Chapitre 10 - Bases de données - Langage SQL

I - Modèle relationnel

Un peu d'histoire !

- Ce sont l'apparition des disques durs qui ont permis d'envisager le stockage des données dans les années 50.
- C'est lors du programme Apollo en 1960 que l'idée de la base de données a été lancée. Le but était de collecter des données afin de permettre d'aller sur la lune avant la fin de la décennie.
- Quasiment toutes les modèles relationnels actuelles sont basées sur les travaux d'Edgar F. Codd (1970).
- A partir des années 2000, les modèles relationnels ont pris leur envol avec l'émergence d'internet et l'apparition du big data, gigantesque collecte de données qu'il faut stocker, modifier, traiter.

Définition 1

Le **modèle relationnel** est une manière de modéliser les relations existantes entre plusieurs informations et de les ordonner entre elles.

Un modèle relationnel est donc basé sur des relations, terme que nous allons définir dans le paragraphe suivant.

1) Relation, attribut, domaine, schéma

Prenons l'exemple d'un disquaire permettant d'emprunter des albums de musique. L'ensemble de ses albums peut être représenté par l'ensemble :

```
Album = {
    ("I Still Do", "Eric Clapton", 2016, Vrai),
    ("Axis: Bold as Love", "Jimi Hendrix", 1967, Faux),
    ("Continumm", "John Mayer", 2006, Faux),
    ("Riding With The King", "Eric Clapton et B.B. King", 2000, Faux),
    ("Don't explain", "Joe Bonamassa et Beth Hart", 2011, Vrai),
    ...
}
```

Un tel ensemble s'appelle une **relation** (la relation *Album* en l'occurrence).

- Les différents éléments d'une relation s'appellent des **enregistrements**, ou tuple, ou n-uplet, ou t-uplet, ou vecteur.
- Les enregistrements d'une relation possèdent les mêmes composantes, que l'on appelle les **attributs** de la relation.

Une relation se conforme toujours à un schéma qui est une description indiquant pour chaque attribut de la relation :

- son **nom**
- son **domaine** (= le « type » de l'attribut : un entier, une chaîne de caractères, une date, etc.)

Ainsi, pour le moment, la relation *Album* possède 4 attributs :

- `titre` : le titre de l'album, une chaîne de caractères
- `artiste` : le ou les artistes de l'album, une chaîne de caractères
- `annee` : l'année de parution de l'album, un entier naturel

- `dispo` : disponibilité de l'album, un booléen

On note le schéma de la relation Album de la façon suivante :

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

ou alors

Album

titre	TEXT
artiste	TEXT
annee	INT
dispo	BOOL

Une relation peut aussi se représenter sous forme d'une **table**, et d'ailleurs on utilise souvent de manière équivalente les deux termes : relation ou table. Par exemple, la table correspondant à notre relation Album ressemble à ceci :

titre	artiste	annee	dispo
I Still Do	Eric Clapton	2016	Vrai
Axis: Bold as Love	Jimi Hendrix	1967	Faux
Continuum	John Mayer	2006	Faux
Riding With The King	Eric Clapton et B.B. King	2000	Faux
Don't explain	Joe Bonamassa et Beth Hart	2011	Vrai
...	...	...	...

2) Base de données relationnelle

Définition 2

Une **base de données relationnelle** est un ensemble de relations.

Par exemple, la base de données de notre disquaire ne contiendra pas uniquement la relation *Album*. Elle peut par exemple contenir deux autres relations : *Client* et *Emprunt* qui correspondent respectivement à l'ensemble des clients du disquaire et à l'ensemble des emprunts en cours.

Le schéma, ou la structure, d'une base de données relationnelle est l'ensemble des schémas des relations de la base. Ainsi, pour le moment, la structure (ou schéma) de la base de données du disquaire, est :

```
Album(titre TEXT, artiste TEXT, annee INT, dispo BOOL)
Client(...)
Emprunt(...)
```

où on n'a pour le moment pas complété les schémas des relations Client et Emprunt (et où le schéma de la relation Album n'est pas satisfaisant pour l'instant)

II - Concevoir la structure d'une base de données relationnelle

1) Domaine d'un attribut

Le **domaine d'un attribut** a déjà été abordé, il s'agit du "type" de l'attribut. Dans le cas de la modélisation d'une base de données, la façon de noter les domaines n'est pas primordiale (INT ou Entier ou Int ou Integer, etc. pour désigner un attribut dont les valeurs sont des entiers), mais elle le deviendra lorsque l'on créera concrètement les tables en base de données car il faudra respecter la syntaxe du **SGBD (Système de Gestion de Base de Données)** utilisé.

Contrainte de domaine

Concrètement, un SGBD doit s'assurer à chaque instant de la validité des valeurs d'un attribut, autrement dit que ces valeurs correspondent toujours au domaine de l'attribut, on appelle cela les contraintes de domaines. C'est pourquoi en pratique, la commande de création d'une table doit préciser en plus du nom des attributs, leurs domaines.

Les contraintes de domaines doivent être respectées en permanence par le SGBD : si un attribut a pour domaine INT et que l'on essaie de saisir une valeur de type FLOAT pour cet attribut, cela provoquera une erreur du SGBD. Il est donc important de bien penser le domaine de chaque attribut dès le départ.

Bien que le domaine d'un attribut paraisse assez simple à déterminer, il faut être prudent dans certains cas. Par exemple, si le domaine d'un attribut correspondant à un code postal est INT, alors si on enregistre un code postal `05000` alors celui-ci sera converti en `5000` (car `05000 = 5000` pour les entiers), ce qui ne correspond pas à un code postal valide... Il est donc nécessaire de donner le domaine TEXT à un code postal.

La conception de la structure d'une base de données n'est pas toujours aisée mais c'est un travail absolument nécessaire pour obtenir une base ne souffrant d'aucune anomalie et offrant des performances optimales.

On trouve en général ces trois étapes :

1. Déterminer les entités (objets, actions, personnes, ...) que l'on souhaite manipuler.
2. Modéliser chaque ensemble d'entités comme une relation en donnant son schéma : attributs et domaine de chaque attribut.
3. Définir les **contraintes d'intégrité** (domaine, relation, référence) de la base de données, c'est-à-dire toutes les propriétés logiques vérifiées par les données à chaque instant.

On réalise souvent ces opérations en parallèle de manière à peaufiner au fur et à mesure la structure de la base.

Nous allons expliquer ces mécanismes de conception en s'appuyant sur la base de données de notre disquaire que l'on va affiner au fur et à mesure.

2) Clé primaire

Définition 3

Une **clé primaire** est un attribut (ou une réunion d'attributs) qui permet d'identifier de manière unique un enregistrement d'une relation.

La relation Album

Quel attribut de notre relation Album peut-il jouer le rôle de clé primaire ?

Réponse : aucun !

En effet, il est (fort) possible que plusieurs albums aient le même titre (ne serait-ce qu'un album disponible en plusieurs exemplaires), que plusieurs albums concernent le même artiste et que plusieurs albums soient sortis la même année ! De manière évidente, l'attribut dispo ne permet pas d'identifier un album de manière unique non plus.

Pour y remédier, on va créer "artificiellement" un attribut `id_album` (de type INT) qui va jouer le rôle de clé primaire, chaque album possédant un attribut `id_album` différent (on utilise "id" pour "identifiant").

Pour symboliser la clé primaire dans le schéma d'une relation, il est de coutume de la souligner. Ainsi, notre relation Album a pour schéma :

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

et correspond à la table suivante :

id_album	titre	artiste	annee	dispo
2	I Still Do	Eric Clapton	2016	Vrai
5	Axis: Bold as Love	Jimi Hendrix	1967	Faux
24	Continuum	John Mayer	2006	Faux
25	Continuum	John Mayer	2006	Faux
8	Riding With The King	Eric Clapton et B.B. King	2000	Faux
11	Don't explain	Joe Bonamassa et Beth Hart	2011	Vrai
...	...	...	...	...

La relation Client

On suppose que le disquaire récolte les informations suivantes sur ses clients : un nom, un prénom et une adresse email.

- Si on choisit le nom ou le prénom comme clé primaire, il sera impossible d'enregistrer deux clients portant le même nom ou portant le même prénom, ce qui n'est pas rare.
- De même, si on choisit le couple (nom, prénom) comme clé primaire, cela empêche d'enregistrer des homonymes, ce qui peut très bien arriver également.
- Si on choisit l'adresse email comme clé primaire, cela impliquerait que deux clients ne peuvent pas avoir la même adresse email. Cela peut sembler convenir... mais on se heurterait au cas où un client ne possède pas d'adresse email (un jeune enfant par exemple, d'ailleurs ses parents ne pourraient même pas lui créer un compte à son nom avec leur propre adresse email s'ils sont eux-mêmes clients)

Comme pour la relation *Album*, il semble judicieux de créer une clé primaire artificielle, nommée `id_client` la relation *Client* qui aurait alors pour schéma :

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

et correspond à la table suivante :

id_client	nom	prenom	email
1	Dupont	Florine	dupont.florine@domaine.net
5	Pacot	Jean	jpacot@music.com
8	Rouger	Léa	NULL
3	Marchand	Grégoire	greg.marchand49@music.com

Remarque : Il est parfois possible de trouver une clé primaire sans avoir besoin d'en créer une artificiellement. Par exemple dans une relation *Livre*, le numéro ISBN pourrait jouer le rôle de clé primaire car il est unique pour chaque livre existant. Cependant en pratique, un SGBDR va souvent créer un identifiant unique pour chaque enregistrement d'une entité dans la base de données. Pour cela, un mécanisme d'auto-incrément est mis en oeuvre (si la clé primaire de la dernière entité créée est l'entier 57, alors la clé primaire d'une nouvelle entité créée sera 58).

Contrainte de relation

Une des contraintes d'intégrité d'une base de données s'appelle la contrainte de relation. Celle-ci impose que chaque enregistrement d'une relation soit unique. C'est donc la présence d'une clé primaire dans chaque relation qui permet de réaliser cette contrainte.

3) Clé étrangère

La relation Emprunt

Pour un emprunt, on aimerait connaître l'album emprunté, le client qui a emprunté l'album et la date d'emprunt.

On voit donc que les enregistrements de la relation *Emprunt* font référence à des enregistrements des relations *Album* et *Client*. On peut imaginer le schéma suivant pour la relation *Emprunt*, qui contient toutes les informations nécessaires :

```
Emprunt(id_client INT, nom TEXT, prenom TEXT, email TEXT, id_album
INT, titre TEXT, artiste TEXT, annee INT, dispo BOOL, date DATE)
```

Cela donnerait une table *Emprunt* du genre :

id_client	nom	prenom	email	id_album	titre	artiste	annee	dispo	date
1	Dupont	Florine	dupontf@domaine.net	5	Axis: Bold as Love	Jimi Hendrix	1967	Faux	10/09/2021
3	Mira	Grégoire	gmira49@music.com	8	Riding With The King	Eric Clapton et B.B. King	2000	Faux	18/08/2021
3	Mira	Grégoire	gmira49@music.com	24	Continumm	John Mayer	2006	Faux	18/08/2021
5	Pacot	Jean	jpacot@music.com	25	Continumm	John Mayer	2006	Faux	12/09/2021

Attention : la relation Emprunt n'est pour le moment pas satisfaisante, nous allons l'améliorer un peu plus bas.

Définition 4

Une **clé étrangère** d'une relation est un attribut qui est clé primaire d'une autre relation de la base de données.

Ainsi, la relation *Emprunt* donnée plus haut possède deux clés étrangères : `id_client` et `id_album` (qui sont des clés primaires respectives des relations Client et Album.).

Mais la relation *Emprunt* ne possède toujours pas de clé primaire.
Que peut-on choisir comme clé primaire de la relation Emprunt ?
Une même personne pouvant emprunter plusieurs albums en même temps, il n'est pas possible d'utiliser les attributs correspondant à la relation *Client*.
En revanche, comme un même album ne peut pas être emprunté par deux clients en même temps, on peut choisir `id_album` comme clé primaire de la relation *Emprunt*.

Redondance des données

Dans une base de données relationnelle, il faut éviter la redondance des données c'est-à-dire qu'une relation ne doit pas contenir des informations déjà disponibles dans d'autres relations (et de manière générale, éviter que des mêmes informations se retrouvent dans plusieurs enregistrements d'une même relation).

Par exemple, la relation *Emprunt* telle que nous l'avons définie plus haut contient beaucoup d'informations redondantes. En effet :

- pour faire le lien avec l'emprunteur, il est inutile de garder simultanément les attributs `id_client`, `nom`, `prenom` et `email` : il suffit de conserver l'attribut `id_client` qui fait entièrement référence à un **unique** client de la relation *Client* dans laquelle on retrouve le nom, le prénom et l'adresse email de celui-ci. Ainsi, on évite la redondance des attributs `nom`, `prenom` et `email` : on ne les garde que dans la relation *Client* ;
- de même, pour faire le lien avec l'album emprunté, il suffit de conserver l'attribut `id_album` qui caractérise entièrement un album et permet de retrouver le titre, l'artiste, l'année et la disponibilité dans la relation *Album*.

Moralité : ce sont les clés étrangères (ici `id_client` et `id_album`) qui permettent à elles seules de faire le lien avec des entités d'autres relations, et on évite ainsi les redondances.

Sachant que l'on peut noter les clés étrangères d'une relation en utilisant un "#", on peut désormais écrire une version satisfaisante de la relation *Emprunt* :

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

Remarque : La clé `id_album` est donc à la fois clé primaire et clé étrangère de la relation *Emprunt*. La clé `id_client` est une clé étrangère mais pas une clé primaire de la relation *Emprunt* : cela implique qu'un même client peut se trouver plusieurs fois dans la relation *Emprunt*, il peut donc emprunter plusieurs albums à la fois (et heureusement !).

On obtient ainsi la table correspondant à la relation *Emprunt* :

id_client	id_album	date
1	5	10/09/2021
3	8	18/08/2021
3	24	18/08/2021
5	25	12/09/2021

Pourquoi éviter la redondance des données ?

La redondance des données est considérée comme une anomalie d'une base de données, synonyme d'une mauvaise conception de la base. En effet, celle-ci est proscrite pour plusieurs raisons :

- faire apparaître des informations non nécessaires à plusieurs endroits (dans plusieurs relations) d'une base de données entraîne un coût en mémoire plus important et des performances moindres lorsqu'il s'agira d'effectuer des requêtes sur la base ;
- si des corrections doivent être faites, elles doivent être faites à un seul endroit : imaginez qu'un emprunteur change de nom (ou d'adresse email), si on a pris soin de ne pas le faire apparaître dans la table *Emprunt*, il suffit alors de le modifier (une seule fois) dans la table Client et on n'a pas à faire la modification sur chaque ligne de la table *Emprunt*. Cela permet d'amener de la cohérence à notre base de données.

Contrainte de référence

La **cohérence et les relations entre les différentes tables sont assurées par les clés étrangères**. Elles permettent de respecter ce qu'on appelle les contraintes de référence :

Une clé étrangère d'une relation doit nécessairement être la clé primaire d'une autre relation
$\Rightarrow$ cela permet de s'assurer de ne pas ajouter des valeurs fictives ne correspondant pas à des entités connues de la base de données ;

Un enregistrement ne peut être effacé que si sa clé primaire n'est pas associée à des enregistrements liés dans d'autres relations
$\Rightarrow$ on ne pourrait pas supprimer le client "Dupont Florine" de la relation *Client* car il apparaît dans les enregistrements de la relation *Emprunt*. En effet, sinon la valeur '1' de la clé étrangère `id_client` de la table *Emprunt* ne serait plus une clé primaire d'une autre table.

Une clé primaire ne peut pas être modifiée si l'enregistrement en question est associé à des enregistrements liés dans d'autres tables
$\Rightarrow$ on ne pourrait pas modifier la clé primaire `id_client` du client "Dupont Florine" dans la relation *Client* car elle apparaît dans les enregistrements de la relation *Emprunt*. En effet, sinon la valeur '1' de la clé étrangère `id_client` de la table *Emprunt* ne serait plus une clé primaire d'une autre table.

Concrètement, une tentative de violation de contrainte de référence provoquerait une erreur du SGBD.

Liens entre albums et artistes

On termine la modélisation de la structure de la base de données du disquaire en définissant un peu mieux le lien entre un album et l'artiste (ou les artistes) de l'album.

Pour le moment, il a été choisi d'utiliser une chaîne de caractères pour l'artiste d'un album directement dans la relation *Album*. Cette façon de faire peut conduire à quelques problèmes :

- rien n'empêche d'associer plusieurs fois le même artiste à un album puisqu'on écrit ce que l'on veut dans une chaîne de caractères : on pourrait écrire "Éric Clapton et Éric Clapton" sans que le SGBD ne provoque une erreur, alors même qu'il y aurait un problème de cohérence.
- on a de plus un problème de redondance dans le cas (bien que rare) où un artiste changerait de nom car il faudrait le modifier pour chaque album de la table *Album*.

Pour pallier à ces problèmes, on peut :

- scinder la relation *Album* en trois relations : *Album*, *Artiste* et *Artiste_de* ;
- et utiliser les clés étrangères pour faire les associations nécessaires entre les artistes et les albums.

Concrètement :

- On retire l'attribut `artiste` de la relation *Album* :

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

- On crée un nouvelle relation, *Artiste*, correspondant uniquement aux différents artistes et ayant le schéma suivant :

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

- On associe, grâce aux clés étrangères, les artistes aux albums en créant une nouvelle relation *Artiste_de* :

```
Artiste_de(#id_artiste INT, #id_album INT)
```

  Dans cette dernière relation, les clés étrangères `id_artiste` et `id_album` permettent d'associer les relations *Artiste* et *Album*. Le couple `(id_artiste, id_album)` forme la **clé primaire** de la relation *Artiste_de*.

Ainsi, un même artiste et un même album ne peuvent se trouver plusieurs fois dans la relation, ce qui empêche d'associer deux fois le même artiste à un même album. Mais un même artiste peut être associé à plusieurs albums différents car `id_artiste` n'est pas clé primaire de la relation.

Ces transformations donnent des tables ressemblant à :

Relation *Album*

id_album	titre	annee	dispo
2	I Still Do	2016	Vrai
5	Axis: Bold as Love	1967	Faux
24	Continuum	2006	Faux
25	Continuum	2006	Faux
8	Riding With The King	2000	Faux
11	Don't explain	2011	Vrai

Relation *Artiste*

id_artiste	nom	prenom
1	Clapton	Éric
3	Hendrix	Jimi
4	Mayer	John
8	B.B. King	NULL
6	Hart	Beth
15	Bonamassa	Joe

Relation *Artiste_de*

id_artiste	id_album
1	2
1	8
8	2
4	24
4	25
3	5
6	11
15	11

Désormais, on n'enregistre qu'à un seul endroit les chaînes de caractères correspondant au nom et au prénom de chaque artiste, et lorsque l'on veut faire référence à cet artiste dans une autre relation, c'est l'entier correspondant à `id_artiste` qui est enregistré en mémoire. Ainsi, si un artiste change de nom, il suffit alors de modifier son nom une seule fois dans la relation Artiste.

Par ailleurs, on obtient également un gain :

- en **mémoire**, car un entier est (presque tout le temps) stocké sur moins de bits qu'une chaîne de caractères ;
- en **performance**, car lorsque l'on effectuera des recherches dans la base (comme la recherche de tous les albums d'un artiste donné par exemple), les comparaisons entre deux entiers sont plus rapides qu'entre deux chaînes de caractères.

4) Schéma (final) de notre base de données

Avec toutes les améliorations apportées, le schéma (ou structure) de la base de données du disquaire est le suivant :

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

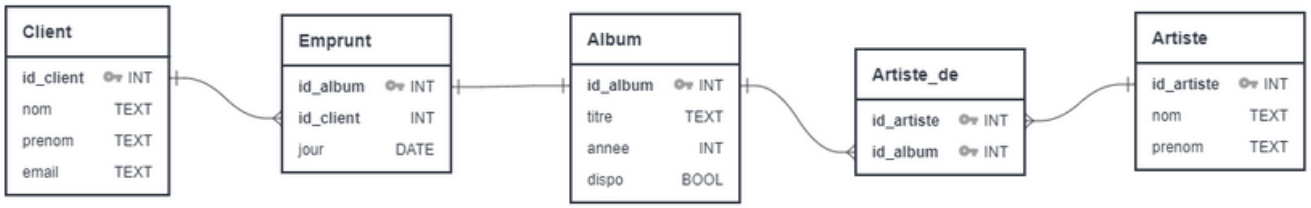
```
Artiste(id_artiste INT, nom TEXT, prenom TEXT)
```

```
Artiste_de(#id_artiste INT, #id_album INT)
```

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

```
Emprunt(#d_client INT, #id_album INT, date DATE)
```

On peut aussi représenter graphiquement ce schéma par le diagramme suivant :



Réalisé avec l'application quickdatabasediagrams.com

Remarques : Dans ce diagramme :

- les clés primaires sont matérialisées ici par un symbole de clé (et en gras). Mais on trouve aussi parfois l'acronyme CP, pour clé primaire, ou plus souvent sa version anglaise PK, pour primary key.
- les clés étrangères sont matérialisées par un trait marquant les associations entre les différentes relations (et en gras). Mais on trouve aussi souvent l'acronyme FK (foreign key, traduction de clé étrangère).

III - Système de gestion de bases de données (SGBD)

Dans une base de données, l'information est stockée dans des fichiers, mais ceux-ci ne sont en général pas lisibles par un humain : ils nécessitent l'utilisation d'un logiciel appelé système de gestion de bases de données (abrégé SGBD) pour les exploiter (écrire, lire ou encore modifier les données).

Ce sont de très gros logiciels, fonctionnant pour la grande majorité en mode client/serveur, assez complexes à mettre en œuvre et à utiliser.

Parmi les SGBD les plus connus, nous avons

- dans le domaine du libre :
  - → MariaDB / MySQL
  - → PostgreSQL
  - → SQLite (qui ne fonctionne pas sur le modèle client/serveur, toute la BDD est stockée dans un fichier qui peut être stocké dans l'arborescence)
- dans le monde propriétaire :
  - → Oracle Database
  - → IBM DB2
  - → Microsoft SQL Server.

Ils sont conçus pour gérer plusieurs millions, voire milliards d'enregistrements de manière fiable et sécurisée. Leur architecture côté serveur est prévue pour être répartie sur plusieurs machines et ainsi permettre une tenue en charge lorsqu'un grand nombre de requêtes parviennent.

Services rendus

Pour exécuter toutes les tâches évoquées précédemment, les services rendus par un SGBD sont les suivants.

Efficacité de traitement des requêtes

Les bases de données pouvant être très volumineuses (jusqu'au pétaoctet), il faut que les requêtes soient efficaces (en temps) et les SGBD optimisent la plupart des requêtes SQL que l'on écrit pour que les recherches soient très rapides (même parmi des millions voire milliards d'enregistrement)

Sécurisation des accès

Les SGBD permettent de gérer les autorisations d'accès à une base de données. Il est en effet souvent nécessaire de contrôler les accès, par exemple en permettant à l'utilisateur A de lire et d'écrire dans la base de données alors que l'utilisateur B aura uniquement la possibilité de lire les informations contenues dans cette même base de données.

Persistance des données

Les fichiers des bases de données sont stockés sur des disques durs dans des ordinateurs, ces ordinateurs peuvent subir des pannes. Il est souvent nécessaire que l'accès aux informations contenues dans une base de données soit maintenu, même en cas de panne matérielle. Les bases de données sont donc dupliquées sur plusieurs ordinateurs afin qu'en cas de panne d'un ordinateur A, un ordinateur B contenant une copie de la base de données présente dans A, puisse prendre le relais.

Les SGBD assurent cette persistance des données (duplication pour accès à tout moment et synchronisation des différentes copies de la base de données).

Gestion des accès concurrents

Plusieurs personnes ou applications peuvent avoir besoin d'accéder aux informations contenues dans une base données en même temps. Cela peut parfois poser problème, notamment si les 2 personnes désirent modifier la même donnée au même moment (on parle d'accès concurrent), ce qui arrive très souvent : réservation ou achat en ligne d'un même objet, paiements en ligne à partir ou en direction d'un même compte bancaire, etc.

Les SGBD assurent qu'une telle situation ne peut se produire : en particulier, ils utilisent un mécanisme de transaction (= ensemble d'opérations qui ne peut pas être interrompu par une autre transaction ; cela peut se faire grâce à l'utilisation de verrous, voir cours sur la gestion des processus).

Évidemment, comme nous l'avons déjà vu, s'il s'agit d'un système de gestion de bases de données relationnelles (SGBDR), alors le SGBD s'assure aussi en permanence que les contraintes d'intégrité soient respectées (contraintes de domaine, de relation et de référence ; voir chapitre sur le modèle relationnel), c'est-à-dire que les données soient constamment cohérentes avec le modèle (mathématique, logique) relationnel de la BDD.

IV - Langage SQL

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.

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.

Le modèle relationnel du disquaire

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

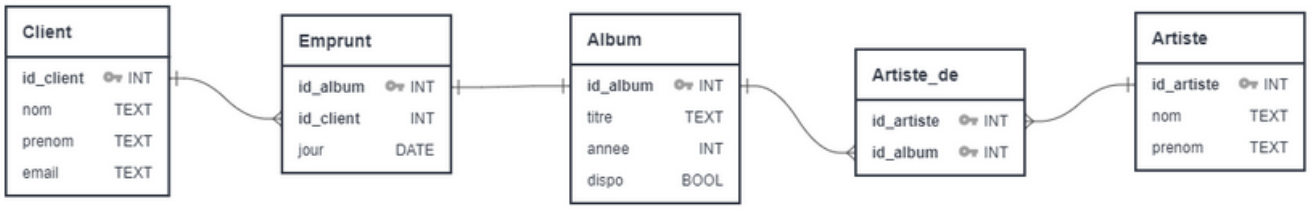
```
Artiste(id_artiste INT, nom TEXT, prenom TEXT)
```

```
Artiste_de(#id_artiste INT, #id_album INT)
```

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

```
Emprunt(#d_client INT, #id_album INT, date DATE)
```

Voici le diagramme représentant ce schéma :



Réalisé avec l'application quickdatabasediagrams.com

1) 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.

Sélectionner des colonnes avec `SELECT`

Première requête SQL

La requête suivante permet d'afficher tout le contenu de la table *Album*.

```
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

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

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

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 donne le résultat ci-contre :

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

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\%";
```

qui donne le résultat ci-contre :

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

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

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;
```

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;
```

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;
```

MAX(nom)	prenom
Wonder	Stevie

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`

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;
```

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	email
1	5	2021-09-10	1	Dupont	Florine	dupont.florine@domaine.net
3	8	2021-08-18	3	Marchand	Grégoire	greg.marchand49@music.com
3	24	2021-08-18	3	Marchand	Grégoire	greg.marchand49@music.com
5	25	2021-09-12	5	Pacot	Jean	jpacot@music.com
5	6	2021-10-10	5	Pacot	Jean	jpacot@music.com
9	20	2021-09-28	9	Dubois	Philippe	pdubois5@chezmoi.net
11	14	2021-10-08	11	Fournier	Marie	mfournier@abc.de
7	15	2021-10-08	7	Moreau	Alain	amoreau1@abc.de
7	19	2021-10-08	7	Moreau	Alain	amoreau1@abc.de
7	16	2021-10-15	7	Moreau	Alain	amoreau1@abc.de
16	29	2021-10-01	16	Bernardin	Stéphanie	sbernard1@chezmoi.net

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 lesquelles l'attribut `id_client` est identique donc celles concernant un même client.
- 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 et le jour d'emprunt, on peut faire la requête

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

Le résultat de cette jointure est :

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.email
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.

Le résultat de cette jointure est :

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 suivante 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.

```
SELECT Album.titre, Emprunt.jour, Client.nom, Client.prenom, Client.email
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;
```

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*.

2) Modifier des données

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`

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', 'Eric'),
                           (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 `Artiste`) 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', 4);
```

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. 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', 'Eric'),
                                         ('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.

Modifier une valeur avec `UPDATE ... SET`

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

Par exemple, le client ci-dessous a changé d'adresse email.

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

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 :

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 la requête suivante :

```
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` :

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

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. 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 !).

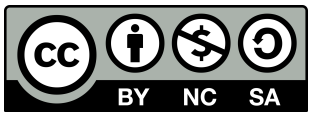
Par exemple, si on essaie de supprimer de la relation `Client` le client "Dupont Florine", dont l'attribut `id_client` est 1, le SGBD empêchera la suppression car ce client apparaît dans la relation `Emprunt` en tant que clé étrangère (si la suppression était effectuée, cette clé étrangère ne ferait 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
```



Sources :

- Lycée Mounier - Angers