Introduction : le développement des traitements informatiques nécessite la manipulation de données de plus en plus nombreuses. Leur organisation et leur stockage constituent un enjeu essentiel de performance. Le recours aux bases de données relationnelles est aujourd'hui une solution très répandue. Ces bases de données permettent d'organiser, de stocker, de mettre à jour et d'interroger des données structurées volumineuses utilisées simultanément par différents programmes ou différents utilisateurs.

Cela est impossible avec les représentations tabulaires étudiées en classe de première, comme un tableur ou la bibliothèque « pandas » de Python.

Des systèmes de gestion de bases de données (SGBD) de très grande taille (de l'ordre du pétaoctet) sont au centre de nombreux dispositifs de collecte, de stockage et de production d'informations. L'accès aux données d'une base de données relationnelle s'effectue grâce à des requêtes d'interrogation et de mise à jour qui peuvent par exemple être rédigées dans le langage **SQL** (Structured Query Language). Les traitements peuvent conjuguer le recours au langage SQL et à un langage de programmation, comme **PHP** ou **Python**.

1 TP : mesurer la difficulté de gérer des données avec un tableur

Pour découvrir la notion de SGDB et de langage SQL, nous allons traiter les données concernant les passagers présents à bord du Titanic (pas le personnel de bord). Voici le début de ce fichier en question.

```
classe;survie;nom;sexe;age;tarif
1;1;Allen, Miss. Elisabeth Walton;2;29;211
1;1;Allison, Master. Hudson Trevor;1;1;152
1;0;Allison, Miss. Helen Loraine;2;2;152
1;0;Allison, Mr. Hudson Joshua Creighton;1;30;152
```

1. Ouvrir le fichier **titanicCSV** avec un simple traitement de texte, et décrire précisément la structure de ce fichier, son contenu et la signification des valeurs présentes. Vous expliquerez la signification complète des trois premières lignes.
2. Démarrer le logiciel tableur OpenOffice Calc, puis importer le fichier **titanicCSV** pour qu'il puisse s'afficher correctement. Décrire la méthode utilisée.
3. Déterminer le nombre de passagers qui ont survécu, ainsi que le taux de survie chez les passagers dans cette catastrophe.
4. Lors d'un naufrage, la priorité (en théorie) pour le capitaine est de sauver les femmes et les enfants d'abord. Pour savoir si ce principe est respecté sur le Titanic, déterminer le taux de survie des femmes, puis celui des hommes, indifféremment de leur classe sociale. Décrire en quelques lignes les outils qui permettent d'y parvenir, même si c'est fastidieux.

Conclusion : le tableur, l'un des plus anciens logiciels professionnel mis à la disposition du grand public fin des années 1970, est un outil pratique mais vite dépassé dès que les données sont volumineuses, et que l'on souhaite effectuer des traitements complexes. Il est cependant pratique en bureautique simple, mais aussi pour des tâches plus complexes comme un « mailing ». Depuis, des langages comme **Visual Basic** permettent d'étendre ses capacités, mais son utilisation rencontre de nombreuses limites.
Pour finir cette partie, une définition :

Définition : relation et bases de données relationnelles

Lorsque des données sont rangées selon différents **descripteurs** (colonnes), chaque ligne étant un **enregistrement** (ligne), c'est à dire sous forme de table, alors on définit une **relation**. Si des données sont rangées dans plusieurs tables, alors on définit une **base de données relationnelle**

2 Découvrir le langage structuré de requête (SQL) pour interroger une base de données

2.1 TP : interrogation d'une base de données : les passagers du Titanic

Pour cette découverte, nous allons reprendre des données sur les passagers du Titanic. Nous savons que l'utilisation de ces données se révèle difficiles du fait du nombre d'enregistrements, qui dépasse 1000. Le parcours à la main ou l'utilisation d'un tableur sont inappropriés. Pour rendre les données exploitables, elles ont été chargées dans **Système de Gestion de Bases de Données (SGDB)**. Pour faire simple, un SGDB est un logiciel que l'on installe, rien de sorcier. La procédure de chargement des données dans le SGDB sera décrite un peu plus loin dans le cours. Une fois ce travail préparatoire fait, il suffit d'utiliser des **requêtes**, qui sont en fait des phrases d'interrogation rédigées suivant un vocabulaire et une grammaire convenus. Pour faciliter la chose, l'interrogation et la visualisation des données se fait avec un navigateur WEB, et le résultat s'affiche sous forme de tableau HTML. Précisément, le SGDB est **MySQL**, et les requêtes lui sont transmises à partir d'un formulaire HTML, puis adressées à MySQL avec le langage PHP.

► *En classe nous ferons le schéma d'organisation de fonctionnement.*

1. Rendez-vous sur le site `http://77.140.110.245/pro/`, puis cliquer sur le menu, choisir « les passagers du Titanic ». Vous obtenez la page suivante :

2. Lire attentivement le descriptif, puis saisir les requêtes demandées, et observer les résultats obtenus.

3. Reprendre les questions de la première partie, puis expliquer quelles requêtes permettent d'obtenir les résultats demandés aux questions 3 et 4.

4. Démarche personnelle : proposer une problématique sur les passagers du Titanic, et une (ou des) requêtes qui permettent d'y répondre. Cette problématique pourra faire intervenir l'âge, la classe, le sexe, la survie...

2.2 Le fonctionnement de base de SELECT

Afin d'extraire les informations qui nous intéressent d'une base de données, on rédige une (ou plusieurs) requêtes. Pour que cette démarche soit universelle, il faut disposer d'un langage de requête. Très vite, un langage dédié s'est détaché, et est maintenant reconnu par la plupart des SGBD : il s'agit de **SQL** (Structured Query Language). Même s'il est relativement standard, il ne l'est pas en fait complètement. Certains éditeurs développent des logiciels avec des fonctionnalités SQL spécifiques, qui sont parfois reprises par les autres, parfois pas. On peut notamment citer la commande **AUTOINCREMENT**, présente dans de nombreuses distributions, mais qui ne fait pourtant pas partie du standard SQL. Ainsi devrait-on plus parler de « dialectes » SQL plutôt que de langage ! Concrètement il y a donc une partie « standard » pour SQL, dont la documentation contient 5000 pages (!), et des parties « optionnelles », dont certaines sont pas libres, donc rendent les utilisateurs dépendants...

SQL n'est en fait pas qu'un langage de requête ; il est constitué de 4 parties : le Langage d'Interrogation de Données (LID) : pour retrouver des informations dans la base de données ; le Langage de Manipulation de Données (LMD) : pour modifier les données de la base ; le Langage de Description de Données (LDD) : pour définir la structure d'une base de données et enfin le Langage de Contrôle de Données (LCD) : pour définir les droits d'accès à la base.

Dans ce cours nous aborderons les aspects LID, LMD et LDD, mais nous ne verrons l'aspect LCD que de manière ponctuelle ou en projets.

SQL : Langage Structuré de Requête

Le langage **SQL** (Structured Query Langage) est un ensemble de commandes qui permettent de :

1. Décrire l'organisation de données en créant des tables (CREATE TABLE) ;
2. Insérer (INSERT) des enregistrement dans des tables ;
3. Modifier (UPDATE) des enregistrements existants ;
4. Supprimer (DELETE)des enregistrements ;
5. Sélectionner (SELECT) des données.

Dans cette partie, c'est la dernière fonctionnalité qui va être étudiée, avec la commande SELECT. C'est la plus couramment utilisée. A partir du moment où des données sont rangées dans les tables d'une base de données, et que l'on souhaite extraire des données qui nous intéressent, on utilise cette commande.

Le modèle de fonctionnement est le suivant :

SELECT *noms des champs* FROM *nom de la table* ;

Littéralement, cette requête SQL signifie « sélectionner parmi les enregistrements les données correspondantes aux champs voulus qui se trouvent dans le table indiquée ». Le résultat retourné est un tableau. Cette commande permet de sélectionner une ou plusieurs colonnes d'une table. Les noms des champs sont séparés par une virgule. En SQL, la fin d'une requête est marquée par le point-virgule.

2.3 Exemples simples de la commande SELECT, projection

Pour illustrer le fonctionnement de la commande SELECT, nous utiliserons la table **clients** suivante :

identifiant	prenom	nom	ville	naissance	email
1	Pierre	Dupond	Paris	1984	pierredupond@gmail.com
2	Sabrina	Durand	Nantes	2001	s.durand@free.fr
3	Julien	Martin	Lyon	1996	Julien69Martin@facebook.fr
4	David	Dubois	Marseille	1989	davidb13@orange.fr
5	Marie	Leroy	Paris	1975	marie-leroy@sfr.fr

Voici quelques exemples.

- Si on veut afficher toute la table **clients**, la commande est : `SELECT * FROM clients;`
  L'étoile signifie donc que tous les champs sont à prendre ;
- Si on veut afficher seulement le prénom et le nom, on parle alors de **projection**. La requête est : `SELECT prenom, nom FROM clients;`

Exercice 1.

1. écrire le résultat renvoyé par : `SELECT email FROM clients;`
2. écrire une requête qui renvoie uniquement les âges ;
3. écrire une requête qui renvoie les noms des clients et leur ville .

2.4 Commandes avancées avec SELECT : filtrer, ordonner, limiter

Il existe plusieurs commandes (on dit aussi clauses) qui permettent de **filtrer** (WHERE) et **ordonner** (ORDER BY), et de **limiter** (LIMIT) les données que l'on souhaite extraire d'une base.
Voici le modèle à retenir :

```
SELECT *
FROM table
WHERE condition1 AND condition2
ORDER BY expression
LIMIT nombre;
```

Quelques commentaires sur les nouvelles commandes abordées ci-dessus.

► La commande **WHERE** permet d'extraire dans une requête les enregistrements qui respectent une condition. Cela permet d'obtenir uniquement les informations désirées. Par exemple :

- `SELECT nom, prenom FROM clients WHERE naissance < 2000;`
  renvoie les noms et prénoms des clients qui sont nés avant 2000.
- `SELECT nom, prenom FROM clients WHERE ville = 'Paris';`
  renvoie les noms et prénoms des clients qui sont à Paris. **Il est important que la chaîne de caractère que l'on cherche soit encadrées par des apostrophes**, ici 'Paris'.

La condition qui faut saisir après WHERE doit contenir le nom d'un champ, un opérateur et enfin une valeur. Pour information voilà la liste des opérateurs qui existent en SQL :

Opérateur	Description
=	Égale
<>	Pas égale
!=	Pas égale
>	Supérieur à
<	Inférieur à
>=	Supérieur ou égale à
<=	Inférieur ou égale à
IN	Liste de plusieurs valeurs possibles
BETWEEN	Valeur comprise dans un intervalle donnée (utile pour les nombres ou dates)
LIKE	Recherche en spécifiant le début, milieu ou fin d'un mot
IS NULL	Valeur est nulle
IS NOT NULL	Valeur n'est pas nulle

Si la sélection porte sur **plusieurs conditions**, il est possible de les combiner avec les opérateurs logiques **AND** et **OR** : on appelle cela une **sélection multiple**. Par exemple, la commande :
```
SELECT * FROM clients WHERE ville = 'Paris' AND naissance >= 1980;
```
renvoie les clients dont la ville est Paris et qui sont nés en 1980 ou après.
Dans la combinaison des commandes logiques AND et OR, on peut utiliser les **parenthèses** pour bien marquer les priorités dans opérations logiques quand c'est nécessaire.

Les opérateurs de comparaison classiques sont parfois un peu restrictifs : par exemple si on cherche à extraire les adresses email des clients qui ont leur adresse chez Google, le « = » ne convient pas.
La clause **LIKE** permet, un peu comme les expressions régulières dans de nombreux langages de programmation, de définir **le modèle** qui correspond à ce qui est cherché. Voici les modèles que l'on doit connaître.

- LIKE '%a' : le caractère "%" est un caractère joker qui remplace tous les autres caractères.
  Ainsi, ce modèle permet de rechercher toutes les chaines de caractère qui se termine par un "a".
- LIKE 'a%' : ce modèle permet de rechercher toutes les lignes de "colonne" qui commence par un "a".
- LIKE '%a%' : ce modèle est utilisé pour rechercher tous les enregistrement qui utilisent le caractère "a".
- LIKE 'pa%on' : ce modèle permet de rechercher les chaines qui commence par "pa" et qui se terminent par "on", comme "pantalon" ou "pardon".
- LIKE 'a_c' : le caractère "_" (underscore) peut être remplacé par n'importe quel caractère, mais un seul caractère uniquement (alors que le symbole pourcentage "%" peut être remplacé par un nombre incalculable de caractères . Ainsi, ce modèle permet de retourner les lignes "aac", "abc" ou même "azc".

La requête qui convient est donc :
```
SELECT email FROM clients WHERE email LIKE '%gmail.com';
```

Enfin, pour **ne pas tenir compte de la casse**, on peut faire une égalité stricte entre les valeurs converties en majuscule grâce à la fonction **UPPER**. Par exemple la requête :
```
SELECT * FROM client WHERE UPPER(nom) = UPPER('durand');
```
permet de trouver dans la base tous les clients dont le nom est « Durand » ou « DURAND » ou « durand »...

► 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 (ASC) ou descendant (DESC).
Voici le modèle de cette commande :

```
SELECT colonne1, colonne2, colonne3
FROM table
ORDER BY colonne1 DESC, colonne2 ASC;
```

Dans ce modèle, les données seront rangées avec selon la colonne 1 par ordre décroissant, puis selon la colonne 2 par ordre croissant. A noter qu'il n'est pas obligatoire d'écrire ASC, car par défaut c'est cet ordre qui est utilisé. Par exemple la requête :
`SELECT nom, prenom ORDER BY naissance DESC;` renvoie les noms et prénoms des clients, du plus jeune au plus vieux.

► La commande **LIMIT** permet de limiter le nombre de résultats que l'on souhaite obtenir. Par exemple la requête :
`SELECT * FROM clients LIMIT 2;` renvoie les deux premiers clients.

► La commande **DISTINCT** sert à éliminer les lignes en double. Par exemple le requête :
`SELECT villes FROM clients;` affichera deux fois la ville de Paris, ce qui peut être évité avec la requête :
`SELECT DISTINCT villes FROM clients;`

Exercice 2.
Décrire ce que permettent d'obtenir les requêtes suivantes :

1. `SELECT nom FROM clients WHERE UPPER(ville) = UPPER('paris') ORDER BY nom;`
2. `SELECT email FROM clients WHERE UPPER(nom) LIKE UPPER('d%');`

Exercice 3.
Écrire les requêtes qui permettent :

1. d'extraire les 10 premiers noms et prénoms des clients qui commencent par 'a', rangés par ordre croissant du nom ;
2. d'extraire l'adresse email des trois plus jeunes clients, rangés du plus jeune au plus vieux.

2.5 SELECT et les fonctions d'agrégation

Il existe également des fonctions d'agrégation dans SQL qui permettent d'effectuer des **statistiques** sur les données de la base. Voici quelques exemples :

- `SELECT COUNT(*) FROM clients;` donne le nombre d'enregistrements dans la table ;
- `SELECT COUNT(DISTINCT ville) FROM clients;` donne le nombre de villes distinctes dans la table ;
- `SELECT MAX(naissance) FROM clients;` donne l'année de naissance du plus jeune client. Il existe aussi la commande **MIN**.

Précisons que la fonction **AVG** permet d'obtenir la moyenne de valeurs, la fonction **SUM** permet de d'additionner des valeurs, et que la fonction **STD** permet d'obtenir l'écart-type.

Exercice 4.
Donner une requête qui donne l'année moyenne de naissance des clients.

Extraits du programme : identifier les composants d'une requête. Construire des requêtes d'interrogation à l'aide des clauses du langage SQL : SELECT, FROM, WHERE. On peut utiliser DISTINCT, ORDER BY ou les fonctions d'agrégation sans utiliser les clauses GROUP BY et HAVING.

3 Les systèmes de gestion de bases de données relationnelles

Définition d'un SGDB

Nombreux sont les logiciels informatiques nécessitant une base de données. Afin de ne pas réinventer la roue à chaque fois, **des logiciels dédiés à la gestion de bases de données ont été développés**. La plupart de ces logiciels gèrent des bases de données relationnelles. Ils sont en général implantés sous la forme d'un logiciel serveur indépendant (pouvant tourner sur la même machine que le logiciel, ou bien sur une autre machine). On appelle ces logiciels des Systèmes de Gestion de Bases de Données (SGBD).

3.1 Caractéristiques d'un SGBD

Quels sont les services que l'ont peut attendre d'un SGDB ? Les SGBD doivent proposer un certain nombre de caractéristiques :

- Disposer d'un moyen de créer une base de données (relationnelle)
- Proposer des moyens pour modifier le contenu d'une base de données relationnelle, tout en contrôlant le respect de contrainte (par exemple les contraintes de référence)
- Proposer un langage de requête pour consulter les données présentes dans la base
- Gérer la persistance des données
- Gérer efficacement des bases de données potentiellement très grandes
- Gérer des accès concurrents à la base
- Proposer une gestion des droits d'accès à la base
- Déclencher automatiquement certaines tâches sur certaines actions
- Proposer des moyens pour limiter les risques de pertent de données en cas de panne ou fausse manipulation
- ...

3.2 Quelques exemples de SGBD

Il y a de nombreux SGBD sur le marché, avec leur avantages et leurs défauts. Voici une présentation sommaire des plus connus :

- **Oracle** : la référence en terme de SGBD relationnel professionnel. Une version gratuite existe depuis quelques années. Un peu lourd, pas en avance sur les évolutions, mais le plus robuste, le plus contrôlable à de nombreux points de vue.
- **MySql** : le plus connu depuis le développement des sites web. C'est un SGBD relativement léger, gratuit, de plus en plus complet - surtout depuis son rachat par Oracle - mais avec certains défauts. Notamment, dans ses premières versions, il ignorait totalement les contraintes de référence. Il manque aussi d'un certain nombre de fonctionnalités nécessaire au développement d'une base de qualité. Il n'est plus disponible sur Raspberry, il a été remplacé par MariaDB. MariaDB dernier est un "clone" libre de MySQL, qui a été racheté en 2010 par le géant Oracle. Les promoteurs de MariaDB veulent garantir la liberté totale du logiciel, avec une utilisation identique à MySQL.

- **PostGres** : un excellent compromis entre Oracle et MySql. Très fiable et efficace, moins lourd à administrer qu'Oracle et plus en avance sur certaines fonctionnalités, gratuit : de nombreuses entreprises se sont mises à l'utiliser
- **Sql Server** : le SGBD de Microsoft. Je ne le connais pas assez pour en parler
- **Access** : le SGBD pour les nuls de Microsoft. Beaucoup hésitent à le classer dans les SGBD. Mais pour une utilisation non professionnelle, il dépanne bien.
- **Libre Office Base** : le SGBD pour les nuls Open Source. Beaucoup hésitent à le classer dans les SGBD. Mais pour une utilisation non professionnelle, il dépanne bien.

3.3 Installation d'un SGDB : SQLite3

Tous ces SGBD fonctionnant sous la forme de serveurs, leur utilisation nécessitent plusieurs choses, que l'on retrouve dans les toutes les pages dédiées à l'installation d'un SGDB :

- Installer le SGBD en lui-même avec des droits de super-utilisateur ; suivant les cas, ce n'est pas toujours très simple ;
- Disposer d'une bibliothèque dédiée pour le langage de programmation choisi, le plus souvent PHP ou Python ; ce n'est pas toujours disponible ;
- Installer un serveur WEB (souvent Apache) pour réaliser des interfaces homme-machine via des pages WEB.

Pour simplifier les choses, dans le cadre de ce cours, nous avons choisi d'abord utiliser **SQLite** (version 3, voir le site http ://www.sqlite.org). C'est une sorte de SGBD léger, dédié aux bases de données embarquées dans l'application. En effet, aucun logiciel serveur n'est nécessaire, et il n'y a donc pas d'installation à faire. C'est un SGBD qui est par exemple très utilisée dans les applications mobiles. Il existe une bibliothèque dédiée dans Python : **sqlite3** (voir le site https ://docs.python.org/3.8/library/sqlite3.html pour son installation).

Même si SQLite3 est disponible sur de nombreux systèmes, nous ne décrirons ici que l'installation sous Linux et donc aussi Raspberry. Voici la démarche à suivre :

1. Dans une fenêtre console en mode administrateur, saisir la commande Linux :
   `sudo apt-get install sqlite3`. Le téléchargement et l'installation se réalisent, avec un message le succès.
2. Démarrer le logiciel avec la commande `sqlite3`. Vous devez alors voir l'invite de commande : `sqlite>`. Rien de très spectaculaire, mais c'est bon signe !
3. Saisir comme il est proposé la commande SQLite `.help`, elle vous donne une vue d'ensemble de toutes les commandes de ce logiciel. Nous en verrons quelques-unes.
4. Pour quitter le logiciel, on peut utiliser la commande `.quit` ou simplement taper « CTRL-D ».

Maintenant que tout est installé, nous allons voir comment créer une base de données, insérer des données, modifier et supprimer des enregistrements.

Extraits du programme : identifier les services rendus par un système de gestion de bases de données relationnelles : persistance des données, gestion des accès concurrents, efficacité de traitement des requêtes, sécurisation des accès. Il s'agit de comprendre le rôle et les enjeux des différents services sans en détailler le fonctionnement.

4 Le langage SQL : création d'une base de données, insertion d'un enregistrement, mise à jour ou suppression d'enregistrement

Maintenant que nous avons installé un SQDB, nous allons voir pas à pas comment créer une base de données, insérer des enregistrements, mettre à jour ou supprimer des enregistrements. C'est une démarche essentielle pour comprendre les étapes essentielles de la création d'une base de données :

1. la création de la base, et sa structure en précisant les tables et pour chacune les champs (colonnes) qui caractérisent chaque table ;
2. l'insertion des données sous forme d'enregistrements (lignes) ;
3. la mise à jour ou la suppression d'enregistrements.

4.1 Créer pas à pas une base de données

La démarche expliquée ci-dessous est un peu aride, mais elle est importante pour comprendre un SGDB.Un peu plus tard, nous verrons des moyens plus rapides et fonctionnels pour communiquer avec un SGDB. Nous verrons aussi qu'avant de nous lancer l'utilisation d'un SGDB, il est essentiel de réfléchir en amont sur l'organisation des données que l'on veut stocker : combien de tables, quels liens entre elles ... mais ce n'est pas la question pour le moment. Nous allons utiliser la table **clients** que voici :

idClient	prenom	nom	ville	naissance	email
1	Pierre	Dupond	Paris	1984	pierredupond@gmail.com
2	Sabrina	Durand	Nantes	2001	s.durand@free.fr
3	Julien	Martin	Lyon	1996	Julien69Martin@facebook.fr
4	David	Dubois	Marseille	1989	davidb13@orange.fr
5	Marie	Leroy	Paris	1975	marie-leroy@sfr.fr

1. Démarrer une console.
2. Créer un dossier dans votre home, que vous pourrez appeler sql, puis aller dans ce dossier.
3. Saisir la commande sqlite3 maBase.db. Elle démarre le SGDB, qui crée aussitôt une base de données appelée maBase.db. En même temps est créé le fichier maBase.db, qui sauvegarde les données à chaque modification de la base. C'est simple !
4. Maintenant nous allons décrire la table **clients**. Pour cela saisir la commande SQL :

```
CREATE TABLE clients (
idClient INT PRIMARY KEY NOT NULL,
prenom TEXT,
nom TEXT,
ville TEXT,
naissance INT,
email TEXT) ;
```

La syntaxe est facile à comprendre. Les commandes spécifiques au SQL sont écrites en majuscules, et les noms spécifiques à la table sont écrites en minuscules. Ce n'est pas une obligation, mais ça facilite la compréhension. Cette commande se traduit en français par « créer la table clients (dans maBase.db) qui contient le champ **idClient**, dont les valeurs sont des entiers (INT), et ce champ sert de clé primaire (PRIMARY KEY), et qu'il doit obligatoirement être renseigné (NOT NULL). Sans pour l'instant

entrer dans les détails, il faut comprendre que pour fonctionner correctement, une table doit comporter une **clé primaire**, c'est à dire un ou plusieurs champs qui permettent d'identifier de manière unique chaque enregistrement. Si une clé primaire n'est pas spécifiée, le SGDB renvoie un message d'erreur. Ensuite le champ **prenom** qui contient un texte, puis le champ **nom** qui est un texte, puis le champ **ville** qui est un texte, puis le champ **naissance** qui est un nombre entier, et enfin le champ **email** qui est un texte ».
Les types standards de données en SQL sont : **INT**, **VARCHAR(n) TEXT**, **BIT**, **DATE**, **TIME**, **REAL** principalement, mais il y en a beaucoup d'autres ! Le choix d'un type est important, il permet au moment de l'insertion d'un enregistrement de vérifier si les informations saisies respectent le type.

5. Pour vérifier que votre table **clients** est bien enregistrée, saisir la commande SQLite .tables sans oublier le point ! Cette commande SQLite donne la liste des tables dans la base. Vous devez lire comme réponse **clients**.

6. Pour vérifier que les champs de votre table sont correctement saisis, exécuter la commande SQLite .schema clients. Elle vous renvoie la structure de votre table **clients**.

7. Enfin, pour afficher la liste des requêtes ayant permis de créer toutes les tables de la base, il faut saisir la commande SQLite .fullschema .

4.2 Insertion des données dans une table

Maintenant que la structure de la table **clients** est donnée, on saisit la commande SQL suivante pour effectuer les enregistrements un à un.

1. Saisir la commande SQL :

```
INSERT INTO clients(idClient,prenom,nom,ville,naissance,email)
VALUES (1 , 'Pierre', 'Dupond',  'Paris', 1984, 'pierredupond@gmail.com');
```

Cette commande d'insertion se lit très facilement. Juste quelques détails pour éviter les erreurs de saisie : les valeurs numériques ne sont écrites avec des apostrophes autour, mais les valeurs sous forme de texte oui. De plus la ligne de commande se termine par un point-virgule. Enfin, il n'est pas nécessaire après le nom de la table **clients** d'écrire les noms des champs entre parenthèses...

2. A présent on peut avec la commande SQL SELECT * FROM clients ; sans oublier le point-virgule pour afficher le contenu de la table **clients** et vérifier que notre enregistrement est bien saisi.

3. Finir la saisie des enregistrements de la table.

4. Expérimenter les commandes SQL simples et avancées qui ont été vues avec cet exemple dans la partie précédente.

4.3 Modification des données dans une table

Imaginons que le client numéro 3, 'Julien', change d'adresse email, et que sa nouvelle adresse soit 'Julien-Martin@protonmail.com'. Voici la commande qu'il convient de saisir :

```
UPDATE clients SET email = 'Julien-Martin@protonmail.com' WHERE idclient = 3 ;
```

Saisir la commande SQLite SELECT * FROM clients ; pour visualiser le changement.

Le modèle de modification est donc le suivant :

UPDATE nomTable
SET $champ_x = val_x, champ_y = val_y$, ...
WHERE condition;

4.4 Suppression d'un enregistrement dans une table

Imaginons que le client numéro 5, 'Marie', souhaite se désabonner de la liste des clients. Voici la commande SQLite qu'il convient de saisir :

```
DELETE FROM clients WHERE icClient = 5 ;
```

Saisir la commande SQL `SELECT * FROM clients;` pour visualiser le changement.

Le modèle de suppression est donc le suivant :

DELETE FROM nomTable
WHERE condition;

4.5 Sauvegarde d'une base de données

Il est possible de sauvegarder une base de données au format SQL, c'est à dire avec toutes les commandes SQL qui permettent de créer la structure et insérer les enregistrements. Voilà les commandes SQLite3 à saisir :

1. saisir la commande SQLite3 `.output export.sql` qui spécifie que la sauvegarde se fera dans le fichier **export.sql** dans le dossier courant;
2. saisir la commande SQLite3 `.dump` qui génère le code nécessaire à la re-création de la base (création et remplissage des tables)
3. saisir la commande SQLite3 `.output` qui enregistre le .dump fait dans la fichier **export.sql**.

Si vous avez suivi toutes les instructions de cette partie, le contenu du fichier **export.sql**, visible avec la commande Linux `nano export.sql`, doit être :

```
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE clients (idClient int primary key, prenom text, nom text,ville text,
naissance int, email text);
INSERT INTO clients VALUES(1,'Pierre','Dupond','Paris',1984,
'pierredupond@gmail.com');
INSERT INTO clients VALUES(2,'Sabrina','Durand','Nantes',2001,'s.durand@free.fr');
INSERT INTO clients VALUES(3,'Julien','Martin','Lyon',1996,
'Julien-Martin@protonmail.com');
INSERT INTO clients VALUES(4,'David','Dubois','Marseille',1989,
'davidb13@orange.fr');
COMMIT;
```

On retrouve l'ensemble des commandes SQL vues. On trouve en plus la mention **PRAGMA foreign_keys=OFF;** qui signifie qu'il n'y a pas de clés étrangères dans cette base de données. Nous verrons plus tard l'utilité

d'utiliser des clés étrangères, quand des données sont réparties dans plusieurs tables liées entre elles. Enfin on peut voir les mentions **BEGIN TRANSACTION;** et **COMMIT;** qui marquent le début et la fin des commandes SQL.

Quand on importe un fichier SQL qui contient la structure de la base et les insertions d'enregistrements, il est possible que les tables existent déjà... Dans ce cas la requête est refusée. Il faut donc **supprimer** au préalables les tables existantes, avec la commande **DROP**. Pour information, voici le fichier qui correspond à la restauration de la configuration initiale de notre base de données **clients**.

```
DROP TABLE IF EXISTS clients  ;

CREATE TABLE clients (
        identifiant INT PRIMARY KEY NOT NULL,
        prenom TEXT,
        nom TEXT,
        ville TEXT,
        naissance INT,
        email TEXT ) ;

INSERT INTO clients
VALUES (1,'Pierre','Dupond','Paris',1984,'pierredupond@gmail.com') ;
INSERT INTO clients
VALUES (2 , 'Sabrina','Durand' , 'Nantes' , 2001 , 's.durand@free.fr') ;
INSERT INTO clients
VALUES (3 , 'Julien' , 'Martin' , 'Lyon'   ,  1996 , 'Julien69Martin@facebook.fr') ;
INSERT INTO clients
VALUES (4 , 'David' , 'Dubois' , 'Marseille' , 1989 , 'davidb13@orange.fr') ;
INSERT INTO clients
VALUES (5 , 'Marie' , 'Leroy' , 'Paris' , 1975 , 'marie-leroy@sfr.fr') ;
```

4.6 Importer une base de données

De la même manière que l'on peut exporter une base de données (structure + enregistrement) dans un fichier SQL, il est possible d'en **importer** une avec la commande SQLite : .read fichier.sql

Tous les SGDB possèdent des fonctions d'export et d'import au format SQL. Certains, comme MySQL peuvent directement importer des fichiers au format CSV. C'est pratique car beaucoup de données sont disponibles dans ce format, mais la structure de la base n'est pas forcément bien écrite...

Exercice 1.

Dans cet exercice nous reprenons les données des passagers du Titanic dans le fichier **titanicCSV.csv** dont voici le début :

```
classe;survie;nom;sexe;age;tarif
1;1;Allen, Miss. Elisabeth Walton;2;29;211
1;1;Allison, Master. Hudson Trevor;1;1;152
```

1. Écrire le code SQL qui permet de décrire la structure de cette base de données.
2. Écrire les commandes SQL qui correspondent à l'insertion des deux premier enregistrements.
3. Il serait absurde de saisir tous les enregistrements à la main. Écrire un programme en Python qui récupère les valeurs de chaque enregistrement dans le fichier titanicCSV.csv, et écrit dans un fichier (titanicSQL.sql) la commande SQL correspondante.

   Vous nommerez ce programme CSV_to_SQL.py.
4. Compléter le fichier titanicSQL.sql pour qu'il soit complet : structure + enregistrements.
5. Importer le fichier titanicSQL.sql avec SQLite3, et vérifier avec quelques requêtes que tout fonctionne correctement.

Extraits du programme : savoir distinguer la structure d'une base de données de son contenu. Construire des requêtes d'insertion et de mise à jour à l'aide de : UPDATE, INSERT, DELETE.

5 Utilisation d'une base de données avec un programme Python

Dans la pratique,il est très rare de dialoguer directement avec un SGBD en SQL, éventuellement au moment de l'installation, de la paramétrisation, de l'attribution des droits aux différents utilisateurs, mais au delà c'est une méthode fastidieuse et peu efficace. On préfère utiliser un langage de programmation pour automatiser les actions sur le SGBD. Le couple le plus connu est **MySQL** et **PHP**, qui associé avec le gestionnaire WEB **PhpMyAdmin** permet de superviser assez facilement le déployement des bases de données.

5.1 Création de la base et insertion des enregistrements

Dans cette partie nous utiliserons **Python**, qui possède une bibliothèque **SQLite** pour transmettre les actions au SGDB. Le code contient tous les éléments de base pour une bonne pratique. Ci-dessous, le code pour se connecter au SGDB, créer la structure de la base de données, et insérer les premiers enregistrements. Quelques commentaires :

- on importe la bibliothèque **SQLite**. Il existe des bibliothèques pour d'autres SGBD. On a choisi de faire appel à cette bibliothèque avec le nom générique **sgbd** ;
- on définit un objet **connexion**, avec la méthode **connect** de **sgbd**. Avec SQLite, il suffit de donner le nom de la base de donnée. Si le fichier portant ce nom est présent dans le même dossier, il est ouvert, sinon un nouveau fichier est créé portant ce nom. Si on utilise un SGBD plus complet, il faut en plus du nom préciser l'hôte, le nom d'utilisateur et le mot de passe.
- Avec notre **connexion**, on définit l'objet **curseur** avec la méthode **cursor()**.
- ensuite avec cet objet **curseur** et sa méthode **execute**, on transmet les commandes SQL comme des chaines de caractères, entre apostrophes si elles sont courtes, et balisées avec trois apostrophes si elles s'étalent sur plusieurs. On remarque qu'il n'est plus nécessaire d'écrire le point-virgule ;
- enfin, avec l'objet **connexion**, on appelle les méthodes **commit()** et **close()** pour activer puis clore la connexion.

Voici la démarche à suivre :

- créer, dans le dossier **SQL**, un nouveau dossier **SQL-Python** ;
- dans ce dossier, copier ce code dans un ficher que vous pouvez nommer creer_BDD.py ;
- exécuter le code, vérifier que le fichier de la base de données est bien créé ;
- finir la saisie des enregistrements en observant la méthode alternative bien pratique !

Le code de base pour créer et alimenter une bases de données en **Python** :

```
import sqlite3 as sgbd

# connexion à la base, par exemple avec SQLite3 :
connexion = sgbd.connect('nomBase.db')
# connexion = sgbd.connect(name = "...", host = "...", user ="...", password = "...")

curseur = connexion.cursor()

# activation de la vérification des contraintes de clé étrangère
curseur.execute('PRAGMA foreign_keys = OFF')

# suppression éventuelle de l'ancienne table
curseur.execute('DROP TABLE IF EXISTS clients')

# création des tables
curseur.execute('''
CREATE TABLE clients (
idClient INT PRIMARY KEY NOT NULL,
prenom TEXT,
nom TEXT,
ville TEXT,
naissance INT,
email TEXT )
''')

curseur.execute('''
INSERT INTO clients
VALUES (1,'Pierre','Dupond','Paris',1984,'pierredupond@gmail.com')
''')

# validation
connexion.commit()

# déconnexion
connexion.close()
```

Le code alternatif pour alimenter une base de données en **Python** , avec un tableau de **Tuples** et **executemany**, à rajouter après le premier enregistrement.

```
listeClients = [
(2 , 'Sabrina','Durand' , 'Nantes' , 2001 , 's.durand@free.fr') ,
(3 , 'Julien' , 'Martin' , 'Lyon'   ,  1996 ,'Julien69Martin@facebook.fr') ,
(4 , 'David' , 'Dubois' , 'Marseille' , 1989 , 'davidb13@orange.fr') ,
(5 , 'Marie' , 'Leroy' , 'Paris' , 1975 , 'marie-leroy@sfr.fr')
 ]

curseur.executemany("INSERT INTO clients VALUES(?,?,?,?,?)", listeClients)
```

5.2 Utilisation de Python pour interroger et exploiter les résultats d'une requête

Dans cette partie, on suppose que la base de données est créée et alimentée. L'objectif est ici de comprendre comment mettre en œuvre la fonction **SELECT** dans un programme **Python**. Contrairement à la création et l'alimentation de la base, il y a ici un retour attendu, qui doit être exploité. Le principe est le suivant : le résultat de la commande **SELECT** est transmis à **Python** sous forme d'un **itérable**, auquel on peut accéder avec la méthode **fetchall()** du curseur, ou bien simplement avec une boucle **for** sur le curseur renvoyé.

On peut remarquer que l'utilisation de la méthode **description** du curseur permet retrouver les noms des champs renvoyés. Voici la démarche à suivre :

- copier et coller le code dans un fichier que vous pourrez appeler interroger_BDD.py ;
- exécuter le code, d'abord avec la méthode **for**, puis avec la méthode **fetchall()** .
- essayer avec d'autres requêtes de sélection...

```
import sqlite3 as sgbd

# connexion à la base, par exemple avec SQLite3 :
connexion = sgbd.connect('nomBase.db')

curseur = connexion.cursor()

# envoi de la requête
curseur.execute('''
SELECT * FROM clients
''')

# affichage des champs
colonnes = [description[0] for description in curseur.description]
print(colonnes)

# affichage du résultat
for ligne in curseur:
        print(list(ligne))

# validation
connexion.commit()

#déconnexion
connexion.close()
```

```
# affichage du résultat avec l'utilisation de fetchall
for ligne in curseur.fetchall() :
        print(ligne[0], ligne[1], ...)
```

Pour conclure, même si aucun exemple n'est donné ici, il est possible de modifier ou supprimer des enregistrements, ce que vous pourrez tester en autonomie !

6 Utiliser une page web pour interagir avec une base de données

L'utilisation d'un SGBD puissant est quasiment hors de portée d'un utilisateur non initié. Néanmoins il est souvent utile que des personnes non initiées puissent les utiliser, pour saisir des données ou obtenir des données. Dans ce cas on fabrique une interface homme-machine (IHM) où l'utilisateur n'a plu qu'à cliquer sur des zones clairement identifiables. Il est possible de réaliser une IHM avec un gestionnaire de fenêtre comme **Tkinter** , ou mieux encore en utilisant les technologies WEB **HTML/CSS**. Cette dernière méthode permet en plus de rendre la base de données accessible à partir de n'importe quel ordinateur client connecté en réseau : c'est de la **programmation Web côté serveur**. Le principe général est :

- on installe un serveur WEB sur le serveur où est installé le SGBD ;
- on écrit une page WEB **index.htm** dans laquelle on fait des liens vers les différentes actions possibles sur la base de données ;
- pour chaque action sur la base prévue, on écrit une page WEB qui contient souvent un formulaire (**form**) , avec des zones de saisies, des menus déroulants, des boutons radios ... et finalement un bouton **submit** qui envoie les données en **POST** ou **GET** ;
- pour chaque formulaire, on écrit une page de **traitement**, en Python ou en PHP, qui effectue directement les actions souhaitées sur la base de données, et qui conclut par un message de succès si tout s'est déroulé comme prévu, et un lien vers le menu pour passer à une autre action.

Le fait de passer par un serveur WEB pour accéder à une base de données sur un serveur distant est pratique. Il est possible de se connecter directement en Python ou en PHP, mais ce n'est pas toujours évident. Par exemple avec MySQL, il faut changer un paramètre des fichiers de configuration pour autoriser les actions à partir d'un client distant. Ce n'est pas évident... Alors qu'en passant par un serveur WEB, pour le SGDB c'est comme si les requêtes venaient en local.

6.1 Création d'un serveur WEB

Pour savoir su un serveur WEB est installé sur votre machine, il suffit de démarrer un navigateur et de saisir l'URL http ://localhost. Le port utilisé est souvent le 80. Si le navigateur indique que la connexion a échoué, c'est qu'aucun serveur WEB n'est actif. On peut naturellement utiliser **Apache**, mais aussi utiliser Python et sa bibliothèque **http.server**. Le code ci-dessous permet, quand il est exécuté, de lancer un serveur WEB sur le port 8080.

- copier-coller le code suivant le dossier **SQL-Python** avec comme nom de fichier web_serveur.py ;
- exécuter ce code, et laisser la fenêtre de console ouverte tout le temps que vous souhaitez que le serveur fonctionne.

```
#!/usr/bin/python3

import http.server

PORT = 8888
server_address = ("", PORT)

server = http.server.HTTPServer
handler = http.server.CGIHTTPRequestHandler
handler.cgi_directories = ["/"]
print("Serveur actif sur le port :", PORT)

httpd = server(server_address, handler)
httpd.serve_forever()
```

Quelques commentaires :

- la première ligne sert à indiquer le chemin de Python ;
- la ligne handler.cgi_directories = ["/"] permet de spécifier le dossier dans lequel il doit aller chercher les scripts CGI. Par défaut, si on ne met rien, c'est le répertoire dans lequel est lancé le serveur qui est utilisé.
- il ne faut pas fermer la fenêtre de console du serveur pour la suite !

Nous allons vérifier que notre serveur fonctionne avec une page WEB statique. Pour cela :

- copier-coller le script ci-dessous dans un fichier appelé index.py dans le même dossier que précédemment.
- dans un navigateur internet, saisir l'url localhost :8888/index.py, et vérifier que ce message s'affiche dans votre navigateur.

```
#!/usr/bin/python3

page= '''
<html>
  <head>
    <title>Essai</title>
  </head>

  <body>
    <h1>Essai</h1>
    <p>Voici un exemple de page html.</p>
  </body>
</html>'''

print(page)
```

6.2 Création d'une page de formulaire

Nous allons donner comme exemple une page WEB qui propose à l'utilisateur d'ajouter un nouveau client dans la base de données. Nous utiliserons la base de données de la partie précédente.

- copier-coller le code ci-dessous dans un fichier appelé form_ajout.py ;
- démarrer un navigateur WEB, puis saisir l'adresse localhost :8888/form_ajout.py ;
- vérifier que le formulaire s'affiche correctement.

Attention : ce script doit avoir les **droits en exécution** (sous linux, faire un chmod 755 nomDuScript) !!

```
#!/usr/bin/python3

import sqlite3

connexion = sqlite3.connect('nomBase.db')

# on écrit un formulaire pour ajouter un client
print("Content-type: text/html; charset=utf-8")
print("""
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Ajouter un client</title>
</head>
<body>
Compléter les champs pour ajouter un client :
<br>

<form action="/traitement_ajout.py" method="get">
<br> Prénom : <input type = "text" name="prenom">
<br> Nom : <input type = text" name = "nom">
<br> Ville : <input type = "text" name = "ville">
<br> Année de naissance : <input type = "text" name = "naissance">
<br> Email : <input type = "text" name = "email">
<br><input type="submit" value="Ajouter">
</form>

</body>
</html>
""")
```

On peut remarquer qu'on ne demande pas à l'utilisateur l'index (**idClient**) : c'est normal, car en général c'est un champ qui doit être géré automatiquement, soit avec la clause **AUTOINCREMENT**, soit pendant l'insertion dans la page de traitement. Pour cela, il faudra, en plus du POST ou du GET, récupérer l'idClient maximal, puis ajouter une unité.

6.3 Création d'une page pour le traitement du formulaire

Nous allons créer le fichier de traitement du formulaire d'ajout d'un client. Le principe : on récupère les valeurs transmises par le formulaire, on détermine l'idClient qui convient, et on exécute la commande d'insertion dans la base de données. Voici la démarche :

- copier-coller le code dans un fichier nommé traitement_ajout.py ;
- retourner sur la page WEB du formulaire, saisir les données pour un nouveau client, puis cliquer sur le bouton « Ajouter » ;
- vérifier que l'opération d'ajout à bien fonctionné .

```
#!/usr/bin/python3

import sqlite3 as sgbd
import cgi

# on récupère les valeurs du formulaire
formulaire = cgi.FieldStorage()

prenom = formulaire.getvalue('prenom')
nom = formulaire.getvalue('nom')
ville = formulaire.getvalue('ville')
naissance = formulaire.getvalue('naissance')
email = formulaire.getvalue('email')

# on détermine l'idClient du nouvel enregistrement
connexion = sgbd.connect('nomBase.db')
curseur = connexion.execute('SELECT MAX(identifiant) FROM clients')
for ligne in curseur.fetchall() :
    for valeur in ligne :
        maxim = valeur
idClient = maxim + 1

# on ajoute le nouveau client
client = [(idClient, prenom, nom, ville, naissance, email)]
connexion.executemany("INSERT INTO clients VALUES (?,?,?,?,?,?)", client)

# on affiche un message de réussite et la nouvelle table
curseur = connexion.execute("SELECT * FROM clients")
print("Content-type: text/html; charset=utf-8\n")
print("<html><body>")
print("<h1>Insertion réussie! Nouvelle table : </h1><br>")
print("<table border='1'>")
for ligne in curseur.fetchall() :
    print("<tr>")
    for valeur in ligne :
        print("<td>" + str(valeur) + "</td>")
    print("</tr>")
print("</table>")
```

```
print("</body></html>")

connexion.commit()

connexion.close()
```

Exercice 1.

Écrire un formulaire et son traitement qui permet de supprimer un enregistrement dans la base.

On pourra dans un premier temps réaliser un formulaire qui affiche la table complète, puis qui finalement demander l'idClient à supprimer, puis réaliser la page de traitement qui convient.
On pourra dans un second temps réaliser un formulaire avec un **SELECT**, qui permet directement à l'utilisateur de sélectionner le client à effacer de la base de données par un simple clic.

Exercice 2.

Écrire un formulaire et son traitement qui permet de modifier un enregistrement dans la base.

Il faut commencer par sélectionner l'idClient qui convient, retrouver ses attributs, les afficher dans un formulaire et faire le lien avec la page de traitement de la mise à jour. Le code de l'exercice précédent peut être utile!

Exercice 3.

Écrire une page index.html qui propose une liste de toutes les actions possibles sur la base de données, avec le lien , et ajouter en fin de chaque traitement un lien vers la page de menu

7 Approfondissement sur les bases de données : le modèle relationnel

7.1 Généralités

La notion de **modèle relationnel** est un concept qui a été défini par l'informaticien américain *Edgar F. Codd* vers 1970 alors qu'il travaillait chez *IBM*. D'une manière générale, un modèle de données est une représentation (mathématique ou informatique) de concepts que l'on souhaite étudier. Les intérêts d'une telle modélisation sont nombreux. Un modèle permet d'énoncer des **propriétés** de ces données en termes **logiques**. Il permet aussi de **programmer** des processus réels complexes sous forme d'opérations élémentaires sur ces données.

Définition du modèle relationnel

Dans le modèle relationnel, un objet modélisé (on parle d'entité) est représenté par un n-uplet de valeurs scalaires (on parle aussi d'attributs) et les collections d'objets par des ensembles de n-uplets.

Pour faire simple, ce modèle convient parfaitement quand les données que l'on souhaite traiter peuvent s'organiser sous forme de tables avec des descripteurs. On fera la distinction entre **modèle de données** et **stucture de données**. Ces deux concepts informatiques se placent à différents niveaux d'abstraction. Le modèle de données indique quelles caractéristiques d'une entité réelle on souhaite manipuler dans un programme, et les relations des entités entre elles. Une structure de données indique la manière dont on va organiser les données en machine (dans un langage de programmation). Ainsi pour un même modèle de données (par exemple les clients de la partie précédente) on peut choisir plusieurs structures de données différentes (un tableau, une liste chaînées, un arbre binaire, ...)

Exemple :
dans la partie précédente, un objet est un client représenté par un 6-uplet, par exemple le premier est :

(1, 'Pierre', 'Dupond', 'Paris', 1984, 'pierredupond@gmail.com')

Dans ce dernier, quatre composantes sont des chaînes de caractères et deux composantes sont des entiers. L'ensemble des clients peut alors être représenté par un ensemble **clients** de tels n-uplets :

clients = {
(1, 'Pierre', 'Dupond', 'Paris', 1984, 'pierredupond@gmail.com'),
(2 , 'Sabrina','Durand' , 'Nantes' , 2001 , 's.durand@free.fr') ,
(3 , 'Julien' , 'Martin' , 'Lyon' , 1996 ,'Julien69Martin@facebook.fr') ,
(4 , 'David' , 'Dubois' , 'Marseille' , 1989 , 'davidb13@orange.fr') ,
(5 , 'Marie' , 'Leroy' , 'Paris' , 1975 , 'marie-leroy@sfr.fr')
...
}

Définition d'une relation

On appelle un tel ensemble une **relation**.
Un élément d'une relation est appelé une **entité**. Il représente généralement un objet, une action, une personne, du monde réel.
Chaque entité possède des propriétés appelées **attributs**.

Chaque relation se conforme à un **schéma**. Ce dernier est une **description** qui indique pour chaque composante des n-uplet de la relation leur **nom** et leur **domaine**.
Par exemple, les éléments de la relation **clients** possèdent 6 **attributs** :

- **idClient** : l'identifiant unique, un entier ;

- **prenom** : le prénom, une chaîne de caractères ;
- **nom** : le nom, une chaîne de caractères ;
- **ville** : la ville, une chaîne de caractère ;
- **naissance** : l'année de naissance, un entier ;
- **email** : l'email, une chaîne de caractères.

Une manière plus compacte de noter un schéma pour une relation est la suivante :

clients(idClient INT, prenom STRING, nom STRING, ville STRING, naissance INT, email STRING)

Définition d'une base de données

Une **base de données** est un **ensemble de relations**.
Par extension, on appelle **schéma** d'une base de données l'ensemble des schémas des relations constituant la base.

D'autres modèles... Il existe d'autres modèles que la modèle relationnel. En effet, ce dernier est parfois trop « rigide » pour représenter certains types de données. Ainsi, pour représenter des documents ayant une structure complexe (chapitres, sections, sous-sections, titres, texte, mise en gras...) le modèle relationnels basé sur des ensembles de n-uplets est peu adapté. On pourra faire appel dans ce cas à des modèles dits *semi-structurés* comme le **XML**.
Un autre modèle de données est le **modèle de graphe**. Par exemple, si on souhaite représenter un réseau social et effectuer des traitements comme « trouver tous les amis qui ont les mêmes centres d'intérêts que moi » alors le modèle relationnel est peu adapté.
Faire cohabiter ces différents modèles dans un même système d'information permet d'effectuer des traitements riches, mais complexifie énormément la conception du système.

7.2 Construire pour une relation un schéma adapté : définir et respecter les contraintes

La modélisation des données se décompose en plusieurs étapes :

1. **déterminer les entités** (objets, actions, personnes, ...) que l'on souhaite manipuler ;
2. **modéliser les ensembles d'entités** comme des relations en donnant leur schéma, en s'attachant en particulier à choisir le bon domaine pour chaque attribut ;
3. définir les **contraintes** de la base de données, c'est à dire l'ensemble des **propriétés logiques** que nos données doivent vérifier à tout moment.

Définition de contraintes d'intégrité

La cohérence des données au sein d'une base de données est assurée par des **contraintes d'intégrité**. Ces dernières sont des invariants, c'est à dire des propriétés que les données doivent vérifier à tout instant.

On distingue parmi ces contraintes :

- Les **contraintes d'entités** qui garantissent que chaque entité d'une relation est unique.

  Pour ce faire, on définit une **clé primaire**, qui est un ensemble d'attributs qui identifie chaque entité de la relation de manière unique, et garantit la contrainte d'entité.

  On indique dans un schéma si un attribut en le soulignant.

  Un moyen souvent utilisé pour définir une clé primaire est l'indexation.

- Les **contraintes de domaine**, qui restreignent les valeurs d'un attribut à celles du domaine et évitent que l'on puisse donner à un attribut une valeur illégale.

  Les principales sont *String*, *Int*, *Boolean*, *Float*, *Date* et *Time*.

- Les **contraintes de référence** qui créent des associations entre deux relations. Elles permettent de garantir qu'une entité d'une relation $\mathscr{B}$ mentionne une entité existante dans une relation $\mathscr{A}$.

  Pour cela on définit une **clé étrangère**, qui est un ensemble d'attributs d'une table qui sont une **clé primaire** dans une autre table.

- Les **contraintes utilisateurs**, qui restreignent encore plus les valeurs d'un ou de plusieurs attributs et sont guidées par la nature des données que l'on souhaite stocker dans la base.

Ces contraintes doivent être utilisées pour assurer la qualité des données : elles permettent de s'assurer que les données sont « conformes » aux entités du monde réel qu'elles représentent.

Au moment de la création du schéma d'une base de données, il faut veiller aux **anomalies** que l'on peut rencontrer au départ. On distingue :

- des redondances dans les données ;
- des anomalies dans les insertions d'entités ;
- des anomalies dans la suppression d'entités ;
- des anomalies dans la mise à jour d'entités.

7.3 Un diagramme de classe pour définir un modèle conceptuel d'une base de données : un exemple

Pour illustrer concrètement les notions de cette partie et de la suivante, nous allons utiliser un exemple. On imagine qu'une personne souhaite élaborer une base de donnée de jeux vidéos. On appellera cette base de données **ludotheque**. Voici la relation proposée.

Nom du jeu	Auteur 1	Auteur 2	Éditeur	NbJoueursMin	NbJoueursMax	Nationalité éditeur	Durée en min	Illustrateur	Nationnalité illustrateur	Thèmes
Les chevaliers de la table ronde	Bruno Cathala	S. Laget	Days of wonder	3	7	Française	90	Julien Delval	Française	Moyen-âge, Légende arthurienne
Cargo Noir	Serge Laget		Days of wonder	2	5	Française	60	Miguel Coimbra	Française	Marché Noir, Navigation marchande
Era : medieval age	Matt Leacock		EggertSpiele	1	4	Allemande	50	Chris Quilliams	Canadienne	Médiéval, Construction
Smash up	Paul Peterson		Iello	2	4	Française	45	Bruno Balixa, Dave Alsop, Franciso Rico Torres	Américaine	Fantastique, Monstre, Pirate

► Analyser cette relation, les anomalies éventuelles, et proposer un schéma complet.

A. Représentation des classes

Sur un diagramme de classes, on représente chaque type de donnée de notre base sous la forme d'une classe, définie par un nom et par un ensemble d'attributs. Une première étape, dans la conception d'une base de données, consiste donc à recenser les types d'éléments que nous allons devoir manipuler et d'identifier leurs éléments, et leur domaine.

► Exercice 1 : déterminer les types de données du problème de la ludothèque, et leur domaine (on dit aussi type).

B. Représentation des associations et cardinalités

Mais il est également indispensable de faire apparaître sur ce schéma les associations (ou relations) entre entités de chacune des classes. Ainsi, dans le cas de notre problème, il y a une association entre auteur et jeu pour représenter le fait qu'un jeu est écrit par un ou plusieurs auteurs. Une relation se notre par un trait nommé entre 2 classes.

► Exercice 2 : complétez le diagramme de classe complet avec toutes les associations nécessaires

Sur le schéma obtenu dans l'exercice précédent, on constate que le fait qu'un jeu puisse avoir plusieurs auteurs ou illustrateur, n'apparaît pas. Il est en effet indispensable de rajouter, pour chaque association, les cardinalités la concernant. Cela consite à préciser, sur le schéma, pour chaque association entre 2 classes A et B, à combien d'instances de B peut être reliée chaque instance de A, et à combien d'instance de A peut être reliée chaque instance de B. Les cardinalités sont alors inscrites sur le schéma à l'extrémité de chaque association sous la forme d'un intervalle. Les cardinalités les plus classiques sont les suivantes :

- 1..* : au moins 1
- 0..* : un nombre quelconque (souvent noté *)
- 0..1 : au plus 1
- 1..1 : exactement 1 (souvent noté 1)

► Exercice 3 : complétez le diagramme de classe en rajoutant les cardinalités

C. Du diagramme de classe au modèle relationnel

Pour transformer un diagramme de classe en schéma de relation, il faut procéder en plusieurs étapes :

1. Créer un schéma de relation par classe ;
2. Pour chaque association N-M (c'est-à-dire dont les 2 cardinalités maximales sont "*"), créer un schéma de relation contenant les clés des classes participant à l'association et dont la clé consiste en l'union des clés des classes ;
3. Pour chaque association 1-N (c'est-à-dire dont 1 cardinalité maximale est "1" et l'autre est "*"), créer un schéma de relation contenant les clés des classes participant à l'association et dont la clé consiste en la clé de la classe "côté *" ;
4. Fusionner les schémas de relation de mêmes clés.

► Exercice 4 : transformez le diagramme de classe obtenu à l'exercice 3 en schéma de relation.

► Exercice 5 : donnez le contenu des tables établies dans l'exercice précédent permettant de représenter les données du problème initial

Pour en savoir plus :
`https://mermet.users.greyc.fr/Enseignement/EnseignementInformatiqueLycee/Havre/Bloc4/modele`
`html`

Voici les 5 tables identifiées pour respecter au mieux le modèle relationnel. Les descripteurs soulignés sont appelés **clés primaires**. Compléter ces tables avec les valeurs qui conviennent.

illustrateurs :

idIllustrateur	**nomIllustrateur**	**prenomIllustrateur**	**natIllustrateur**
1	Delval	Julien	Fr
2			
3			
4			
5			
6			

auteurs :

idAuteur	**nomAuteur**	**prenomAuteur**
1	Laget	Serge
2		
3		
4		

editeurs :

idEditeur	**nomEditeur**	**natEditeur**
1	Days of wonder	Fr
2		
3		

themes :

idTheme	**nomTheme**
1	Moyen-âge
2	
3	
4	
5	
6	
7	
8	
9	

jeux :

idJeu	**nomJeu**	**nbJoueursMin**	**nbJoueursMax**	**duree**	**idEditeur**
1	Les chevaliers de la table ronde	3	7	90	1
2					
3					
4					

Pour la table **jeux**, le dernier descripteur **idEditeur** fait référence à l'indexation de la table **editeurs**. Au moment de la création des tables en SQL, il faudra donc que la table **editeurs** soit crée avant la table **jeux**.

Voici les tables qui permettent de faire les associations N-M. Chaque couple est une association. Les valeurs de ces descripteurs font **référence** aux identifiants des tables précédemment définies. De plus, comme chaque association est sensée est unique, chaque couple devra-t-être déclaré comme **clé primaire**.
Compléter ces tables pour faire les bonnes associations.

aPourIllustrateur :

idJeu	**idIllustrateur**
1	1

aPourAuteur :

idJeu	**idAuteur**
1	1

aPourTheme :

idJeu	**idTheme**
1	1

7.4 Traduction en SQL du modèle relationnel

Dans la partie 4 de ce cours, nous avons vu pas à pas comment créer une table en SQL. Dans cette partie déjà nous avons évoqué l'utilité d'une clé primaire par indexation, et le respect du domaine des valeurs dans la table. Cependant comme la relation étudiée ne contenait qu'une seule table, la table **clients**, nous n'avons pas évoqué les contraintes de références sur plusieurs table, ni les contraintes utilisateurs.

Voici la forme générale de la définition d'une classe avec ses descripteurs, ses contraintes de domaines, et ses autres contraintes.

```
CREATE TABLE nomTable (
premierAttribut DOMAINE CONTRAINTES ,
secondAttribut DOMAINE CONTRAINTES ,
... ,
AUTRES CONTRAINTES ) ;
```

Les clauses SQL pour traduire les contraintes du modèle relationnel

1. les contraintes de domaines, qui permettent de préciser de quel type sont les valeurs d'un descripteur, sont traduites par **VARCAHR**(n), **TEXT**, **INT**, **BOOLEAN**, **BIGINT**, **DECIMAL**(t,f), **DATE**, **DATE**, **TIME**. Il y en a d'autres, voir la documentation pour en savoir plus ;
2. la clause **PRIMARY KEY** permet de spécifier qu'un attribut est une clé primaire. Si la clé primaire est composée de plusieurs descripteurs, on rajoute après la liste des descripteurs la phrase PRIMARY KEY (descripteur1, descripteur2, ...) ;
3. la clause **NOT NULL** permet de spécifier que les valeurs d'un attribut doivent être renseignées. Par défaut une clé primaire est NOT NULL ;
4. on peut spécifier que les valeurs d'un attribut soient uniques, sans pour autant que cet attribut fasse partie de la clé primaire.Pour cela on utilise la clause **UNIQUE** ;
5. un attribut peut être qualifié de clé étrangère, c'est à dire qu'il fait référence à l'attribut d'une autre table, avec la clause **REFERENCES**. Un exemple de syntaxe : nomDescripteur1 INT REFERENCES nomTable(nomDescripteur2) .
6. les contraintes utilisateurs sont elles aussi ajoutées après la liste des descripteurs. Ils utilisent la clause **CHECK**. Par exemple si on veut que l'attribut **naissance** ne dépasse pas 2020, on écrira la phrase CHECH (naissance <= 2020) .

Maintenant nous allons utiliser ces clauses pour traduire la ludothèque et ses tables en SQL.

Nous allons mettre en application ces notions. Le fichier suivant contient le code **Python** qui permet de créer la base de donnée **ludotheque**.

1. Copier et coller ce code dans un fichier appelé **ludotheque_creation.py**.
2. Etudier la structure des tables en SQL.
3. Compléter les tuples avec les bonnes valeurs de la base de données **ludotheque**.
4. Exécuter le code, la base de données **ludotheque** est crée sous forme d'un fichier portant ce nom dans le même dossier.

```
# un script pour créer la base de données ludotheque
import sqlite3 as sgdb

connexion = sgdb.connect('ludotheque.db')
curseur = connexion.cursor()
curseur.execute('PRAGMA foreign_keys = ON')

# avant la création, on supprime toutes les tables existantes
# on supprime dans l'ordre inverse de création
curseur.execute('DROP TABLE IF EXISTS aPourTheme')
...

# création de la table : illustrateurs
curseur.execute('''CREATE TABLE illustrateurs (
    idIllustrateur INT PRIMARY KEY NOT NULL,
    nomIllustrateur TEXT,
    prenomIllustrateur TEXT,
    natIllustrateur TEXT )  ''')

illustrateurs = [
(1, 'Delval', 'Julien', 'Fr') ,
... ]
curseur.executemany("INSERT INTO illustrateurs values(?,?,?,?)",illustrateurs)

# création de la table : auteurs
curseur.execute('''CREATE TABLE auteurs (
     idAuteur                INT PRIMARY KEY NOT NULL,
     nomAuteur                TEXT NOT NULL,
     prenomAuteur         TEXT )''')

auteurs = [        (1, 'LAGET', 'Serge'),
                ... ]
curseur.executemany("INSERT INTO auteurs VALUES(?,?,?)", auteurs)

# création de la table : editeurs
curseur.execute('''CREATE TABLE editeurs (
     idEditeur INT PRIMARY KEY NOT NULL,
     nomEditeur TEXT NOT NULL,
```

```
     natEditeur TEXT)''')

editeurs = [(1, 'Days of wonder', 'Fr'),
            ... ]
curseur.executemany("INSERT INTO editeurs values(?,?,?)", editeurs)

# création de la table : jeux
curseur.execute('''CREATE TABLE jeux (
     idJeu INT PRIMARY KEY NOT NULL,
     nomJeu TEXT NOT NULL,
     nbJoueursMin INT,
     nbJoueursMax INT,
     duree INT,
     idEditeur INT REFERENCES editeurs(idEditeur))''')

jeux = [
    (1, 'Les chevaliers de la table ronde', 3, 7, 90, 1),
    ...  ]
curseur.executemany("INSERT INTO jeux values(?,?,?,?,?,?)", jeux)


# création de la table : themes
curseur.execute('''CREATE TABLE themes (
     idTheme          INT PRIMARY KEY NOT NULL,
     nomTheme    TEXT NOT NULL) ''')

themes = [        (1, 'Moyen-âge'),
                ...  ]
curseur.executemany("INSERT INTO themes VALUES(?,?)", themes)

# création de la table : aPourIllustrateur
curseur.execute('''CREATE TABLE aPourIllustrateur (
     idJeu INT NOT NULL,
     idIllustrateur INT NOT NULL,
     PRIMARY KEY (idJeu, idIllustrateur),
     FOREIGN KEY(idJeu) REFERENCES jeux(idJeu),
     FOREIGN KEY(idIllustrateur) REFERENCES illustrateurs(idIllustrateur))''')

aPourIllustrateur = [(1, 1),
                     ... ]
curseur.executemany("INSERT INTO aPourIllustrateur values(?,?)", aPourIllustrateur)

# création de la table : aPourAuteur
curseur.execute('''CREATE TABLE aPourAuteur (
    idJeu INT NOT NULL ,
    idAuteur  INT NOT NULL ,
    PRIMARY KEY ( idJeu, idAuteur ) ,
    FOREIGN KEY (idJeu) REFERENCES jeux(idJeu) ,
    FOREIGN KEY (idAuteur) REFERENCES auteurs(idAuteur) ) ''' )
```

```
aPourAuteur =        [ (1, 1) ,
                       ... ]
curseur.executemany("INSERT INTO aPourAuteur VALUES (?,?)",aPourAuteur)

# création de la table : aPourTheme
curseur.execute('''CREATE TABLE aPourTheme (
    idJeu  INT NOT NULL ,
    idTheme INT NOT NULL ,
    PRIMARY KEY ( idJeu, idTheme ) ,
    FOREIGN KEY (idJeu) REFERENCES jeux(idJeu) ,
    FOREIGN KEY (idTheme) REFERENCES themes(idTheme) ) ''' )

aPourTheme = [ (1, 1) ,
               ... ]

curseur.executemany("INSERT INTO aPourTheme VALUES (?,?)",aPourTheme)

connexion.commit()
connexion.close()
```

Vous devez ensuite vérifier que toutes les pages s'affichent correctement. Pour cela, vous pourrez utiliser directement **SQLite** en ligne de commande, ou utiliser le code suivant, que vous pourrez enregistrer dans le fichier **ludotheque_lecture.py**.

```
# un script python pour afficher le résultat d'un SELECT sur la  ludotheque
import sqlite3 as sgdb

connexion = sgdb.connect('ludotheque.db')
curseur = connexion.cursor()
connexion.execute('PRAGMA foreign_keys = ON')

requete = """
SELECT *
FROM illustrateurs """
curseur = connexion.execute(requete)

#Affichage des champs
colonnes = [description[0] for description in curseur.description]
print(colonnes)

#Exécution de la requête et affichage du résultat
for tuple in curseur.fetchall() :
    print(tuple)

connexion.commit()
connexion.close()
```

Enfin, avant de passer à la section suivante, on se pose deux questions :

1. on souhaite écrire une requête qui affiche le **nom de chaque jeu** et **son éditeur** ;
2. on souhaite, pour un jeu, afficher la liste de ses illustrateurs.

Comment faire ?

Extraits du programme :

- *Modèle relationnel : relation, attribut, domaine, clef primaire, clef étrangère, schéma relationnel. Identifier les concepts définissant le modèle relationnel.*
- *Base de données relationnelle : Repérer des anomalies dans le schéma d'une base de données. La structure est un ensemble de schémas relationnels qui respecte les contraintes du modèle relationnel. Les anomalies peuvent être des redondances de données ou des anomalies d'insertion, de suppression, de mise à jour. On privilégie la manipulation de données nombreuses et réalistes.*

Exercice 1.

On souhaite modéliser un **annuaire téléphonique** simple dans lequel chaque personne (identifiée par son nom et son prénom) est associée à un numéro de téléphone.

1. Proposer, sous forme de schéma, une modélisation relationnelle de cet annuaire.
2. Traduire la création de cette relation en SQL.

Exercice 2.

On reprend la solution proposée pour l'exercice précédent. Dire si chacun des ensembles est une relation valide pour le schéma **Annuaire**.

1. {}
2. $\{('Titi','Toto','0123456789')\}$
3. $\{('Titi','Toto','0123456789'),('John','Doe','0123456789'\}$
4. $\{('Titi','Toto','0123456789'),('John','Doe')\}$
5. $\{('Titi','Toto',42)\}$

Exercice 3.

Un livre est défini par un titre, un résumé un prix et son éditeur. Quant à l'éditeur, il est défini par son nom et son adresse.

1. Proposer, sous forme de schéma, une modélisation relationnelle pour stocker ces informations.
2. Traduire la création de cette relation en SQL.

Exercice 4.

Donner une modélisation relationnelle d'un **bulletin scolaire**. Cette dernière doit permettre de mentionner :

- des élèves, possédant un identifiant numérique unique ;
- un ensemble de matières fixées, mais qui ne sont pas données, elles aussi indexées ;
- au plus une note sur 20, par matière et par élève.

Exercice 5.

1. On souhaite représenter la structure (simplifiée) des **collectivités** territoriales de France métropolitaine. Nous nous limiterons aux régions (nom), départements (nom, numéro) et villes (nom, code postal principal, nombre d'habitants). Bien sûr, la base de données doit permettre de retrouver que Le Havre est en Seine-Maritime, dans la région Normandie. Proposez un diagramme de classe correspondant au modèle conceptuel de la base de données permettant de stocker ces informations.
2. On souhaite maintenant associer à chaque département sa préfecture. Modifiez votre schéma conceptuel pour faire figurer cette information.
3. Traduire la création de cette relation en SQL.

8 Exploiter des données réparties sur plusieurs tables : les jointures

Dans la partie précédente, nous avons vu que l'organisation des données sur une seule table n'est pas cohérente, ou pas satisfaisante, pour des raisons de redondance, de facilité de mise à jour, ou de capacité d'évolution. Lorsque des données sont organisées sur plusieurs tables, le principe pour extraire des informations qui sont réparties sur plusieurs tables est la **jointure** : JOIN en SQL. Pour illustrer ce principe, nous allons prendre comme exemple la **ludothèque** étudiée et programmée à la partie précédente.

8.1 Le principe des jointures

La clause JOIN

Étant données deux tables A et B, la **jointure** de la table A et B consiste à **créer toutes les combinaisons** de lignes de A et de B. On appelle en mathématiques cette opération le « produit cartésien ».

Exemple :
avec la requête `SELECT * FROM jeux` on obtient le contenu de la table avec la relation :

```
(1, 'Les chevaliers de la table ronde', 3, 7, 90, 1)
(2, 'Cargo Noir', 2, 5, 60, 1)
(3, 'Era: medieval age', 1, 4, 50, 2)
(4, 'Smash up', 2, 4, 45, 3)
```

avec la requête `SELECT * FROM editeurs` on obtient le contenu de la table avec la relation :

```
(1, 'Days of wonder', 'Fr')
(2, 'EggertSpiele', 'All')
(3, 'Iello', 'Fr')
```

avec la requête `SELECT * FROM jeux JOIN editeurs` on obtient le contenu de la table avec la relation :

```
(1, 'Les chevaliers de la table ronde', 3, 7, 90, 1, 1, 'Days of wonder', 'Fr')
(1, 'Les chevaliers de la table ronde', 3, 7, 90, 1, 2, 'EggertSpiele', 'All')
(1, 'Les chevaliers de la table ronde', 3, 7, 90, 1, 3, 'Iello', 'Fr')
(2, 'Cargo Noir', 2, 5, 60, 1, 1, 'Days of wonder', 'Fr')
(2, 'Cargo Noir', 2, 5, 60, 1, 2, 'EggertSpiele', 'All')
(2, 'Cargo Noir', 2, 5, 60, 1, 3, 'Iello', 'Fr')
(3, 'Era: medieval age', 1, 4, 50, 2, 1, 'Days of wonder', 'Fr')
(3, 'Era: medieval age', 1, 4, 50, 2, 2, 'EggertSpiele', 'All')
(3, 'Era: medieval age', 1, 4, 50, 2, 3, 'Iello', 'Fr')
(4, 'Smash up', 2, 4, 45, 3, 1, 'Days of wonder', 'Fr')
(4, 'Smash up', 2, 4, 45, 3, 2, 'EggertSpiele', 'All')
(4, 'Smash up', 2, 4, 45, 3, 3, 'Iello', 'Fr')
```

On voit bien que la nouvelle relation proposée est l'ensemble des combinaisons formée les 6-uplets de la table **jeux** et des triplets de la table **editeurs**. C'est le principe d'une jointure : la réponse donnée est un produit cartésien des tables précisées, c'est-à-dire que l'on obtient tous les tuples qu'il est possible de faire en mettant pour les premières composantes les tuples de la première table et pour les dernières composantes

tous les tuples de la deuxième table. Ceci n'est pas très intéressant! En effet il serait préférable dans cette nouvelle de **garder que les tuples qui ont la même valeur pour le champ idEditeur de la table jeux et le champ idEditeur de la table editeurs**.

La clause ON

La clause **ON** permet de spécifier quels sont les tuples qui seront conservés dans le produit cartésien. En général il faut spécifier quels champs de la première table doivent être égaux à ceux de la seconde.

Exemple :
avec la requête `SELECT * FROM jeux JOIN editeurs ON jeux.idEditeur = editeurs.idEditeur` on retient que les couples pour lesquels le champ **idEditeur** de la table **jeux** et égale à la valeur du champ **idEditeur** de la table **editeurs**. Le résultat obtenu est donné ci-dessous, et cette relation est intéressante car elle permet de voir pour chaque jeu le nom de son éditeur.
On remarque que la séparation entre le nom de la table et le nom du champ dans une requête se fait avec un **point**, par exemple `jeux.nomJeu` désigne le champ **nomJeu** de la table **jeux**.

```
(1, 'Les chevaliers de la table ronde', 3, 7, 90, 1, 1, 'Days of wonder', 'Fr')
(2, 'Cargo Noir', 2, 5, 60, 1, 1, 'Days of wonder', 'Fr')
(3, 'Era: medieval age', 1, 4, 50, 2, 2, 'EggertSpiele', 'All')
(4, 'Smash up', 2, 4, 45, 3, 3, 'Iello', 'Fr')
```

L'écriture des requêtes avec des jointures peut être fastidieuse quand les noms des tables est un peu long. Pour raccourcir les requêtes, on peut utiliser la clause **AS**. Aucun exemple ne sera donné sur cette clause.

La clause AS

La clause **AS** permet de **renommer ponctuellement** le nom d'une table pour faciliter l'écriture et la compréhension des jointures.

Il est possible d'effectuer des **sélections (WHERE)** et des **projections (ne garder que certains champs)** sur le résultat d'une jointure. De plus, si les champs sur lesquels faire la jointure portent le même nom dans les 2 tables, cette requête peut aussi être écrite en utilisant la clause **USING**.

La clause USING

La clause **USING** permet au moment de la jointure, quand les champs censés être égaux portent le même nom, de spécifier ce nom de champ pour faire la jointure.

Exemple :
Par exemple si nous voulons afficher la relation donnant pour chaque jeu le nom de son éditeur, alors on saisit la requête :

```
SELECT jeux.nomJeu,  editeurs.nomEditeur
FROM jeux
JOIN editeurs USING(idEditeur)
```

Le résultat affiché est alors :

```
('Les chevaliers de la table ronde', 'Days of wonder')
('Cargo Noir', 'Days of wonder')
('Era: medieval age', 'EggertSpiele')
('Smash up', 'Iello')
```

On pourra retenir le modèle général de l'utilisation d'une jointure :

SELECT table_i.champ_j, ...
FROM table_1
JOIN table_2 ON table_1.champ_j = table_2.champ_j
JOIN table_3 ON table_2.champ_j = table_3.champ_j
WHERE conditions sur certains champs

Pour en savoir plus sur cette partie du cours, voir
`https://mermet.users.greyc.fr/Enseignement/EnseignementInformatiqueLycee/Havre/Bloc4/sql.html`

8.2 D'autres exemples de jointures

Toujours avec la relation **ludotheque**, écrire en utilisant les jointures une requête qui répond à chaque situation, puis vérifier cette requête avec Python.

1. Écrire la requête permettant d'associer à chaque nom de jeu son éditeur, la nationalité de son éditeur, avec un tri par nationalité, puis nom de jeu.
2. Écrire la requête permettant d'afficher tous les noms de jeu édités par un éditeur français.
3. Écrire la requête permettant d'afficher le nom du jeu et ses auteurs pour le jeu 1.
4. Écrire la requête permettant d'afficher le nom du jeu et ses illustrateurs et leur nationalité pour le jeu 4.
5. Écrire la requête permettant d'afficher le nom du jeu et ses thèmes pour le jeu 3.
6. Écrire une requête qui permet, pour un thème donné (par exemple le 6) de retrouver tous les jeux qui correspondent.

Extraits du programme : construire des requêtes d'interrogation à l'aide des clauses du langage SQL : JOIN. On peut utiliser DISTINCT, ORDER BY ou les fonctions d'agrégation sans utiliser les clauses GROUP BY et HAVING.

Exercices...