doublant (ex : 'l''attribut').

Types dates et durées

Nom du type	Description
DATE	Une date au format 'AAAA-MM-JJ'
TIME	Une heure au format 'hh :mm :ss'
TIMESTAMP	Un instant (date et heure) au format 'AAAA-MM-JJ hh :mm :ss'

Les valeurs de ces types s'écrivent comme de simples chaînes de caractères.
Il est possible d'utiliser l'addition pour ajouter des jours à une valeur de type DATE et il y a de nombreuses fonctions permettant de manipuler des dates. CURRENT_DATE représente la date du jour.

La valeur NULL

La valeur NULL représente l'absence de valeur. Ce n'est pas un type. Elle peut être utilisée à la place de n'importe quelle autre valeur, quel que soit le type attendu (peut être assimilé à None en Python).

SQL interdit l'utilisation de NULL comme valeur pour une clé primaire. Elle est en revanche autorisée pour les clés étrangères, ce qui se traduit par une violation de la contrainte de référence.
Dans une requête, on peut tester si un attribut d'une table à la valeur NULL au moyen des expressions IS NULL ou IS NOT NULL.

Spécification des contraintes d'intégrité

a) Clé primaire

Les mots clés PRIMARY KEY permettent d'indiquer qu'un attribut est une clé primaire.

Exemple :

```
CREATE TABLE personne(
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   nom VARCHAR(99),
   prenom VARCHAR(99),
   age INTEGER
);
```

Si on souhaite utiliser plusieurs attributs comme clé primaire, on peut spécifier la contrainte après les attributs.

Exemple :

```
CREATE TABLE point (
  x INTEGER NOT NULL,
  y INTEGER NOT NULL,
  PRIMARY KEY (x,y)
);
```

b) Clé étrangère

Un attribut peut être qualifié de clé étrangère en utilisant le mot clé REFERENCE suivi du nom de la table où se trouve la clé primaire et de son nom. **Il est à noter que la plupart des SGBD ne supportent pas l'utilisation de clés étrangères composites, ce qui amoindrit l'intérêt de l'utilisation de clés primaires composites**.

Exemple :

```
CREATE TABLE employe(
   idemp INTEGER NOT NULL UNIQUE REFERENCES personne(id),
   service VARCHAR(99),
   email VARCHAR(60) NOT NULL UNIQUE,
   idsup INTEGER REFERENCES personne(id)
);

DROP TABLE IF EXISTS employe;
CREATE TABLE employe(
   idemp INTEGER NOT NULL UNIQUE,
   service VARCHAR(99),
   email VARCHAR(60) NOT NULL UNIQUE,
   idsup INTEGER,
   FOREIGN KEY (idemp) REFERENCES personne(id),
   FOREIGN KEY (idsup) REFERENCES personne(id)
);
```

c) Unicité et non nullité

Il peut être intéressant de spécifier qu'un attribut est unique sans pour autant en faire une clé primaire. Cela peut être spécifié au moyen du mot clé UNIQUE.
Une autre bonne pratique consiste à déclarer qu'un attribut ne peut pas être NULL (notons que PRIMARY KEY implique automatiquement NOT NULL).

d) Contraintes utilisateur

Il est possible de spécifier des contraintes sur les attributs au moyen du mot clé CHECK suivi d'une formule booléenne.
Cette contrainte peut être placée sur la même ligne que la ligne de déclaration d'un attribut ou en fin de déclaration de la table si elle porte sur plusieurs attributs.

Exemple :

```sql
CREATE TABLE personne(
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   nom VARCHAR(99),
   prenom VARCHAR(99),
   sexe CHAR(1) CHECK( sexe IN ('F','M') ) DEFAULT 'F',
   age INTEGER NOT NULL CHECK(age>=18 AND age<=99),
   CHECK (nom NOT NULL OR prenom NOT NULL)
);
```

2) Suppression d'une table

Une fois une table créée, il n'est pas possible d'en créer une autre de même nom.
Il est possible de supprimer une table à l'aide de l'instruction DROP TABLE.

```sql
DROP TABLE personne;
```

L'instruction ci-dessus provoquera une erreur si la table n'existe pas. Il est possible d'écrire

```sql
DROP TABLE IF EXISTS personne;
```

pour éviter qu'une telle erreur ne se produise.
La suppression d'une table entraîne la suppression de toutes les données qu'elle contient.

Attention :

Il n'est pas possible de supprimer une table si elle sert de référence pour une clé étrangère d'une autre table car cela violerait la contrainte de référence.
Il convient alors de supprimer les tables dans le bon ordre, c'est-à-dire en commençant par les tables contenant les clés étrangères, avant les tables contenant les clés primaires référencées.

Certains SGBD permettent de spécifier que la suppression d'une table doit détruire automatiquement toutes les tables qui dépendent d'elle.

3) Insertion dans une table

L'insertion dans une table se fait à l'aide de l'instruction INSERT INTO.
Le mot clé INSERT INTO doit être suivi du nom de la table puis du mot-clé VALUES suivi d'une suite de n-uplets, chacun entre parenthèses. Chaque n-uplet représente une nouvelle ligne de la table.

Exemple :

```sql
INSERT INTO personne VALUES
  (1,'Dupond','Jean','M',45),
  (2,'Mercier','Pierre','M',45);
```

Les valeurs des attributs sont supposées être dans le même ordre que lors de l'instruction CREATE TABLE. Il est possible de les passer dans un ordre différent ou encore de ne passer que certaines valeurs. Dans ce cas, il faut spécifier le nom et l'ordre des attributs avant le mot clé VALUES.

Exemple :

```sql
INSERT INTO personne (prenom,nom,sexe,age) VALUES
  ('Jean','Dupond','M',45),
  ('Pierre','Mercier','M',45);
```

Les contraintes d'intégrité sont exécutées au moment d'une insertion et une instruction INSERT INTO qui violerait ces contraintes conduira à une erreur. Les données ne seront dans ce cas pas ajoutées à la table.

Exemple :

Dans la table personne, id est une clé primaire. L'insertion suivante

```sql
INSERT INTO personne (id, prenom,nom,sexe,age) VALUES
  (1,'Jean','Dupond','M',45),
  (1,'Pierre','Mercier','M',45);
```

provoquera une erreur :

```
UNIQUE constraint failed: personne.id
```

4) Sélection de données

Nous avons vu comment créer des tables et les remplir avec des données cohérentes vis-à-vis des contraintes d'intégrité. Nous allons voir maintenant comment récupérer les données des tables d'une base de données.
La sélection va consister en l'écriture de *requêtes* SQL permettant de trouver toutes les données de la base vérifiant certains critères.
On utilise pour cela l'instruction **SELECT**. Celle-ci comporte de nombreuses clauses, qui permettent d'extraire et de réaliser certaines opérations sur les données récupérées, qui nous seront retournées sous la forme d'une table.
Dans cette partie, on ne s'intéresse qu'à la syntaxe des requêtes SELECT. Nous verrons ultérieurement dans quel cadre ces requêtes peuvent être utilisées et les données récupérées traitées.

La projection

Le mot clé SELECT permet de désigner la liste des champs de la table pour lesquels on souhaite récupérer les valeurs. L'astérisque * est utilisé pour désigner la totalité des champs.
Le mot clé FROM indique le nom de la table contenant les champs. Il ne faut pas oublier le ';' à la fin de l'instruction.
Dans une instruction SELECT, la clause FROM est obligatoire.

Exemple :

On considère la relation :
personne(id, nom, prenom, sexe, age)

```
SELECT * FROM personne;
SELECT nom, prenom FROM personne;
```

La première requête va sélectionner tous les champs de la table et retourner tous les enregistrements, c'est-à-dire les valeurs des champs pour toutes les personnes de la table.
La seconde requête va sélectionner les seuls champs *nom* et *prenom* de la table et retourner tous les enregistrements avec les valeurs pour ces seuls deux champs.
L'opération qui consiste à ne retenir que certains champs d'une table s'appelle une projection.

Remarque :

Lorsque le nom d'un élément d'une base de données (table, colonne par exemple) est identique à un mot clef du SQL, il convient de l'entourer de guillemets (double quote). En principe, les mots réservés du SQL sont déconseillés pour nommer des objets du modèle physique de données...

La sélection

La sélection consiste à ne choisir que certaines lignes de la table.
On utilise pour cela la clause WHERE suivie du ou des critères de sélection.
Un critère de sélection est une expression booléenne, qui peut être construite à partir d'opérateurs de comparaison <, <=, >, >=, = et <> (ou !=), d'opérateurs arithmétiques (+, -, *, / et %) , de constantes (comme PI), de noms d'attributs, d'opérateurs logiques (AND, OR, NOT), d'opérateurs spéciaux (comme LIKE, BETWEEN, IN, IS NULL, IS NOT NULL), etc..
La liste est longue !

Exemple :

```
SELECT nom, prenom
FROM personne
WHERE age >=18 AND age<= 25 AND sexe='F';
```

Cette requête permet de récupérer les nom et prénom des personnes de plus de 18 ans et de moins de 25 ans, de sexe féminin.

Attention : on remarque que le test d'égalité se fait avec l'opérateur = et non avec l'opérateur ==.

*Le mot clé **AS***

Le résultat d'une requête est retourné sous la forme d'une table, ici avec deux colonnes. Il est possible de renommer ces colonnes à l'aide du mot clé **AS**.

Exemple :

```
SELECT nom AS le_nom, prenom AS le_prenom
FROM personne
WHERE age>=18 AND age<=25 and sexe='F';
```

*L'opérateur **LIKE***

Pour effectuer des recherches sur les chaînes de caractères, on dispose des symboles :

- % qui représente une chaîne de caractère quelconque, éventuellement vide ;
- _ (le caractère de soulignement), qui représente un caractère quelconque.

Une chaîne de caractères incluant de tels symboles est un motif. **Pour tester si une chaîne vérifie un motif, on utilise l'opérateur LIKE**.

Exemple :

```
SELECT nom, prenom
FROM personne
WHERE nom LIKE 'M%';
```

L'opérateur **LIKE** s'évalue à vrai si et seulement si la chaîne de caractères représentant le nom correspond au motif 'M%', c'est-à-dire si le nom commence par la lettre M.

Attention : le symbole % ne sera pas interprété si on utilise l'opérateur = pour effectuer la comparaison.

*L'opérateur **IS NULL***

Il retourne la valeur booléenne TRUE si l'opérande est NULL et FALSE sinon.
L'opérateur IS NOT NULL retourne le résultat contraire.

Exemple :

```
SELECT nom, prenom
FROM personne
WHERE age IS NOT NULL;
```

*L'opérateur **BETWEEN***

L'opérateur BETWEEN permet de rechercher si une valeur se trouve dans un intervalle donné, quel que soit le type des valeurs de référence spécifiées (alpha, numérique, date...).

Exemple :

```
SELECT *
FROM personne
WHERE nom BETWEEN 'M' AND 'N';
```

*L'opérateur **IN***

L'opérateur IN permet de tester si une valeur appartient à une liste.

Exemple :

```
SELECT *
FROM personne
WHERE age IN (10,20,30,40);
```

La suppression des doublons

Le mot clé DISTINCT permet de retirer les doublons d'un résultat d'une requête SELECT.

Exemple :

```
SELECT DISTINCT prenom
FROM personne;
```

La requête ci-dessus permet d'obtenir la liste des prénoms des personnes de la table, sans doublon.

Attention :

Si on précise plusieurs champs, c'est la combinaison de ces champs qui sera évaluée.

Exemple :

```
SELECT DISTINCT nom, prenom
FROM personne;
```

Ici, il n'y aura pas de personne avec le même nom et prénom dans les résultats de la requête. Mais il pourra y avoir des doublons sur les prénoms ou les noms.

Le tri

A priori les résultats renvoyés par une commande SELECT ne sont pas classés. Ils peuvent être présentés dans n'importe quel ordre. Il faut considérer une table comme un ensemble d'enregistrements qui n'est pas ordonnée. Le tri va permettre de classer les enregistrements en fonction d'un ou plusieurs critères. Pour cela on utilise la clause ORDER. Il y a deux possibilités de tri, par ordre croissant (tri par défaut) ou décroissant.

Exemple :

```
-- tri croissant par nom
SELECT nom, prenom
FROM personne
ORDER BY nom;

-- tri croissant par nom+prenom
SELECT nom, prenom
FROM personne
ORDER BY nom, prenom ASC;

-- tri décroissant par nom+prenom
SELECT nom, prenom
FROM personne
ORDER BY nom, prenom DESC;
```

On utilise le mot clé ASC pour effectuer un tri croissant et le mot clé DESC pour effectuer un tri décroissant.

Les agrégats

Dans la clause SELECT, il est possible d'utiliser des fonctions d'agrégation. Ces dernières permettent d'appliquer une fonction à l'ensemble des valeurs d'une colonne et de retourner le résultat comme une table ayant une seule case. On citera quelque unes de ces fonctions :

COUNT :

La fonction COUNT sert à compter des enregistrements. Le résultat de l'opération est affiché dans un nouveau champ.

Exemple :

```
SELECT COUNT(nom)
FROM personne
WHERE age>=18 AND age<=25;
```

COUNT(nom)
26

Le résultat sera une table pour laquelle en général on utilisera AS pour renommer la colonne contenant le résultat. La fonction COUNT compte ici le nombre d'enregistrements vérifiant la clause WHERE et elle retournera le même résultat avec un autre champ. On utilisera donc plutôt la syntaxe suivante :

Exemple :

```
SELECT COUNT(*) AS Nombre
FROM personne
WHERE age>=18 AND age<=25;
```

SUM :

Cette fonction sert à additionner les valeurs d'un champ numérique.

Sa syntaxe est la même que celle de COUNT.

AVG :

Cette fonction sert à calculer la moyenne (average en anglais) des valeurs d'un champ numérique.
Sa syntaxe est également la même que celle de COUNT.

MIN et MAX :

Ces fonctions recherchent la plus petite et la plus grande valeur d'un champ, en utilisant la comparaison sur son type.

Exemple :

```
SELECT MIN(age) AS minimum,
       MAX(age) AS maximum,
       SUM(age) AS somme,
       AVG(age) as moyenne
FROM personne
WHERE age>=18 AND age<=25;
```

Les champs calculés

Il est possible d'obtenir le résultat de calculs sur les champs.

```
SELECT (note1*2+note2)/3 AS moyenne
FROM notes;
```

Dans cet exemple, on calcule une moyenne à partir des notes obtenues.

La jointure

La jointure permet de mettre en relation plusieurs tables, par l'intermédiaire des liens qui existent en particulier entre la clé primaire de l'une et la clé étrangère de l'autre.
La jointure est une opération de sélection qui permet de ne retenir que les enregistrements pour lesquels la valeur de la clé primaire d'une table correspond à la valeur de la clé étrangère d'une autre table.
C'est une opération importante, qui permet de récupérer en une seule requête des données issues de plusieurs tables.
On considère le modèle relationnel suivant, utilisé pour permettre l'emprunt de livres dans une médiathèque :

Emprunteur(code_barre : string, nom : string, prenom : string, adresse : string, cp : string, commune : string)
Livre(isbn : string, titre : string, editeur : string, annee : integer)
Auteur(ida : integer, nom : string, prenom : string)
AEcrit(#ida : integer, #isbn : string)
Emprunt(#isbn : string, #code_barre : string, date_retour : date)

Un emprunteur est identifié par un code barre, un livre par son code isbn et un auteur par un id quelconque.
La table Emprunt identifie un livre par son code isbn et un emprunteur par son code barre, ce qui n'est pas très parlant.
Si on souhaite afficher le titre des livres empruntés, il faut réaliser la jointure des tables Emprunt et Livre, à partir du code isbn. On écrira :

```
SELECT * FROM Emprunt
  JOIN Livre ON Emprunt.isbn=Livre.isbn;
```

Cette requête affichera, pour chaque enregistrement de la table Emprunt, tous les champs de la table Emprunt, ainsi que tous les champs de la table Livre avec le code isbn correspondant.
La condition de jointure est indiquée par le mot clé ON. Elle indique selon quel critère les tables doivent être fusionnées.
Les tables Emprunt et Livre ont toutes les deux un champ nommé isbn. Pour les distinguer, il est nécessaire de mentionner le nom de la table avant celui du champ.
La requête ci-dessus comportera deux colonnes nommées isbn (une pour chaque table). Ce code isbn ne présente pas forcément d'intérêt et a essentiellement été utilisé pour permettre la jointure.

Si on souhaite simplement connaître les titres des livres à retourner avant le 25/11/2020, on écrira :

```
SELECT Livre.titre, Emprunt.date_retour
  FROM Emprunt
  JOIN Livre ON Emprunt.isbn=Livre.isbn,
  WHERE date_retour<='2020-11-25';
```

Il est possible de réaliser des jointures impliquant plusieurs tables.
La requête suivante

```
SELECT Livre.titre, Emprunt.date_retour, Emprunteur.nom
  FROM Emprunt
  JOIN Livre ON Emprunt.isbn=Livre.isbn,
  JOIN Emprunteur ON Emprunt.code_barre=Emprunteur.code_barre,
  WHERE date_retour<='2020-11-25';
```

retourne la liste des livres empruntés pour lesquels la date de retour est antérieure au 25/11/2020, et affiche pour chacun d'eux son titre, la date de retour ainsi que le nom de l'emprunteur.
La jointure permet d'agréger les données qui ont été découpées dans la modélisation relationnelle. Grâce aux jointures, il est possible de reconstituer « à la volée » de grandes tables contenant toutes les informations.

Remarque :

La jointure est équivalente à une sélection sur le produit cartésien des tables.
La requête

```
SELECT Emprunt.date_retour, Livre.titre
  FROM Emprunt, Livre;
```

effectue en effet le produit cartésien des deux tables Emprunt et Livre.
Si la table Emprunt a n enregistrements et la table Livre m enregistrements, le produit cartésien des deux tables a $n \times m$ enregistrements. A chaque enregistrement de la table Emprunt correspondent m enregistrements de la table résultat du produit cartésien des deux tables : chacun de ces enregistrements est constitué des valeurs de l'enregistrement de la table Emprunt auxquelles on ajoute les valeurs d'un enregistrement de la table Livre.
La requête

```
SELECT Livre.titre, Emprunt.date_retour
  FROM Emprunt
  JOIN Livre ON Emprunt.isbn=Livre.isbn,
  WHERE date_retour<='2020-11-25';
```

est alors équivalente à la requête

```
SELECT Emprunt.date_retour, Livre.titre
  FROM Emprunt, Livre
  WHERE Emprunt.isbn=Livre.isbn AND Emprunt.date_retour<='2020-11-25';
```

La seconde version est l'ancienne syntaxe utilisée avant l'apparition du mot clé JOIN. Même si les deux syntaxes sont équivalentes en performance, la bonne pratique nous fera privilégier la première version.

Remarque :

Il existe en fait plusieurs types de jointures. La jointure utilisée ici est dite jointure interne. C'est le type de jointure le plus couramment utilisé et qui s'exerce par défaut si on ne précise pas le type de la jointure.
La requête

```
SELECT Livre.titre, Emprunt.date_retour
  FROM Emprunt
  JOIN Livre ON Emprunt.isbn=Livre.isbn,
  WHERE date_retour<='2020-11-25';
```

est ainsi équivalente à la requête

```
SELECT Livre.titre, Emprunt.date_retour
  FROM Emprunt
  INNER JOIN Livre ON Emprunt.isbn=Livre.isbn
  WHERE date_retour<='2020-11-25';
```

Une jointure interne permet de sélectionner les enregistrements ayant des correspondances entre les deux fichiers joints. Les enregistrements pour lesquels il n'y a pas de correspondance seront ignorés.

Dans l'exemple ci-dessus, la clé étrangère `isbn` de la table `Emprunt` a été définie comme clé primaire, de telle sorte qu'il ne peut y avoir qu'une seule ligne dans le résultat de la requête réalisant cette association. Cette unicité n'est cependant pas obligatoire.

Considérons le modèle relationnel suivant :

Client(id_client : int, nom : string, prenom : string, adresse : string, cp : string, commune : string)
Commande(id_commande : int, #id_client : int, date_commande : date, montant : decimal)

Dans ce schéma, un client passer plusieurs commandes.
Si je veux lister toutes les commandes en y associant les nom et prenom du client, je peux par exemple effectuer la requête suivante :

```
SELECT id_commande, nom, prenom, date_commande, montant
FROM Client
JOIN Commande ON Client.id_client=Commande.id_client;
```

Dans la table retournée par la commande SELECT, il y aura alors plusieurs lignes réalisant la condition de jointure.
La requête suivante est équivalent, et peut être plus intuitive :

```
SELECT id_commande, nom, prenom, date_commande, montant
FROM Commande
JOIN Client ON Client.id_client=Commande.id_client;
```

5) Les requêtes imbriquées

Dans le langage SQL, une requête imbriquée, ou sous-requête, est une requête exécutée à l'intérieur d'une autre requête. Il y a plusieurs façons d'utiliser les sous-requêtes.

Une requête SELECT permet d'obtenir des enregistrements dans un tableau de résultat qui peut être assimilé à une nouvelle table (on parlera plutôt de vue).

1er cas : la table retournée par la requête SELECT imbriquée contient une seule valeur.

Si la table retournée par la requête SELECT contient une seule « case », c'est-à-dire un seul enregistrement pour un seul champ, le langage SQL assimile la valeur du champ à une valeur scalaire utilisable dans une autre requête.

Supposons par exemple une table avec les notes des élèves pour un contrôle, suivant le schéma :

Eleves(id_eleve, nom, prenom, note, observation)

Pour connaître les élèves ayant une note inférieure de 2 points à la moyenne, on peut écrire :

```
SELECT nom, prenom
  FROM Eleves
  WHERE note<(SELECT AVG(note)-2 FROM Eleves);
```

Dans cet exemple, la sous requête calcule la valeur correspondant à la note moyenne moins 2 points. Le résultat est retourné sous la forme d'une table ayant une seule valeur, laquelle peut être utilisée dans la clause WHERE d'une requête SELECT. Les parenthèses sont exigées pour une syntaxe correcte.

Si on désire connaître le nom des élèves ayant eu la même note qu'un élève donné, identifié par son `id_eleve`, on écrira :

```
SELECT nom, prenom
  FROM Eleves
  WHERE note=(SELECT note FROM Eleves WHERE id_eleve=13);
```

Ici, il n'y a pas d'ambiguïté car la requête imbriquée retourne une seule valeur. Un message d'erreur sera affiché si la requête retourne plusieurs valeurs.

Une requête imbriquée est souvent utilisée au sein d'une clause WHERE pour remplacer une ou plusieurs constantes.

2ème cas : la table retournée par la requête SELECT imbriquée contient plusieurs valeurs pour un seul champ.

Dans ce cas, il est possible d'utiliser le résultat de la requête imbriquée avec l'opérateur IN de la clause WHERE d'une requête SELECT. Les valeurs de la table retournée par la requête imbriquée seront interprétées comme une liste de valeurs.

Si je veux connaître la moyenne les élèves ayant eu une note supérieure à celle d'un élève donné identifié par son `id_eleve` (requête probablement sans intérêt mais donnée pour exemple) je dois :

- en premier récupérer la liste des élèves ayant une note supérieure à l'élève ;
- puis calculer la moyenne des notes sur cette liste.

Pour cela, je peux écrire :

```
SELECT AVG(note) AS moyenne
  FROM Eleves
  WHERE id_eleve in (
    SELECT id_eleve
    FROM Eleves
    WHERE note>(SELECT note FROM Eleves WHERE id_eleve=13));
```

3ème cas : requête SELECT effectuée sur la table retournée par la requête SELECT imbriquée.

Supposons une première requête qui sélectionne les enregistrements de la table correspondant à une note inférieure à 15. Je peux utiliser le résultat de cette requête comme une table, à condition de lui donner un alias, et sélectionner sur cette table les enregistrements correspondants à une note supérieure à 5.

```
SELECT * FROM (SELECT * FROM Eleves WHERE note<15) AS tmp
  WHERE tmp.note>5;
```

Il aurait été plus simple d'écrire directement :

```
SELECT * FROM Eleves
  WHERE note>5 AND note<15;
```

6) La mise à jour des données

Nous avons vu comment insérer des enregistrements dans une table avec l'utilisation du mot clé INSERT.
Il est bien sûr possible de modifier les valeurs des champs d'un enregistrement ou de le supprimer.

a) La mise à jour des enregistrements

La mise à jour des données s'effectue à l'aide d'une requête UPDATE.
La requête UPDATE permet de remplacer les valeurs d'un ou plusieurs champs pour une ou plusieurs lignes.
Cette requête est très souvent associée à une clause WHERE qui précise la ou les lignes sur lesquelles doivent porter la modification.

Syntaxe :

```
UPDATE nom_de_la_table
  SET liste_des_modifications
  WHERE condition;
```

La clause WHERE est optionnelle. Sans cette clause, la modification concerne toutes les lignes de la table.

Exemple :

```
UPDATE Eleves
  SET observation='Abs'
  WHERE note=NULL;
```

Cette requête modifie la valeur du champ observation pour les élèves avec la valeur du champ `note` égale à `NULL`. Si aucun élève ne correspond à ce critère, aucune modification ne sera effectuée et la requête sera sans effet.
La même remarque s'applique à la requête suivante

```
UPDATE Eleves
  SET note=14, observation='Bien compris'
  WHERE id_eleve=13;
```

qui suppose la présence dans la table d'un enregistrement pour l'élève identifié par son id.

La clause WHERE est optionnelle.

Exemple :

```
UPDATE Eleves
  SET note=MIN(20,note+2);
```

Cette requête modifie toutes les lignes de la table.

b) La suppression des enregistrements

Pour supprimer des lignes d'une table, il faut utiliser la requête DELETE.

```
DELETE FROM nom_de_la_table
  WHERE condition;
```

Si aucune clause WHERE n'est mentionnée, toutes les lignes de la table seront supprimées. Mais la table elle-même ne sera pas supprimée pour autant.

Exemple :

```
DELETE FROM Eleves
  WHERE id_eleve=13;
```

Cette requête supprime de la table l'élève identifié par son id.

```
DELETE FROM Eleves;
```

Cette requête supprime tous les enregistrements de la table.

Attention :

Les contraintes de référence sont vérifiées le cas échéant par le SGBD. Si on essaie de supprimer un enregistrement contenant une clé primaire référencée comme clé étrangère dans une autre table, un message d'erreur sera retourné par le SGBD.

Si on reprend le schéma suivant,

Livre(isbn : string, titre : string, editeur : string, annee : integer)
Emprunt(#isbn : string, #code_barre : string, date_retour : date)

il n'est pas possible de supprimer un livre de la table Livre s'il a été emprunté.

Dans une requête DELETE, les ordres de modifications de la table sont soit tous exécutés, soit tous annulés. Considérons la requête suivante :

```
DELETE FROM Livre
  WHERE annee=1990;
```

Si un seul livre de l'année 1990 a été emprunté, la requête ne sera simplement pas exécutée. Les livres de l'année 1990 non empruntés ne seront alors pas retirés de la table.

7) Opérations sur les tables

Nous avons vu qu'une requête SELECT retourne des lignes dans un tableau de résultat qui peut être assimilé à une nouvelle table. Il est possible d'enregistrer cette nouvelle table à l'aide de la clause INTO.

Exemple :

```
SELECT * INTO Annee90 FROM Livre
  WHERE annee=1990;
```

Cette requête génère une nouvelle table, nommée `Annee90`, qui aura le même schéma que la table `Livre`, mais ne contiendra que les enregistrements de l'année 1990.

```
SELECT isbn, titre, editeur INTO Annee90 FROM Livre
  WHERE annee=1990;
```

Cette requête génère une nouvelle table, nommée `Annee90`, qui ne contient que les champs sélectionnés, avec toujours un filtrage sur l'année de parution du livre.

Pour créer une table vide suivant le même schéma que la table Livre, on peut écrire :

```
SELECT * INTO Livre2 FROM Livre
  WHERE 1=0;
```

La condition étant toujours fausse, aucun enregistrement ne sera copié. Mais tous les champs de la table `Livre` seront bien créés dans la table `Livre2`.

Attention :

La requête SELECT … INTO ne copie pas les contraintes de la table d'origine dans la table créée. Elle doit donc être essentiellement utilisée à des fins de sauvegarde des données.
Dans l'exemple ci-dessus, le champ `isbn` est une clé primaire de la table `Livre`, mais pas de la table `Livre2`.

Pour créer une table comme copie d'une autre en respectant ses contraintes, il faut utiliser la commande CREATE comme suit :

```
CREATE TABLE Livre2 (LIKE Livre INCLUDING ALL);
```

La table ainsi créée est cependant vide. On peut la remplir en utilisant une variante de la requête INSERT :

```
INSERT INTO Livre2 (SELECT * FROM Livre);
```

Dans la clause SELECT imbriquée, il est possible d'utiliser une clause WHERE.