Le langage SQL

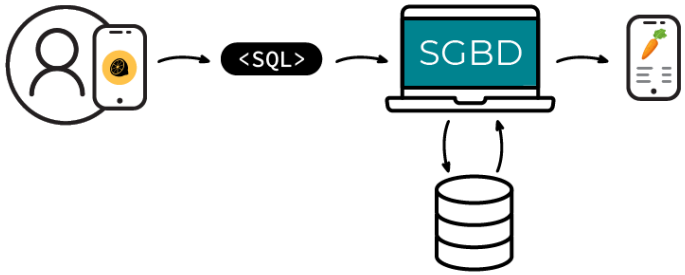
Introduction

On a vu que le modèle relationnel permettait de représenter la structure des données d'une base de données, mais aucune considération informatique n'entrait en jeu (c'était plutôt une vision mathématique de la base de données).

Le modèle relationnel est réalisé par des logiciels appelés *systèmes de gestion de bases de données*, abgrégé SGBD. Les SGBD relationnels sont les SGBD qui utilisent le modèle relationnel pour la représentation des données (on avait dit qu'il y en avait d'autres).

La grande majorité des SGBD relationnels utilisent le langage SQL (*Structured Query Language*) qui permet d'envoyer des *ordres*, appelés requêtes, au SGBD. Ces ordres sont de deux types :

- les *requêtes d'interrogation* permettent de récupérer des données vérifiant certains critères ;
- les *requêtes de mise à jour* permettent de modifier la base de données



Source : Cours Implémentez vos bdd relationnelles avec SQL de Quentin Durantay sur OpenClassrooms

On a vu que le modèle relationnel définissait différentes *contraintes d'intégrité*, et le SGBD est garant du respect de ces contraintes. Concrètement, c'est lui qui empêchera d'effectuer des modifications ne respectant pas les contraintes.

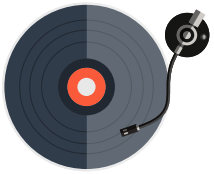
Les SGBD relationnels les plus utilisés sont Oracle, MySQL, Microsoft SQL Server, PostgreSQL, Microsoft Access, SQLite, MariaDB.

> Il existe de plus en plus de SGBD *non relationnels*, spécialement adaptés à des données plus diverses et moins structurées. On les appelle souvent des SGBD NoSQL (pour *Not only SQL*) : citons par exemple MongoDB, Cassandra (Facebook), BigTable (Google), ...

La très grande majorité des SGBD sont basés sur un modèle client-serveur, nécessitant le démarrage d'un serveur pour effectuer les requêtes. Ce n'est pas le cas du SGBD SQLite car la base de données peut être représentée dans un fichier indépendant de la plateforme. Cette particularité rendant les choses plus simples fera que nous utiliserons le SGBD SQLite dans ce chapitre.

> Faites l'activité d'introduction

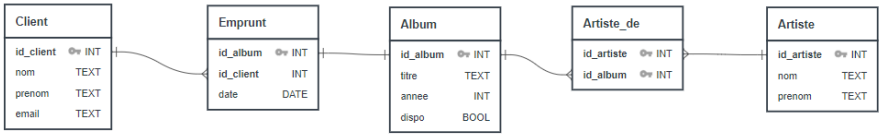
Le retour du disquaire



On suppose dans la suite de ce chapitre que l'on travaille avec la base de données de notre disquaire (voir Chapitre 1). Le schéma de la base est le suivant :

```
Album(id_album INT, titre TEXT, annee INT, dispo BOOL)

Artiste(id_artiste INT, nom TEXT, prenom TEXT)

Artiste_de(#id_artiste INT, #id_album INT)

Client(id_client INT, nom TEXT, prenom TEXT, email TEXT)

Emprunt(#id_client INT, #id_album INT, jour DATE)
```

Voici le diagramme représentant ce schéma :



Réalisé avec l'application quickdatabasediagrams.com

> Remarque importante : comme SQLite ne gère pas les booléens, l'attribut `dispo` de la table `Album` est représenté à la place par un entier : `1` pour vrai et `0` pour faux. D'ailleurs, vous verrez dans les exercices qu'il n'était pas utile de conserver cet attribut car on peut retrouver les albums empruntés en croisant les tables.

Sélectionner des données

On commence par voir les requêtes SQL permettant de récupérer des données selon certains critères, on appelle cela les requêtes d'interrogation.

> Faites l'exercice 1.

Sélectionner des colonnes avec `SELECT`

Première requête SQL

La requête suivante permet d'afficher tout le contenu de la table `Eleve`.

```
SELECT * FROM Album;
```

Analyse :

- En SQL, chaque requête contient au moins les clauses `SELECT` et `FROM` et se termine par un point-virgule `;`.
- Le mot-clé `SELECT` demande au SGBD d'afficher ce que contient une table.
- Après `SELECT`, il faut indiquer quels champs (ou *attributs*) de la table, le SGBD doit récupérer dans celle-ci. Ici le caractère « `*` » indique qu'il faut récupérer tous les champs de la table.
- Après le mot clé `FROM` (de l'anglais « de ») on indique la table dans laquelle on veut récupérer des informations (ici `Album`)

Bilan : cette requête se traduit par « prend tout ce qu'il y a dans la table `Album` », sous-entendu prend tous les champs de cette table. Elle produit le résultat suivant :

id_album	titre	annee	dispo
1	Blues Breakers	1966	1
2	I Still Do	2016	1
3	Aftermath	1966	1
4	Off the Wall	1979	1
5	Axis : Bold As Love	1967	0
6	Thriller	1982	0
7	Black and Blue	1976	1
8	Riding With The King	2000	0
9	Bad	1987	1
10	It's Only Rock'n Roll	1974	1
11	Don't Explain	2011	1
12	Aretha	1980	1
13	Abbey Road	1969	1
14	Let It Be	1970	0
15	44/876	2018	0
16	Lady Soul	1968	0
17	Back in Black	1980	1
18	Sacred Love	2003	1
19	Songs in the Key of Life	1976	0
20	Power Up	2020	0
21	The Last Ship	2013	1
22	Signed, Sealed & Delivered	1970	1
23	Fire on the Floor	2016	0
24	Continuum	2006	0
25	Continuum	2006	0

26	Exodus	1977	1
27	Sex Machine	1970	1
28	T.N.T.	1975	1
29	Leave the Light On	2003	1
30	Blues Deluxe	2003	1

Et si on ne veut pas toutes les colonnes ?

En SQL, il est possible de sélectionner certaines colonnes de la table (et pas toutes) simplement en indiquant après le `SELECT`, les noms des attributs à conserver.

Par exemple, la requête

```
SELECT titre, annee FROM Album;
```

permet de ne sélectionner que les attributs `titre` et `annee` de la table `Album` (on dit que l'on fait une projection sur ces deux attributs) :

titre	annee
Blues Breakers	1966
I Still Do	2016
Aftermath	1966
Off the Wall	1979
Axis : Bold As Love	1967
Thriller	1982
Black and Blue	1976
Riding With The King	2000
Bad	1987
It's Only Rock'n Roll	1974
Don't Explain	2011
Aretha	1980
Abbey Road	1969
Let It Be	1970
44/876	2018
Lady Soul	1968
Back in Black	1980
Sacred Love	2003
Songs in the Key of Life	1976
Power Up	2020
The Last Ship	2013
Signed, Sealed & Delivered	1970
Fire on the Floor	2016
Continuum	2006
Continuum	2006

Exodus	1977
Sex Machine	1970
T.N.T.	1975
Leave the Light On	2003
Blues Deluxe	2003

Sélectionner des lignes avec `WHERE`

En plus de sélectionner des colonnes, on peut sélectionner certaines lignes en utilisant la clause `WHERE` suivie de la condition de sélection.

Par exemple, la requête

```
SELECT titre, annee FROM Album WHERE annee >= 2005;
```

permet d'obtenir les titres et l'année de sortie des albums (de la table `Album`) qui sont sortis en 2005 ou après :

titre	annee
I Still Do	2016
Don't Explain	2011
44/876	2018
Power Up	2020
The Last Ship	2013
Fire on the Floor	2016
Continuum	2006
Continuum	2006

Remarque : L'expression se trouvant après `WHERE` est une *expression booléenne* qui peut être construite à partir :

- des opérateurs de comparaison : `<`, `<=`, `>`, `>=`, `=` et `<>` (ou `!=` qui est généralement supporté par les SGBD)
- des opérateurs arithmétiques : `+`, `-`, `*`, `/`, `%`
- de constantes et noms d'attributs
- d'opérateurs logiques : `AND`, `OR`, `NOT`
- et d'opérateur spéciaux tels que l'opérateur de comparaison de texte `LIKE` (voir un peu plus loin)

Combiner les conditions avec les opérateurs logiques

Par exemple, pour obtenir les titres et années des albums sortis entre 2005 et 2015, on peut écrire la requête

```
SELECT titre, annee FROM Album WHERE annee >= 2005 AND annee <= 2015
```

qui produit le résultat suivant :

titre	annee
Don't Explain	2011
The Last Ship	2013
Continuum	2006
Continuum	2006

Remarque : Le langage SQL n'est pas sensible aux blancs et aux indentations (ni à la casse), ce qui fait que lorsque les requêtes commencent à être un peu longues, on peut améliorer leur lisibilité en jouant sur les retours à la ligne et les indentations. Ainsi, la requête qui suit est équivalente à la précédente :

```
SELECT titre, annee
FROM Album
WHERE annee >= 2005 AND annee <= 2015;
```

Rechercher par motif avec `LIKE`

On peut effectuer des requêtes effectuant des recherches de certains motifs en utilisant `LIKE`.

Par exemple, on peut chercher les identifiants et les titres des albums dont le titre contient le mot `"Love"`. La requête s'écrirait

```
SELECT id_album, titre FROM Album WHERE titre LIKE "%Love%";
```

et produit le résultat :

id_album	titre
5	Axis : Bold As Love
18	Sacred Love

Analyse :

- Contrairement au `=` qui fait une recherche exacte, l'opération `titre LIKE "%Love%"` effectue une recherche approchée. Ainsi, `titre LIKE "%Love%""` est évaluée à vrai si et seulement si l'attribut `titre` correspond au motif `"%Love%"`.
- Dans un motif le symbole `%` est un *joker* et peut être substitué par n'importe quelle chaîne de caractères.

Trier les données avec `ORDER BY`

On peut trier des données en utilisant `ORDER BY` à la fin d'une requête, suivi de l'attribut à trier et de `ASC` (pour un tri croissant) ou `DESC` (pour un tri décroissant).

Ainsi, la requête

```
SELECT titre, annee
FROM Album
WHERE annee >= 2005 AND annee <= 2015
ORDER BY annee ASC;
```

permet de trier les résultats d'une des requêtes précédente par ordre chronologique d'année de sortie.

titre	annee
Continuum	2006
Continuum	2006
Don't Explain	2011
The Last Ship	2013

En remplaçant `ASC` par `DESC` on aurait obtenu les mêmes résultats mais affichés dans l'ordre inverse, du plus récent au plus ancien.

Supprimer les doublons avec `DISTINCT`

On voit que le résultat de la dernière requête contient 4 enregistrements, dont deux sont identiques (pour les deux attributs conservés !). On peut utiliser le mot clé `DISTINCT` avec la clause `SELECT` pour retirer les doublons d'un résultat :

```
SELECT DISTINCT titre, annee
FROM Album
WHERE annee >= 2005 AND annee <= 2015
ORDER BY annee ASC;
```

titre	annee
Continuum	2006
Don't Explain	2011
The Last Ship	2013

Faire des calculs grâce aux fonctions d'agrégation

Les *fonctions d'agrégation* permettent d'appliquer une fonction à toutes les valeurs d'une colonne et renvoyer le résultat comme une table ayant une seule case (une ligne et une colonne). Voici quelques fonctions d'agrégation :

- `COUNT()` : pour compter le nombre de résultats (le nombre de colonnes)
- `AVG()` : pour calculer la moyenne des valeurs d'une colonne
- `SUM()` : pour calculer la somme des valeurs d'une colonne
- `MIN()` et `MAX()` : pour calculer respectivement la valeur minimale et la valeur maximale d'une colonne

Compter avec `COUNT()`

Par exemple, pour calculer le nombre d'albums sortis entre 2005 et 2015, (plutôt que de renvoyer ces albums en question), on écrira la requête :

```
SELECT COUNT(*) AS total
FROM Album
WHERE annee >= 2005 AND annee <= 2015;
```

qui renvoie le résultat

total
4

Remarque : On a choisi ici de renommer `total` la colonne donnant le résultat de la requête. En effet, sinon le SGBD choisi lui-même un nom, souvent peu parlant, puisque le résultat n'est pas une colonne d'une table existante.

Trouver le minimum et le maximum avec `MIN()` et `MAX()`

Si on souhaite connaître l'année de l'album le plus ancien du disquaire, il suffit de calculer la valeur minimale de l'attribut `annee` avec la requête

```
SELECT MIN(annee) AS annee_mini FROM Album;
```

qui renvoie le résultat :

annee_mini
1966

Ces fonctions peuvent également comparer des chaînes de caractères. Ainsi, si on souhaite connaître le nom de l'artiste arrivant en dernier par ordre alphabétique, il suffit de "calculer" la valeur maximale de l'attribut `nom` (de la table `Artiste`) avec la requête

```
SELECT MAX(nom), prenom FROM Artiste;
```

qui renvoie le résultat :

MAX(nom)	prenom
Wonder	Stevie

> Ici, on a utilisé la fonction `MAX()` (sur le `nom`) tout en sélectionnant l'attribut `prenom` pour récupérer également le prénom de l'artiste. Ce n'était pas utile et on aurait pu écrire `SELECT MAX(nom) FROM Artiste;`

Les fonctions `AVG()` et `SUM()` s'utilisent de la même manière mais n'ont pas de sens avec les données présentes dans la base de données du disquaire, donc on n'en parle pas ici.

Recherches croisées : les jointures avec `JOIN`

> Faites les exercices 2 et 3.

Les requêtes abordées jusqu'à présent ne portaient à chaque fois que sur une seule table. C'est malheureusement insuffisant pour chercher certaines informations qui nécessitent de *croiser* (les informations de) plusieurs tables.

Imaginons que l'on veuille connaître les clients ayant des emprunts en cours. Ces derniers sont ceux présents dans la table `Emprunt` et on peut les obtenir avec la requête

```
SELECT * FROM Emprunt;
```

qui produit la réponse suivante :

id_client	id_album	jour
1	5	2021-09-10
3	8	2021-08-18
3	24	2021-08-18
5	25	2021-09-12
5	6	2021-10-10
9	20	2021-09-28
11	14	2021-10-08
7	15	2021-10-08
7	19	2021-10-08
7	16	2021-10-15
16	29	2021-10-01

Mais ce n'est pas très satisfaisant car on aimerait plutôt afficher les noms, prénoms et adresse email de ces clients plutot que `id_client`.

Le problème est que les noms, prénoms, adresses email sont uniquement présents dans la table `Client`. Il est nécessaire de faire une jointure entre les deux tables `Emprunt` et `Client`.

Première jointure

Une jointure consiste à créer toutes les combinaisons de lignes des deux tables ayant un attribut de même valeur (l'attribut `id_client` dans notre exemple). Pour effectuer une jointure, on utilise la clause `JOIN`. Une jointure consiste à créer toutes les combinaisons de lignes des deux tables ayant un attribut de même valeur qui est précisé après le mot clé `ON`.

Ainsi, dans le cas de notre exemple, on peut effectuer la jointure entre les tables `Emprunt` et `Client`, sur l'attribut `id_client` pour ne garder que les lignes concernant le même client.

Cela s'écrit avec la requête suivante.

```
SELECT *
FROM Emprunt
JOIN Client ON Emprunt.id_client = Client.id_client;
```

Le résultat de cette jointure est :

id_client	id_album	jour	id_client_1	nom	prenom	
1	5	2021-09-10	1	Dupont	Florine	dupont.florir
3	8	2021-08-18	3	Marchand	Grégoire	greg.marchan
3	24	2021-08-18	3	Marchand	Grégoire	greg.marchan
5	25	2021-09-12	5	Pacot	Jean	jp
5	6	2021-10-10	5	Pacot	Jean	jp
9	20	2021-09-28	9	Dubois	Philippe	pdubo
11	14	2021-10-08	11	Fournier	Marie	m
7	15	2021-10-08	7	Moreau	Alain	a
7	19	2021-10-08	7	Moreau	Alain	a
7	16	2021-10-15	7	Moreau	Alain	a
16	29	2021-10-01	16	Bernardin	Stéphanie	sberna

Analyse :

- La jointure (`SELECT * FROM Emprunt JOIN Client`) a permis de recopier toutes les colonnes des deux tables.
- Le choix des lignes à conserver, appelée *condition de jointure* , suit le mot clé `ON`. Cela permet de fusionner uniquement les lignes vérifiant la condition `Emprunt.id_client = Client.id_client`, autrement dit les lignes pour lesquelle l'attribut `id_client` est identique donc celles concernant un même client.

  > Essayez d'enlever le `ON ...`, vous constaterez que toutes les lignes sont fusionnées, ce qui est absurde car une même ligne peut alors correspondre à deux clients distincts.

- Vous avez constaté que l'on a préfixé chaque attribut par le nom de la table auquel il appartient. Cela permet de faire la différence entre deux attributs portant le même nom dans deux tables différentes, et c'est une bonne pratique de toujours le faire même lorsqu'il n'y a pas d'ambiguïté.

> Ce sont les clés étrangères qui permettent de faire le lien entre les tables, il est donc normal que la condition de jointure fasse intervenir `id_client` (puisque c'est une clé étrangère de la table `Emprunt` qui la lie à la table `Client`).

On peut combiner une jointure avec la clause `SELECT` pour n'afficher que ce qui nous intéresse. Par exemple, si on ne veut que les noms, prénoms et adresses email des clients ayant des emprunts en cours ainsi que les albums empruntés, on peut faire la requête

```
SELECT Emprunt.id_album, Client.nom, Client.prenom, Client.email
FROM Emprunt
JOIN Client ON Emprunt.id_client = Client.id_client;
```

qui produit le résultat

id_album	jour	nom	prenom	email
5	2021-09-10	Dupont	Florine	dupont.florine@domaine.net
8	2021-08-18	Marchand	Grégoire	greg.marchand49@music.com
24	2021-08-18	Marchand	Grégoire	greg.marchand49@music.com
25	2021-09-12	Pacot	Jean	jpacot@music.com
6	2021-10-10	Pacot	Jean	jpacot@music.com
20	2021-09-28	Dubois	Philippe	pdubois5@chezmoi.net
14	2021-10-08	Fournier	Marie	mfournier@abc.de
15	2021-10-08	Moreau	Alain	amoreau1@abc.de
19	2021-10-08	Moreau	Alain	amoreau1@abc.de
16	2021-10-15	Moreau	Alain	amoreau1@abc.de
29	2021-10-01	Bernardin	Stéphanie	sbernard1@chezmoi.net

Combiner les jointures

Plutôt que d'affichier l' `id_album`, qui est peu lisible, on peut préférer afficher le titre de l'album. Mais pour récupérer cette information dans la table `Album`, il faut une nouvelle jointure :

```
SELECT Album.titre, Emprunt.jour, Client.nom, Client.prenom, Client.
FROM Emprunt
JOIN Client ON Emprunt.id_client = Client.id_client
JOIN Album ON Emprunt.id_album = Album.id_album;
```

Analyse : On a ajouté la dernière ligne qui permet de faire une jointure sur l'attribut `id_album` entre la table produite par la requête précédente et la table `Album`. Et on a remplacé la première colonne `Emprunt.id_album` par `Album.titre` pour faire apparaître les titres des albums comme souhaité :

titre	jour	nom	prenom	email
Axis : Bold As Love	2021-09-10	Dupont	Florine	dupont.florine@domaine.net
Riding With The King	2021-08-18	Marchand	Grégoire	greg.marchand49@music.com
Continuum	2021-08-18	Marchand	Grégoire	greg.marchand49@music.com
Continuum	2021-09-12	Pacot	Jean	jpacot@music.com
Thriller	2021-10-10	Pacot	Jean	jpacot@music.com
Power Up	2021-09-28	Dubois	Philippe	pdubois5@chezmoi.net
Let It Be	2021-10-08	Fournier	Marie	mfournier@abc.de
44/876	2021-10-08	Moreau	Alain	amoreau1@abc.de
Songs in the Key of Life	2021-10-08	Moreau	Alain	amoreau1@abc.de
Lady Soul	2021-10-15	Moreau	Alain	amoreau1@abc.de
Leave the Light On	2021-10-01	Bernardin	Stéphanie	sbernard1@chezmoi.net

On peut combiner les jointures avec tout ce qui a été vu précédemment, par exemple ajouter des conditions, trier, etc.

La requête

```
SELECT Album.titre, Emprunt.jour, Client.nom, Client.prenom, Client.
FROM Emprunt
JOIN Client ON Emprunt.id_client = Client.id_client
JOIN Album ON Emprunt.id_album = Album.id_album
WHERE Emprunt.jour >= '2021-10-02'
ORDER BY Client.nom ASC;
```

permet de récupérer les mêmes informations qu'au-dessus mais seulement pour les emprunts à partir du 2 octobre 2021, les résultats étant triés par ordre alphabétique des noms des emprunteurs.

titre	jour	nom	prenom	email
Let It Be	2021-10-08	Fournier	Marie	mfournier@abc.de
44/876	2021-10-08	Moreau	Alain	amoreau1@abc.de
Songs in the Key of Life	2021-10-08	Moreau	Alain	amoreau1@abc.de
Lady Soul	2021-10-15	Moreau	Alain	amoreau1@abc.de
Thriller	2021-10-10	Pacot	Jean	jpacot@music.com

Utiliser des alias

Certaines requêtes peuvent commencer à être assez longues à écrire. Pour réduire leur longueur on peut utiliser des *alias* pour les noms de table grâce au mot clé `AS`.

Ainsi, la requête précédente peut aussi s'écrire

```
SELECT a.titre, e.jour, c.nom, c.prenom, c.email
FROM Emprunt AS e
JOIN Client AS c ON e.id_client = c.id_client
JOIN Album AS a ON e.id_album = a.id_album
WHERE e.jour >= '2021-10-02'
ORDER BY c.nom ASC;
```

Analyse : `Emprunt AS e` permet de renommer la table `Emprunt` par `e`, ce qui permet de raccourcir les écritures du type `Emprunt.id_client` en `e.id_client`. Idem pour `c` et `a` qui sont les alias respectifs des tables `Client` et `Album`.

■ Modifier des données


Faites les exercices 4 et 5.

Les données stockées dans une base de données n'ont pas vocation à être figées, elles peuvent être modifiées au cours du temps grâce à des requêtes de mise à jour de la base de donnnées.

Nous allons voir les requêtes permettant d'ajouter des données à une table, de modifier les données d'une table et de supprimer les données d'une table.

Avant cela, faisons une petite digression sur la création de tables dans une base de données.

Créer une table avec `CREATE TABLE`

> La création d'une base de données n'est pas au programme de Terminale NSI. Néanmoins, connaître les requêtes permettant de créer une table permet de mieux appréhender les requêtes de mise à jour que l'on verra ensuite.

Créer une base de données consiste à créer les tables de la base. Pour créer une table, on utilise `CREATE TABLE`.

Par exemple, pour créer la table `Artiste` correspondant à la relation suivante :

```
Artiste(id_artiste INT, nom TEXT, prenom TEXT)
```

on peut exécuter cette commande SQL :

```
CREATE TABLE Artiste (
    id_artiste INTEGER PRIMARY KEY,
    nom TEXT,
    prenom TEXT
);
```

Remarque : On a bien précisé le nom et le type de chaque attribut, et indiqué avec `PRIMARY KEY` quelle était notre clé primaire.

Insérer des données avec `INSERT INTO ... VALUES`

Supposons que l'on veuille insérer les 3 enregistrements suivants dans la table `Artiste`.

id_artiste	nom	prenom
1	Clapton	Éric
2	Mayall	John
3	Hendrix	Jimi

Pour cela, on peut écrire la requête SQL

```
INSERT INTO Artiste VALUES (1, 'Clapton', 'Éric'),
                           (2, 'Mayall', 'John'),
                           (3, 'Hendrix', 'Jimi');
```

Analyse :

- Après `INSERT INTO` (que l'on traduit par "insérer dans") on indique le nom de la table (ici `Note`) dans laquelle on veut insérer des données ;
- Puis on indique grâce au mot clé `VALUES` les enregistrements que l'on veut insérer, ces derniers étant séparés par des virgules s'il y en a plusieurs (on n'oublie pas le `;` pour terminer) ;
- Avec cette requête, les valeurs des différents enregistrements (ou n-uplets) doivent être données dans le même ordre que lors du `CREATE TABLE`. Néanmoins, il est possible de les passer dans un ordre différent comme on l'explique juste en-dessous.

Si on désire passer les valeurs des enregistrements dans un ordre différent de celui de la création de la table, il suffit de préciser l'ordre juste après le nom de la table :

```
INSERT INTO Artiste (prenom, nom, id_artiste) VALUES ('John', 'Mayer
```

On peut vérifier en affichant les 4 enregistrements ainsi insérés dans la table `Artiste` :

```
SELECT * FROM Artiste;
```

id_artiste	nom	prenom
1	Clapton	Éric
2	Mayall	John
3	Hendrix	Jimi
4	Mayer	John

Respect de la contrainte de relation

On rappelle que le SGBD est garant du respect des contraintes d'intégrité de la base (voir Chapitre 1). En particulier, de la contrainte de relation qui impose que chaque enregistrement d'une relation doit posséder une clé primaire *unique*.

Ainsi, si on essaie d'insérer un nouvel enregistrement avec une clé primaire existante, le SGBD n'acceptera pas l'insertion proposée en indiquant l'erreur :

```
INSERT INTO Artiste VALUES (2, 'Dylan', 'Bob');
```

```
UNIQUE constraint failed: Artiste.id_artiste
```

La base de données ne sera alors pas modifiée !

Remarque : Pour ne pas avoir à saisir nous-mêmes l'attribut `id_artiste` de chaque artiste, on aurait pu indiquer au SGBD d'utiliser le principe d'*auto-incrément* : dès qu'un nouvel enregistrement est inséré, `id_artiste` est incrémenté automatiquement d'une unité. Pour cela, la commande de création de la table aurait été :

```
CREATE TABLE Artiste (
    id_artiste INTEGER PRIMARY KEY AUTOINCREMENT,
    nom TEXT,
    prenom TEXT
);
```

et on aurait pu insérer les enregistrements sans préciser l'attribut `id_artiste` :

```
INSERT INTO Artiste (nom, prenom) VALUES ('Clapton', 'Éric'),
                                         ('Mayall', 'John'),
                                         ('Hendrix', 'Jimi');
```

On doit alors préciser que l'on ne saisit que les attributs `nom` et `prenom`.

Dans ce cas, l'insertion d'un nouvel enregistrement s'écrit

```
INSERT INTO Artiste (nom, prenom) VALUES ('Dylan', 'Bob');
```

et le SGBD détermine lui-même l'attribut `id_artiste` lors de l'insertion.

> Pour voir le code SQL qui a permi de créer la base de données du disquaire, vous pouvez suivre ce lien. Regardez notamment comment on a lié les tables avec les clés étrangères en utilisant `REFERENCES` et comment on définit la réunion de plusieurs attributs comme clé primaire. Vous constaterez également que l'on n'a pas utilisé le type `TEXT` pour le domaine des attributs de type "texte". Enfin, vous trouverez dans ce fichier .sql les ordres d'insertions des différents enregistrements de chaque table.

Modifier une valeur avec `UPDATE ... SET`

Il est possible de modifier des valeurs existantes dans une table, avec `UPDATE`.

Par exemple, le client

id_client	nom	prenom	email
4	Michel	Valérie	vmichel5@monmail.com

a changé d'adresse email. Pour modifier cette adresse dans la base de données, on peut écrire :

```
UPDATE Client SET email = 'valerie.michel@email.fr'
WHERE id_client = 4;
```

Analyse :

- Après `UPDATE` on indique le nom de la table dans laquelle on veut modifier une valeur (ici `Client`) ;
- Ensuite, on écrit `SET` puis une expression de la forme `attribut = valeur` qui permet de définir une nouvel valeur `valeur` pour l'attribut `attribut` (ici `'valerie.michel@email.fr'` est la nouvelle valeur de l'attribut `email`) ;
- Enfin, on précise avec `WHERE` la condition permettant de sélectionner les enregistrements sur lesquels la modification doit être apportée (ici une seule ligne car la condition `id_client = 4` ne correspond qu'à un seul enregistrement).

On peut vérifier que la modification a bien été faite :

```
SELECT * FROM Client
WHERE id_client = 4;
```

id_client	nom	prenom	email
4	Michel	Valérie	valerie.michel@email.fr

Supprimer un enregistrement avec `DELETE`

Il est possible de supprimer une ligne d'une table en utilisant `DELETE`.

Par exemple, le client Marchand Grégoire a rendu l'album `Continuum` (dont l'attribut `id_album` vaut 25) qu'il avait emprunté. Il faut supprimer la ligne correspondante dans la table `Emprunt` :

id_client	id_album	jour
1	5	2021-09-10
3	8	2021-08-18
3	24	2021-08-18
5	25	2021-09-12
5	6	2021-10-10
9	20	2021-09-28
11	14	2021-10-08
7	15	2021-10-08
7	19	2021-10-08
7	16	2021-10-15
16	29	2021-10-01

Pour cela, on peut écrire l'ordre suivant :

```
DELETE FROM Emprunt
WHERE id_album = 25;
```

Analyse :

- Après `DELETE` on indique dans quelle table on veut supprimer une ligne avec `FROM [nom_table]` ;
- Ensuite on précise avec `WHERE` la condition permettant de sélectionner les enregistrements à supprimer (ici une seule ligne est supprimée car la condition `id_album = 25` ne correspond qu'à un seul enregistrement).

On peut vérifier que la ligne correspondante a bien été supprimée de la table `Emprunt` :

```
SELECT * FROM Emprunt;
```

id_client	id_album	jour
1	5	2021-09-10
3	8	2021-08-18
3	24	2021-08-18
5	6	2021-10-10
9	20	2021-09-28
11	14	2021-10-08
7	15	2021-10-08
7	19	2021-10-08
7	16	2021-10-15
16	29	2021-10-01

Remarque : Avec le schéma de la base de données il faut aussi mettre à jour la table `Album` puisque l'album en question est à nouveau disponible. La requête de mise à jour suivante permet de faire cela :

```
UPDATE Album
SET dispo = 1
WHERE id_album = 25;
```

Respect des contraintes de référence

Le SGBD est garant du respect des *contraintes de référence* (voir Chapitre 1). L'une d'elles consiste à ne pas pouvoir supprimer un enregistrement si sa clé primaire est associée à des enregistrements liés dans d'autres tables (liés par une clé étrangère !).

Par exemple, si on essaie de supprimer de la relation `Client` le client "Dupont Florine", dont l'attribut `id_client` est `1`, le SGBD empêchera la suppression car ce client apparaît dans la relation `Emprunt` en tant que clé étrangère (si la suppression était effectuée, cette clé étrangère ne ferait plus référence à une clé primaire de la table `Client`, ce qui est impossible par définition d'une clé étrangère).

Ainsi, l'exécution de la requête

```
DELETE FROM Client
WHERE id_client = 1;
```

produit l'erreur suivante :

```
FOREIGN KEY constraint failed
```

Il est possible d'apporter d'autres types de modifications à une table, mais elles ne sont a priori pas exigibles en Terminale NSI. Voir par exemple les commandes `ALTER TABLE` (https://sql.sh/cours/alter-table) et `DROP TABLE` (https://sql.sh/cours/drop-table)

Faites les exercices 6 et 7.

Bilan

- Ce sont les logiciels de type SGBD qui jouent le rôle d'interface entre l'humain et la base de données. Il existe différentes sortes de SGBD, des gratuits, des payants, des libres, des propriétaires. On a utilisé le SGBD *SQLite* pour sa simplicité d'utilisation.
- On peut utiliser le langage SQL pour écrire des requêtes destinées à donner des ordres à la base de données. On distingue deux types de requêtes : les *requêtes d'interrogation* et les *requêtes de mise à jour*.
- Les requêtes d'interrogation permettent de trouver toutes les lignes d'une table vérifiant un certain critère booléen. Si les données à trouver sont réparties dans plusieurs tables, on utilise une jointure pour fusionner les tables.
- Les requêtes de mise à jour apportent des modifications aux données enregistrées. Cela peut être des insertions, des modifications ou des suppressions de lignes.
- Des opétateurs avancés permettent de *trier* les résultats selon un certain critère, de *supprimer les doublons* des résultats. Des *fonctions d'agrégation* permettent de faire des calculs sur les données d'une colonne.
- Une modification qui ne provoque pas d'erreur est définitive. En revanche, le SGBD ne réalisera concrètement des modifications que si ces dernières ne violent pas les *contraintes d'intégrité* (voir Chapitre 1).