Chapitre :
Bases de données

I. Introduction

1. Historique

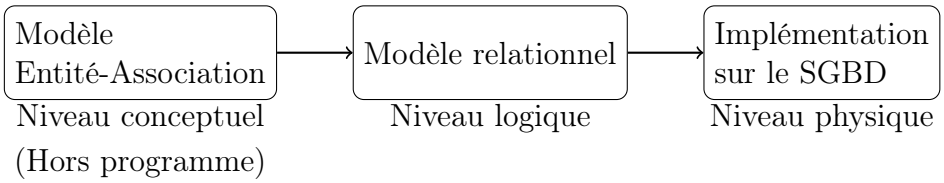
Les disques durs ont été inventés en 1956 et a permis d'utiliser les ordinateurs pour collecter, classer et stocker de grandes quantités d'informations.
Le terme database (base de données) est apparu en 1964.
En 1965, Charles Bachman conçoit l'architecture Ansi/Sparc encore utilisée de nos jours. Charles Bachman a reçu le prix Turing en 1973 pour ses « contributions exceptionnelles à la technologie des bases de données ».
En 1970, Edgar F. Codd soutient sa thèse qui est à l'origine des bases de données relationnelles. Edgar F. Codd a reçu le prix Turing en 1981.
Le modèle entité-association a été inventé par Peter Chen en 1975 ; il est destiné à clarifier l'organisation des données dans les bases de données relationnelles.

2. Quelques exemples

- La collecte de renseignements administratifs : dossiers médicaux, fiscaux.
- Billetterie informatisée : billets de voyage ou de concert.
- Catalogue en ligne comme amazon.com, une des plus grandes bases de données au monde avec plus de 250 millions d'ouvrages catalogués.

3. Trois niveaux de conception

1. Niveau conceptuel : représenter la base de données indépendamment de toute considération informatique.
2. Niveau logique : adaptation du schéma conceptuel en tableaux.
3. Niveau physique : implémentation informatique sur un SGBD.



Nous allons introduire les notions de base de données par un exemple : celui de la modélisation l'emprunt de livres dans un CDI.
Après avoir décrit les données à gérer, nous élaborerons le modèle entité-association. Ce modèle est hors programme en terminale NSI mais courant en bases de données et il introduit tout de même des notions du programme. Nous montrerons ensuite comment obtenir le modèle relationnel, qui lui est au programme.

II. Exemple introductif : gestion de CDI

1. Description des données à gérer

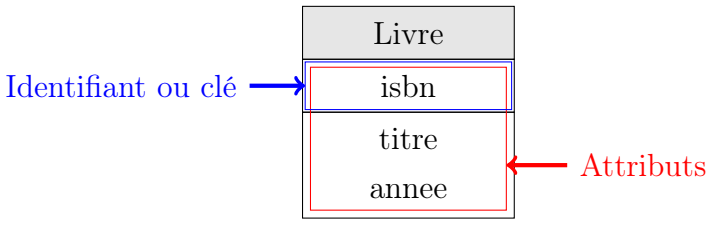
Dans un CDI, on trouve des livres, identifiés de manière unique par leur ISBN (International Standard Book Number), pour lesquels on dispose en plus de l'information de leur titre et de l'année de publication.
Ces livres ont été écrits par un ou plusieurs auteurs, dont on connaît le nom mais qui ont également un numéro d'identification unique, qui peuvent avoir écrit un ou plusieurs livres.
Ces livres ont été publiés par des éditeurs, dont on connaît là encore le nom et qui sont identifiés de manière unique par le SIRET (Système d'identification du répertoire des établissements). Chaque livre n'est publié que par un éditeur, mais un éditeur peut avoir publié un ou plusieurs livres.
Le CDI possède une liste des élèves, identifiés de manière unique par leur numéro d'étudiant, avec les informations sur le nom, le prénom et la classe.
Chaque livre peut être emprunté par un seul élève, un même élève pouvant emprunter plusieurs livres. Pour chaque emprunt, une date de retour est mise en mémoire.
On souhaite stocker toutes ces informations dans une base de données.
Dans un premier temps nous établirons le modèle entité-association (ici sans les éditeurs). Ensuite, nous transformeront ce modèle en modèle relationnel. Plus tard, à l'aide du langage SQL, nous créerons puis remplirons les tables.

2. Modèle entité-association

L'emprunt d'un livre au CDI

Chaque élève d'un lycée peut emprunter des livres au CDI, les données concernant ces livres et les emprunts en cours sont stockés dans une base de données.
Les **entités** qui constituent cette base sont les *élèves* et les *livres*. Mais on peut en considérer d'autres comme les *auteurs* ou les *éditeurs*.

L'entité Livre



Les attributs sont un ensemble de propriétés qui décrivent l'entité. L'identifiant (ou la clé) est un (ou plusieurs) attribut(s) qui permet d'identifier de manière unique l'entité.

Occurrences de l'entité Livre

L'ensemble des livres de la bibliothèque peut être représenté par des *n-upets* (ou tuples).

isbn	titre	annee
978-2-07-046614-6	La fête de l'insignifiance	2013
978-2-264-02881-5	Ubik	1969
978-2-13-054387-9	Histoire des sciences	2018

Les entités Eleve et Auteur

Eleve
num_etudiant
nom
prenom
classe

Auteur
identifiant
nom
prenom

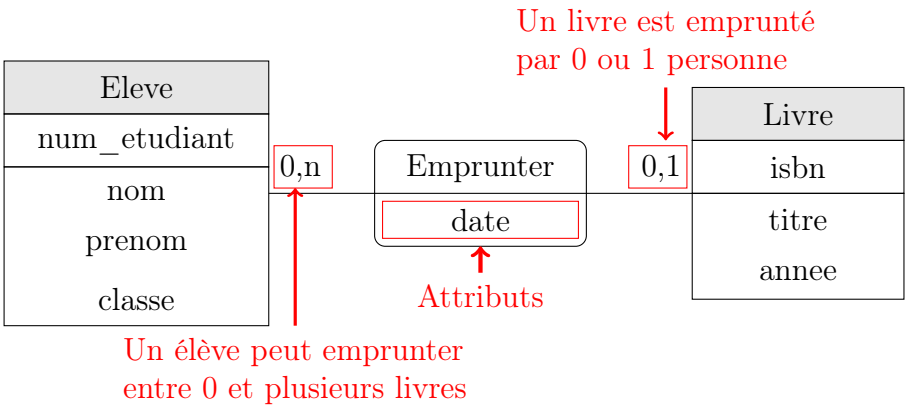
Occurrences des entités Eleve et Auteur

num_etudiant	nom	prenom
1592051067r	Bolognaise	Thomas
2561485589o	Paquito	Enzo
4975322589k	Royal	Henri

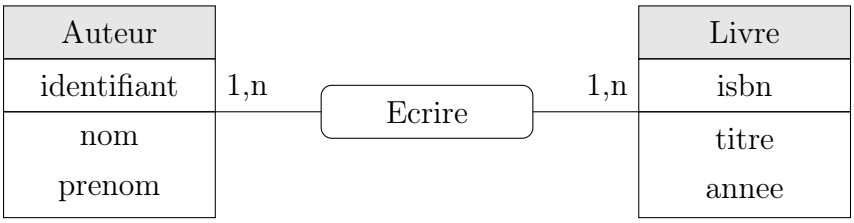
identifiant	nom	prenom
1	Kundera	Milan
2	K. Dick	Philip
3	Gingras	Yves

L'association Emprunter

Une **association** est un lien entre des entités.
Les **cardinalités** sont des couples de valeurs (min,max) qui précisent le nombre de fois qu'un élément d'une entité donnée peut être associé à des éléments de l'entité associée.

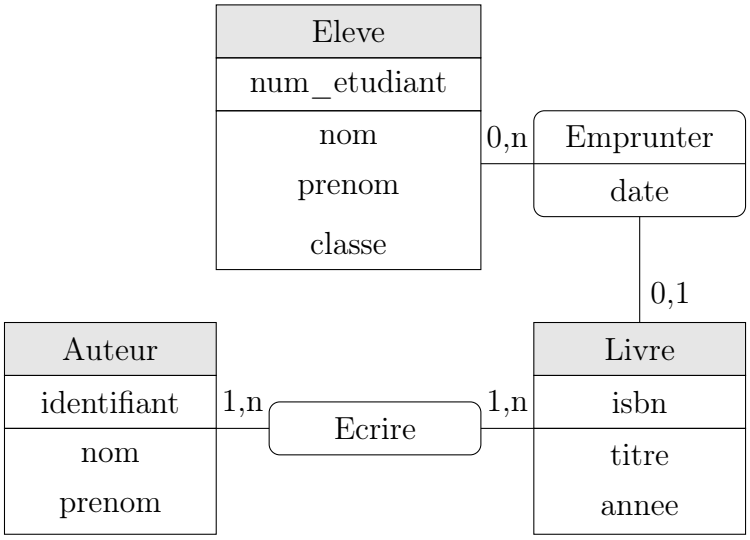


L'association Ecrire



- Un Auteur écrit entre 1 et plusieurs livres.
- Un livre est écrit par 1 ou plusieurs Auteurs.

Modèle entité-association complet

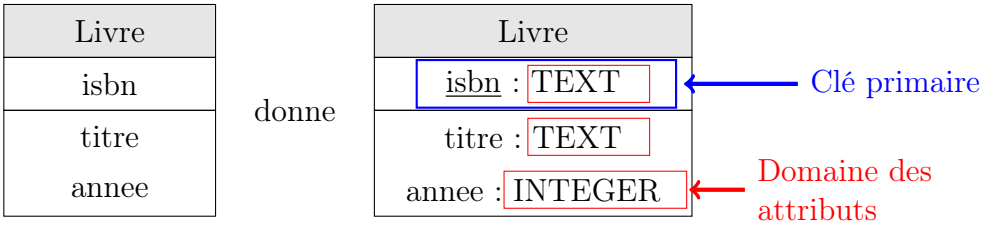


À titre d'exercice : Compléter ce schéma conceptuel avec une entité *Editeur* qui comporte un attribut *siret*, qui l'identifie de manière unique, et un *nom* ainsi qu'une relation *Editer* qui la relie à l'entité *Livre*.

3. Modèle relationnel

Transformation d'une entité en relation

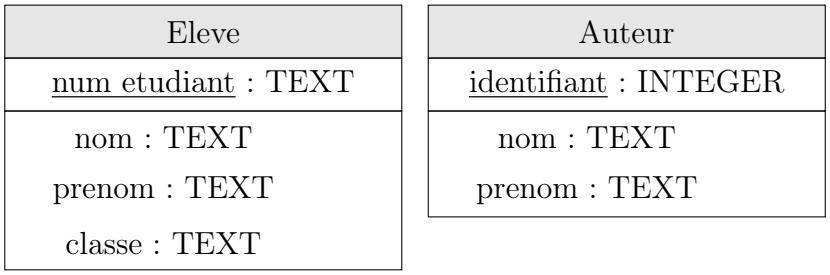
Dans le schéma relationnel, les entités et les associations sont transformées en tableaux appelés **relations**.



- L'ensemble des valeurs possibles d'un attribut définit un **domaine**.
- L'identifiant de l'entité est la **clé primaire** de la relation.

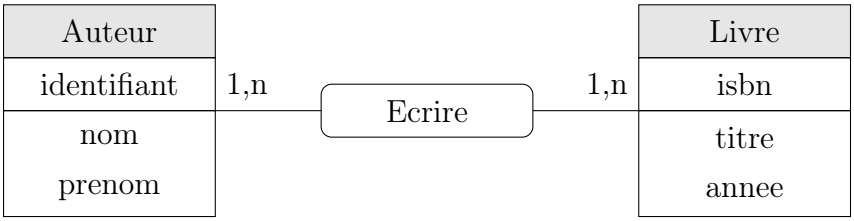
Notation textuelle (appelé schéma de la relation) :
Livre(isbn TEXT, titre TEXT, annee INTEGER)

Relation Eleve et Auteur

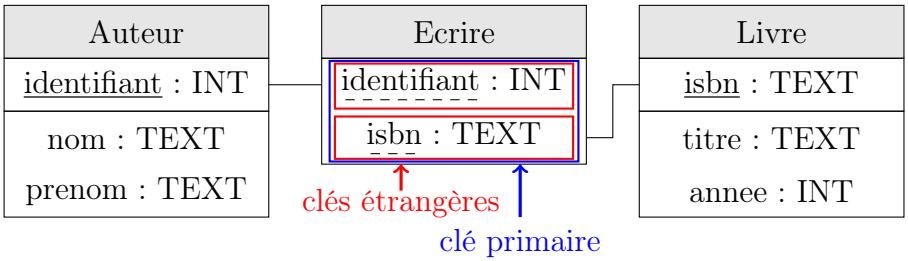


Eleve(num etudiant TEXT, nom TEXT, prenom TEXT, classe TEXT)
Auteur(identifiant INTEGER, nom TEXT, prenom TEXT)

Transformation d'une association en relation

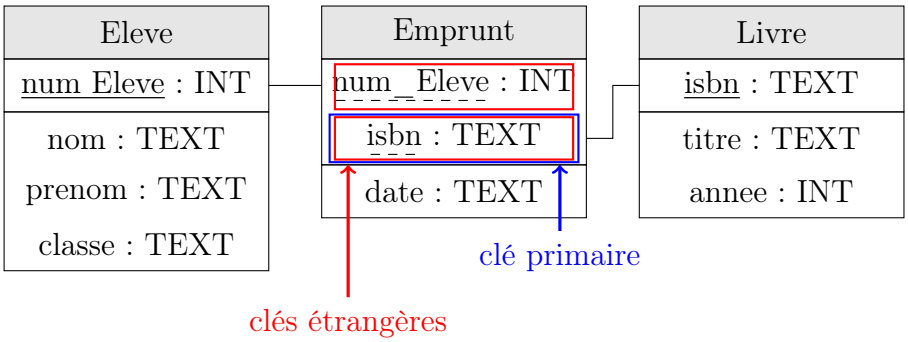


Devient :



Ecrire(identifiant INT,isbn TEXT)

La relation Emprunt



Remarque Si on donne pour clé primaire le couple (num_Eleve,isbn), alors un même livre peut être emprunté par deux élèves différents.
Pour résoudre cela, on choisit de prendre pour clé primaire le seul isbn. Un même livre ne peut plus apparaître dans deux emprunts différents.
Il faut cependant effacer de la base un emprunt une fois que celui ci est fini.

Emprunt(isbn TEXT,num Eleve TEXT,date TEXT)
À titre d'exercice : Écrire les relations *Editeur* et *Editer*.

4. Contraintes d'intégrité

Une contrainte d'intégrité permet de garantir la cohérence des données lors des mises à jour de la base.

1. **Contrainte d'entité** (de relation) : chaque relation dans le schéma relationnel est identifiée par une clé primaire qui doit être unique et non nulle.
2. **Contrainte de référence** : Une valeur de clé étrangère existe bien en tant que clé primaire dans une autre table.
3. **Contrainte de domaine** : chaque attribut doit prendre une valeur dans le domaine de valeurs.
4. **Contrainte utilisateur** : des contraintes supplémentaires, non modélisables dans les relations, peuvent être imposées selon la nature des données stockées.

III. langage SQL

Le langage SQL (*Structured Query Language*) permet de manipuler les bases de données sous forme d'instructions qui sont de deux natures :

- Les *mises à jour* permettent la création de relations, l'ajout de données dans ces dernières, leur modification et leur suppression.
- Les *requêtes* permettent de récupérer les données répondant à des critères particuliers.

1. Mises à jour

a. Création de tables

La commande `CREATE TABLE` permet de créer une table (relation) en définissant le nom et le type des attributs.
Pour créer les tables correspondants aux relations de la modélisation du CDI, on peut saisir les ordres suivants :

```
CREATE TABLE Eleve (num_etu INT PRIMARY KEY,
                    nom VARCHAR(90),
                    prenom VARCHAR(90),
                    classe VARCHAR(90));
CREATE TABLE Livre (isbn CHAR(14) PRIMARY KEY,
                    siret INT REFERENCES Editeur(siret),
                    titre VARCHAR(300),
                    annee INT);
CREATE TABLE Editeur (siret INT PRIMARY KEY,
                      nom VARCHAR(90));
CREATE TABLE Auteur (a_id INT PRIMARY KEY,
                     nom VARCHAR(200));
CREATE TABLE Ecrire (a_id INT REFERENCES Auteur(a_id),
                     isbn CHAR(14) REFERENCES Livre(isbn),
                     PRIMARY KEY (a_id,isbn));
CREATE TABLE Emprunt (isbn CHAR(14) REFERENCES Livre(isbn),
                      num_etu INT REFERENCES Eleve(num_etu),
                      date_ret DATE,
                      PRIMARY KEY (isbn));
```

Les différents types de données

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 après la virgule
REAL	approché	flottant 32 bits
DOUBLE PRECISION	approché	flottant 64 bits

Types textes

nom du type	description
CHAR(n)	chaîne d'exactement n caractères, les caractères manquant sont complétés par des espaces
VARCHAR(n)	chaîne d'au plus n caractères
TEXT	chaîne de taille quelconque

Types dates

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'

Une fonctionnalité intéressante des types dates est la possibilité d'ajouter ou soustraire une durée. Par exemple si `d` est de type `DATE` alors `d+10` est de type `DATE` et représente la date 10 jours après `d`.

Les contraintes d'intégrité

Clé primaire : Les mots clés `PRIMARY KEY` permettent d'indiquer qu'un attribut est clé primaire :

```
CREATE TABLE Editeur (siret INT PRIMARY KEY,
                      nom VARCHAR(90));
```

Si plusieurs attributs sont forment la clé primaire, on peut spécifier la contrainte après les attributs :

```
CREATE TABLE Ecrire (a_id INT REFERENCES Auteur(a_id),
                     isbn CHAR(14) REFERENCES Livre(isbn),
                     PRIMARY KEY (a_id,isbn));
```

Clé étrangère : Un attribut peut être qualifié de clé étrangère avec le mot clé `REFERENCES` suivi du nom de la table où se trouve la clé primaire et de son nom.

```
CREATE TABLE Ecrire (a_id INT REFERENCES Auteur(a_id),
                     isbn CHAR(14) REFERENCES Livre(isbn),
                     PRIMARY KEY (a_id,isbn));
```

Unicité, 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 à l'aide du mot clé `UNIQUE`.

```
CREATE TABLE Eleve (num_etu INT PRIMARY KEY,
                    nom VARCHAR(90),
                    prenom VARCHAR(90),
                    classe VARCHAR(90),
                    email VARCHAR(60) UNIQUE);
```

Une autre pratique consiste à déclarer qu'un attribut ne peut être vide (`NULL`) à l'aide des mots clés `NOT NULL`.

```
CREATE TABLE Auteur (a_id INT PRIMARY KEY,
                     prenom VARCHAR(200) NOT NULL,
                     nom VARCHAR(200) NOT NULL);
```

Contraintes utilisateurs : Il est possible de spécifier des contraintes arbitraires sur les attributs d'une même ligne au moyen du mot clé `CHECK`.

```
CREATE TABLE Produit (pid INT PRIMARY KEY,
                      nom VARCHAR(100) NOT NULL,
                      quantite INT NOT NULL,
                      prix DECIMAL(10,2) NOT NULL,
                      CHECK (quantite >= 0 AND prix >= 0));
```

b. Modification de tables

Suppression de tables

Pour recréer une table, par exemple avec un schéma différent, il faut d'abord supprimer celle portant le même nom à l'aide de l'instruction `DROP TABLE`.

```
DROP TABLE Ecrire;
```

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 une contrainte de référence :

```
DROP TABLE Auteur;

constraint failed
```

Il convient donc de supprimer les tables dans le bon ordre : d'abord les tables contenant les clé étrangères puis celles contenant les clés primaires référencées.

Insertion dans une table

Insérer une valeur dans une table s'effectue grâce à l'instruction `INSERT INTO`.

```
INSERT INTO Auteur VALUES (1,'Milan','Kundera'),
                          (2,'Philip','K.Dick'),
                          (3,'Yves','Gingras');
```

Si on souhaite passer les valeurs dans un autre ordre, il faut le spécifier.

```
INSERT INTO auteur VALUES(nom,prenom,a_id)
                   VALUES ('Rousseau','Jean-Jaques',4);
```

Les contraintes d'intégrités sont vérifiées au moment de l'insertion.

```
INSERT INTO auteur VALUES (1,'Alexandre Dumas');

UNIQUE constraint failed: auteur.a_id
```

c. Modification des données

Suppression de lignes

La commande `DELETE` en SQL permet de supprimer des lignes dans une table. En utilisant cette commande associé à `WHERE` il est possible de sélectionner les lignes concernées qui seront supprimées. La syntaxe pour supprimer des lignes est la suivante :

```
DELETE FROM nom_de_table WHERE condition
```

Exemple Si un élève rend le livre *1984*, dont l'ISBN est 978-0547249643, il faut supprimer la ligne correspondant dans la table `emprunt`.

```
DELETE FROM emprunt WHERE isbn = '978-0547249643';
```

Exemple Si l'élève *Head Bryar*, dont le numéro d'étudiant est 160307224881 rend tous ces livres, il faut supprimer les lignes correspondantes.

```
DELETE FROM emprunt WHERE num_etu = 160307224881;
```

Mise à jour

La commande `UPDATE` permet d'effectuer des modifications sur des lignes existantes. Très souvent cette commande est utilisée avec `WHERE` pour spécifier sur quelles lignes doivent porter la ou les modifications.
La syntaxe basique d'une requête utilisant `UPDATE` est la suivante :

```
UPDATE table
SET nom_colonne_1 = 'nouvelle valeur'
WHERE condition
```

Exemple Si l'on souhaite par exemple prolonger de 30 jours l'emprunt du livre *1984*, dont l'ISBN est 978-0547249643, il suffit d'effectuer la requête SQL ci-dessous.

```
UPDATE emprunt SET date_ret = date_ret + 30
WHERE isbn = '978-0547249643';
```

2. Requêtes

a. Sélection des données

L'utilisation la plus courante de SQL consiste à lire des données issues de la base de données. Cela s'effectue grâce à la commande `SELECT`, qui retourne des enregistrements dans un tableau de résultat. Cette commande peut sélectionner une ou plusieurs colonnes d'une table.

Obtenir une colonne d'une table

L'utilisation basique de `SELECT` s'effectue de la manière suivante :

```
SELECT nom_du_champ FROM nom_du_tableau
```

Cette requête SQL va sélectionner (`SELECT`) le champ « nom_du_champ » provenant (`FROM`) du tableau appelé « nom_du_tableau ».

Exemple Si l'on veut connaître tous les noms des éditeurs présent dans notre base de données, il suffit d'effectuer la requête SQL ci-dessous.

```
SELECT nom FROM Editeur;
```

De cette manière on obtient par exemple un résultat similaire à celui-ci :

	nom
1	iMinds Pty Ltd
2	Educa Books
3	Editions Albert René
4	J'ai Lu
5	LGF/Le Livre de Poche

À noter qu'avec sqlite3, le nom des colonnes ne sont pas donnés, et les lignes ne sont pas numérotées.

Obtenir plusieurs colonnes
Il est possible de lire plusieurs colonnes à la fois. Il suffit tout simplement de séparer les noms des champs souhaités par une virgule. Pour obtenir les prénoms et les noms des élèves il faut alors faire la requête suivante :

```
SELECT nom, prenom FROM Eleve;
```

	nom	prenom
1	Rhodes	Lucian
2	Ward	Ralph
3	Mayo	Giacomo
4	Hernandez	Zeus
5	Alston	Levi

Il est possible de retourner automatiquement toutes les colonnes d'un tableau sans avoir à connaître le nom de toutes les colonnes. Au lieu de lister toutes les colonnes, il faut simplement utiliser le caractère « * » (étoile). C'est un joker qui permet de sélectionner toutes les colonnes. Il s'utilise de la manière suivante :

```
SELECT * FROM Eleve;
```

DISTINCT

L'utilisation de la commande `SELECT` en SQL permet de lire toutes les données d'une ou plusieurs colonnes. Cette commande peut potentiellement afficher des lignes en double. Pour éviter les doublons, autrement dit des redondances dans les résultats, il faut simplement ajouter `DISTINCT` après le mot `SELECT`.
On obtient autrement dit la syntaxe suivante :

```
SELECT DISTINCT ma_colonne
FROM nom_du_tableau
```

Exemple Si l'on veut connaître toutes les années de parution des livres présents au CDI sans afficher de doublon, il suffit d'effectuer la requête SQL ci-dessous :

```
SELECT DISTINCT annee FROM Livre;
```

WHERE

La commande `WHERE` dans une requête SQL permet d'extraire les lignes d'une base de données qui respectent une condition. Cela permet d'obtenir uniquement les informations désirées. La commande `WHERE` s'utilise en complément à une requête utilisant `SELECT`.
La façon la plus simple de l'utiliser est la suivante :

```
SELECT nom_colonnes FROM nom_table WHERE condition
```

Exemple Si l'on veut connaître les noms et prénoms des élèves de la classe 2-GT5, il suffit d'effectuer la requête SQL ci-dessous :

```
SELECT nom, prenom FROM Eleve WHERE classe = '2-GT5';
```

Opérateurs de comparaisons
Il existe plusieurs opérateurs de comparaisons. La liste ci-dessous présente quelques uns des opérateurs les plus couramment utilisés :

Opérateur	Description
=	égal
<> ou !=	différent de
> / >=	supérieur strictement à / supérieur ou égal à
< / <=	inférieur strictement à / inférieur ou égal à
IN	liste de plusieurs valeurs possible
BETWEEN	valeur comprise dans un intervalle donné
LIKE	valeur ayant une forme donnée

Exemple Pour connaître les livres dont le titre contient le mot « homme », il suffit d'effectuer la requête SQL ci-dessous :

```
SELECT * FROM livre WHERE titre LIKE '%homme%';
```

Le caractère « % » est un caractère joker qui remplace tous les autres caractères. Ainsi, ce modèle permet de rechercher toutes les chaînes de caractères qui contiennent le mot « homme ».

ORDER BY
La commande `ORDER BY` permet de trier les lignes dans un résultat d'une requête SQL. Il est possible de trier les données sur une ou plusieurs colonnes, par ordre ascendant ou descendant.
Une requête où l'on souhaite filtrer l'ordre des résultats utilise la commande `ORDER BY` de la sorte :

```
SELECT colonne1, colonne2
FROM table
ORDER BY colonne1
```

Exemple Pour afficher les titres des livres classés par année d'édition, il suffit d'effectuer la requête SQL ci-dessous :

```
SELECT titre, annee FROM Livre ORDER BY annee;
```

Par défaut les résultats sont classés par ordre ascendant (croissant), toutefois il est possible d'inverser l'ordre en utilisant le suffixe `DESC` après le nom de la colonne :

```
SELECT titre, annee FROM livre ORDER BY annee DESC;
```

b. Fonctions d'agrégation

Les fonctions d'agrégation dans le langage SQL permettent d'effectuer des opérations statistiques sur un ensemble d'enregistrement. Étant donné que ces fonctions s'appliquent à plusieurs lignes en même temps, elle permettent des opérations qui servent à récupérer l'enregistrement le plus petit, le plus grand ou bien encore de déterminer la valeur moyenne sur plusieurs enregistrements.
Les principales fonctions d'agrégation sont les suivantes :

- `AVG()` pour calculer la moyenne sur un ensemble d'enregistrement ;
- `COUNT()` pour compter le nombre d'enregistrement sur une table ou une colonne distincte ;
- `MAX()` pour récupérer la valeur maximum d'une colonne sur un ensemble de ligne. Cela s'applique à la fois pour des données numériques ou alphanumériques ;
- `MIN()` pour récupérer la valeur minimum de la même manière que MAX() ;
- `SUM()` pour calculer la somme sur un ensemble d'enregistrement.

L'utilisation la plus générale consiste à utiliser la syntaxe suivante :

```
SELECT fonction(colonne) FROM table
```

La fonction `COUNT()` possède une subtilité : Pour compter le nombre total de lignes d'une table, il convient d'utiliser l'étoile '*' qui signifie que l'on cherche à compter le nombre d'enregistrement sur toutes les colonnes. La syntaxe est alors la suivante :

```
SELECT COUNT(*) FROM table
```

Exemple L'année de publication la plus récente :

```
SELECT MAX(annee) FROM Livre;
```

c. Requêtes imbriquées

La réponse à une requête étant une table et les requêtes portant sur des tables, il est possible d'imbriquer les requêtes.
Dans le langage SQL, une sous-requête (aussi appelé requête imbriquée ou requête en cascade) consiste à exécuter une requête à l'intérieur d'une autre requête.
Une requête imbriquée est souvent utilisée au sein d'une clause `WHERE` pour signifier que le contenu d'une colonne doit se situer dans une table construite par une requête.

Exemple On souhaite afficher l'isbn de tous les livres écrits par Uderzo :

```
SELECT isbn FROM Ecrire
WHERE a_id in
(SELECT a_id from Auteur WHERE nom = 'Albert Uderzo');
```

d. Jointure

Les jointures en SQL permettent d'associer plusieurs tables dans une même requête. Cela permet d'exploiter la puissance des bases de données relationnelles pour obtenir des résultats qui combinent les données de plusieurs tables de manière efficace.
Pour deux tables A et B avec une clef commune, la syntaxe pour obtenir leur jointure est :

```
SELECT *
FROM A
JOIN B ON A.key = B.key
```

Exemple Pour afficher les titres des livres et les noms des éditeurs, il suffit d'effectuer la requête SQL ci-dessous.

```
SELECT Livre.titre, Editeur.nom
FROM Livre JOIN Editeur ON Livre.siret = Editeur.siret;
```

On peut également associer plus de 2 tables, si l'on veut par exemple obtenir la liste des livres empruntés avec le nom et prénom de l'élève correspondant et la date retour, il suffit d'effectuer la requête SQL ci-dessous :

```
SELECT Livre.titre, Eleve.nom, Eleve.prenom, Emprunt.date_ret
FROM Livre
JOIN Emprunt ON Livre.isbn = Emprunt.isbn
JOIN Eleve ON Eleve.num_etu = Emprunt.num_etu;
```

Utilisation de `USING` : Dans les exemples précédents, les colonnes qui permettent de faire les jointures ont les mêmes noms. Dans ce cas on peut utiliser la commande `USING` de la manière suivante :

```
SELECT livre.titre, editeur.nom
FROM livre JOIN editeur USING(siret);
```

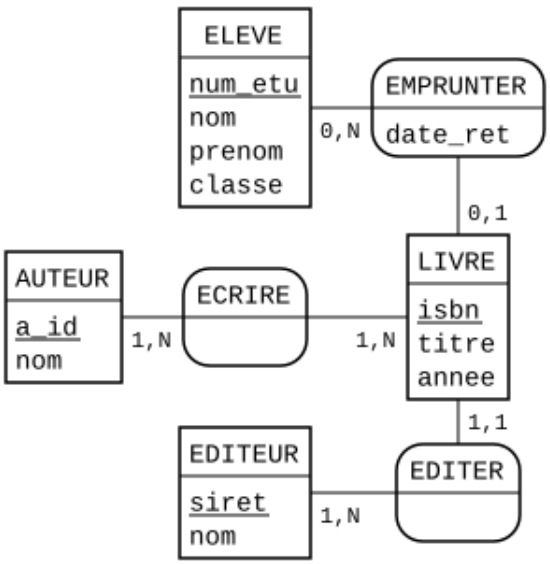
```
SELECT livre.titre, eleve.nom, eleve.prenoms, emprunt.date_ret
FROM livre
JOIN emprunt USING(isbn)
JOIN eleve USING(num_etu);
```

3. Retour sur l'exemple du CDI

Nous donnons ici à titre d'illustration la construction complète de la base de donnée introduite plus haut, y compris les requêtes permettant de la remplir.

a. Modèle entité-association

Le modèle est le suivant :



b. Modèle relationnel

Le modèle relationnel peut être le suivant :

Eleve	(num_etu, *nom*, *prenom*, *classe*)
Livre	(isbn, *titre*, *annee*, #*num_etu*, *date_ret*, #*siret*)
Editeur	(siret, *nom*)
Auteur	(a_id, *nom*)
Ecrire	(#*a_id*, #*isbn*)

Nous utiliserons cependant celui-ci, dans lequel la relation Emprunt est traduite dans une table plutôt qu'insérée dans la table Livre :

Eleve	(num_etu, *nom*, *prenom*, *classe*)
Livre	(isbn, *titre*, *annee*, #*siret*)
Editeur	(siret, *nom*)
Auteur	(a_id, *nom*)
Ecrire	(#*a_id*, #*isbn*)
Emprunt	(#*isbn*, #*num_etu* , *date_ret*)

Cependant ce modèle rend moins immédiate l'information sur l'état d'un livre (emprunté ou non). Dans aucun des cas on a accès à l'historique des emprunts.

Un **attribut souligné** indique une clé primaire de la table, autrement dit ce qui identifie de manière unique un élément de cette table. Parfois il faut plus d'un élément pour identifier de manière unique ; c'est le cas dans les tables Ecrire et Emprunt.
Un **attribut précédé d'un dièse** indique une clé étrangère d'une autre table (en principe une clé primaire de celle-ci).

c. Création des tables

Voici les instructions en langage SQL de création des tables :

```
CREATE TABLE Eleve (num_etu INT PRIMARY KEY,
                    nom VARCHAR(90),
                    prenom VARCHAR(90),
                    classe VARCHAR(90));
CREATE TABLE Livre (isbn CHAR(14) PRIMARY KEY,
                    siret INT REFERENCES Editeur(siret),
                    titre VARCHAR(300),
                    annee INT);
CREATE TABLE Editeur (siret INT PRIMARY KEY,
                      nom VARCHAR(90));
CREATE TABLE Auteur (a_id INT PRIMARY KEY,
                     nom VARCHAR(200));
CREATE TABLE Ecrire (a_id INT REFERENCES Auteur(a_id),
                     isbn CHAR(14) REFERENCES Livre(isbn),
                     PRIMARY KEY (a_id,isbn));
CREATE TABLE Emprunt (isbn CHAR(14) REFERENCES Livre(isbn),
                      num_etu INT REFERENCES Eleve(num_etu),
                      date_ret DATE,
                      PRIMARY KEY (isbn));
```

IV. Systèmes de gestion

1. Introduction

Dans une base de données, l'information est stockée dans des fichiers, mais à la différence des fichiers au format CSV, il n'est pas possible de travailler sur ces données avec un simple éditeur de texte. Pour manipuler les données présentes dans une base de données (écrire, lire ou encore modifier), il est nécessaire d'utiliser un type de logiciel appelé « *système de gestion de base de données* » très souvent abrégé en SGBD. Il existe une multitude de SGBD : des gratuites, des payantes, des libres ou bien encore des propriétaires. Les SGBD permettent de grandement simplifier la gestion des bases de données :

- Elles permettent de gérer la **lecture, l'écriture** ou **la modification** des informations contenues dans une base de données.
- Elles permettent également 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.
- D'autre part, le système doit assurer la **persistance des données**. Ceci consiste à garder en mémoire des versions antérieures lorsque des modifications sont effectuées. Avec une structure de données persistante, les modifications ne sont pas effectuées en place, la structure est immuable.
- 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. Tout cela est très complexe à gérer ; en effet, toute modification de la base de données présente sur l'ordinateur A doit entraîner la même modification de la base de données présente sur l'ordinateur B. Cette synchronisation entre A et B doit se faire le plus rapidement possible, il est fondamental d'avoir des copies parfaitement identiques en permanence. Les SGBD ont donc aussi pour rôle d'assurer la maintenance des différentes copies de la base de données.
- Plusieurs personnes peuvent avoir besoin d'accéder aux informations contenues dans une base de données en même temps. Cela peut parfois poser problème, notamment si les deux personnes désirent modifier la même donnée au même moment (on parle d'**accès concurrent**). Ces problèmes d'accès concurrent sont aussi gérés par les SGBD.
- Enfin, l'un des attendus principaux est une **efficacité de traitement des requêtes**. Lorsque ces données se comptent pas milliers ou par millions, il faut disposer d'algorithmes performants et de structures de données adaptées. Un point important se trouve dans la manière de stocker les différentes clés primaires. Celles-ci sont en fait stockées sous une forme avancée d'arbres binaires de recherche, dans laquelle l'insertion et la suppression sont optimisées. Il existe d'autres notions encore plus complexes, comme le hachage, qui dépassent nettement le cadre du programme de NSI.

Comme nous venons de la voir, les SGBD jouent un rôle fondamental. L'utilisation des SGBD explique en partie la supériorité de l'utilisation des bases de données sur des solutions plus simples à mettre en œuvre mais aussi beaucoup plus limitées comme les fichiers au format CSV.

Nous parlerons plus loin de l'utilisation de SQLite, qui est une bibliothèque directement intégrable dans un programme, notamment en Python. Il existe d'autres SGBD, les plus connus étant MySQL et PostgreSQL, qui fonctionnent sur des serveurs, en relation donc avec le Web. L'utilisation de ces derniers peut s'effectuer à l'aide de code en PHP

2. Transactions

Supposons que l'on veut supprimer le livre *Les Aventures de Hucklberry Finn*, dont l'ISBN est '978-2081509511', de la base de données du CDI. Il faut, en plus de supprimer le livre de la table `Livre`, supprimer les relations correspondantes de la table `Ecrire` et, si on a supprimé le dernier livre d'un auteur, supprimer cette auteur de la base.
Ce processus s'exprime par plusieurs ordres SQL :

```
-- supprime les relations dans la table Ecrire
DELETE FROM Ecrire WHERE isbn = '978-2081509511';
-- supprime les auteurs qui n'apparaissent plus dans la table Ecrire
DELETE FROM Auteur
        WHERE NOT (a_id IN (SELECT a_id FROM Ecrire));
-- supprime le livre de la table Livre
DELETE FROM Livre WHERE isbn = '978-2081509511';
```

Ces trois ordres forment un tout et ne doivent pas être dissociés. En effet, considérons la situation suivante : un élève a reposé le livre directement en rayon sans passer par le documentaliste pour le rendre. Il reste alors dans la table `Emprunt` une référence vers l'ISBN du livre.

```
INSERT INTO Emprunt VALUES
        ('978-2081509511',166903291091,'2020-02-01');
DELETE FROM Ecrire WHERE isbn = '978-2081509511';
DELETE FROM Auteur
        WHERE NOT (a_id IN (SELECT a_id FROM Ecrire));
DELETE FROM Livre WHERE isbn = '978-2081509511';
```

On obtient le message d'erreur suivant :

```
FOREIGN KEY constraint failed:
DELETE FROM Livre WHERE isbn = '978-2081509511';
```

Nos données sont dans un état incohérent car les deux premiers ordres `DELETE` sont exécutés, retirant de la base l'auteur du livre et la relation entre le livre et son auteur, alors que le dernier ordre à échoué et le livre est toujours présent dans la base. On souhaite donc que que, si l'un des trois ordres échoue, les trois ordres soient annulés. Cette notion fondamentale des SGBD s'appelle une *transaction*.

```
BEGIN TRANSACTION;
DELETE FROM Ecrire WHERE isbn = '978-2081509511';
DELETE FROM Auteurs
        WHERE NOT (a_id IN (SELECT a_id FROM Ecrire));
DELETE FROM Livre WHERE isbn = '978-2081509511';
COMMIT;
```

Pour déclarer qu'une suite d'ordres est une transaction, il suffit de la faire précéder des mots clé `BEGIN TRANSACTION` et de la conclure avec `COMMIT` afin de la valider.
Si la transaction échoue, on utilise la commande `ROLLBACK` pour l'annuler.

3. Interaction avec Python

a. Exécution de requêtes

Un programme interagissant avec une SGBD effectue généralement les actions suivantes :

1. Connexion au SGBD. C'est lors de cette phase que l'on spécifie où se trouve la base de données et, en cas de nécessité, le nom d'utilisateur et le mot de passe pour y accéder.
2. Envoi d'ordres au SGBD. On effectue entre autres des requêtes SQL.
3. On récupère les données correspondant aux résultats dans des structures de données du langage.
4. On peut ensuite exécuter du code Python sur les données récupérées.

Il faut utiliser un module Python qui nous permet de réaliser ces actions. Nous allons ici utiliser `sqlite3` ; pour une autre SGBD il faudra utiliser un autre module. Cependant Python utilise une interface *unifiée* d'accès aux bases de données ; ainsi, même pour des SGBD différents, les méthodes Python sont toujours les mêmes.

```
import sqlite3 as sgbd

# connexion à la base de donnée
cnx = sgbd.connect('bdd_cdi.db')

# envoi d'un ordre
c = cnx.cursor()
c.execute("SELECT * FROM Livre WHERE titre LIKE '%Astérix%'")

# récupération et utilisation des données
for l in c.fetchall():
    print(l[0],l[2])

# fermeture de la connexion
cnx.close()
```

La variable `cnx` est un objet représentant la *connexion* à la SGBD. La création d'un curseur `c` à l'aide de la méthode `cursor()` permet d'interagir avec la SGBD. Les deux principales méthodes sont :

- `execute(s,p)` permet d'exécuter un ordre SQL `s`. Le paramètre `p` est un tableau de valeurs Python.
- `fetchall()` renvoie tous les résultats du dernier ordre passé, sous la forme d'un tableau de tuples de valeurs. Chaque tuple représente une ligne du résultat de la requête.

b. Ordres paramétrés

Une fonctionnalité importante est la possibilité de pouvoir insérer dans des ordres SQL des valeurs venant du monde Python, par exemple des saisies d'utilisateur.

```
import sqlite3 as sgbd
cnx = sgbd.connect('bdd_cdi.db')
# chaîne de caractères saisie par l'utilisateur
texte = input("Texte à rechercher dans le titre :")
# motif à rechercher dans les titres
motif = '%'+texte+'%'

c = cnx.cursor()
c.execute("SELECT titre FROM Livre WHERE titre LIKE ?" , [motif])
for l in c.fetchall():
    print(l)
cnx.close()
```

Les '?' prennent les valeurs du tableau donné en deuxième paramètre dans le même ordre.

c. Transactions

Les transactions sont une série d'ordres que l'on exécute à l'intérieur d'un bloc `try` ; la commande `ROLLBACK` est quand elle à l'intérieur du bloc `expect` de sorte que si la transaction échoue, la base de données est dans son état antérieur.

```
import sqlite3 as sgbd

cnx = sgbd.connect("cdi.db")

c = cnx.cursor()

try:
    c.execute("BEGIN TRANSACTION")
    c.execute("INSERT INTO Emprunt VALUES
    ('978-2081509511',166903291091,'2020-02-01');")
    c.execute("DELETE FROM Ecrire WHERE isbn = '978-2081509511';")
    c.execute("DELETE FROM Auteur
                    WHERE NOT (a_id IN (SELECT a_id FROM Ecrire));")
    c.execute("DELETE FROM Livre WHERE isbn = '978-2081509511';")
    c.execute("COMMIT")

except sgbd.Error as er:
    print('SQLite error: %s' % (' '.join(er.args)))
    c.execute("rollback")

cnx.close()
```