TDLOG séance 8 - SQL et NoSQL

Xavier Clerc – xavier.clerc@enpc.fr

1er décembre 2025

Table des matières

Ce document propose quelques compléments à celui sur les bases de donnés et l'algèbre relationnelles. Les sections 1 à 3 montrent comment accéder à une base de données depuis un programme. Après une présentation générale à la section 1, on présente les grandes lignes de la connexion via une API (section 2) ou un ORM (section 3).

Les sections 4 et 5 montrent comment procéder pour modéliser une base de données, et comment s'assurer que son schéma évite des redondances en respectant des formes normales.

Enfin, la section 6 donne un aperçu des bases de données NoSQL. On s'intéresse aux grandes familles de NoSQL, ainsi qu'aux différences avec SQL, et les compromis à l'œuvre lorsque l'on doit arbitrer entre SQL et NoSQL.

1 Utilisation de SQL depuis un langage de programmation

Il est très fréquent qu'une base de données soit consultée et modifiée depuis un programme écrit dans un langage de programmation autre que SQL. Dans ce cadre, il faut que ce langage puisse indiquer les instructions et requêtes qu'il souhaite voir exécutées par le SGBD.

On distingue classiquement deux dimensions pour qualifier la façon dont le langage de programmation accède au SGBD :
- la manière dont l'instruction SQL est construite ;
- la manière dont l'instruction est soumise au SGBD.

1.1 Interfaces statiques et dynamiques

L'interface est dite statique (*static SQL*) si la portion de code SQL est figée lors de la création du programme. En revanche, elle est dite dynamique (*dynamic SQL*) si elle est construite lors de l'exécution du programme.

Il faut être vigilant quant au fait que *statique* n'est ici pas exactement équivalent à *constant*. En effet, la plupart des requêtes que l'on souhaite exécuter n'ont d'intérêt que si on peut en modifier des paramètres. Ainsi, les requêtes *statiques* disposent-elles de moyens permettant au programme de placer à des endroits prédéterminés (typiquement `?` ou `:nom`) les valeurs qu'il souhaite. On trouvera ainsi des requêtes statiques de la forme :

```
SELECT * FROM Table WHERE attr = ?
```

ou encore :

```
SELECT * FROM Table WHERE attr = :valeur
```

Par opposition, une interface est dynamique si la requête est construite à l'exécution, par exemple tel que montré par le code Code 1.1.

Code 1.1 – Appel dynamique depuis Python.

```
def appel_bd(k = None, v = None):
  sql = "SELECT * FROM Table"
  if k is not None:
    sql = sql + f" WHERE {k} = {str(v)}"
  ...

appel_bd()
appel_bd("X", 1)
```

1.2 Modes intégrés et appelés

Le mode d'utilisation est dit intégré (*embedded SQL*) si la portion de code SQL à écrire est insérée au milieu du code du programme comme s'il s'agissait d'une construction du langage de programmation. En revanche, il est dit appelé (*called SQL*) si elle est passée au SQL au travers d'une API.

Dans le premier cas, on peut imaginer écrire au milieu d'un script Python quelque chose de la forme :

```
for ligne in SELECT * FROM Table:
  print(ligne[0], ligne[2])
```

Dans le second cas, on peut imaginer écrire au milieu d'un script Python quelque chose de la forme :

```
for ligne in executer('SELECT * FROM Table'):
  print(ligne[0], ligne[2])
```

1.3 Notion d'ORM

Avec ce qui précède, on voit qu'il existe diverses possibilités d'interfacer un langage de programmation avec un SGBD. Cependant, on voit aussi que si l'on veut représenter les éléments d'une base de données sous forme d'objets dans le programme, il va falloir écrire des requêtes SQL :

- pour lire les données depuis la base et les créer les objets correspondants ;
- pour écrire les données dans la base lorsque les objets correspondants sont modifiés.

Or ce travail de conversion n'est pas seulement répétitif, inintéressant et sujet à erreur, il est de plus compliqué par un problème d'impédance (*impedance mismatch*). En informatique, on parle de problème d'impédance lorsque l'on souhaite *brancher* deux technologies qui manient des concepts différents.

Ici, le problème d'impédance tient au fait que le modèle relationnel et le modèle objet ne sont pas tout à fait alignés. Typiquement :

- une case d'une base de données relationnelle ne peut pas contenir l'équivalent d'une ligne, cela signifie que le concept de donnée membre (ou attribut) en programmation objet va devoir être traduit par des mécanismes relationnels fondés sur les clefs ;
- un objet n'est pas fondé sur la notion de clef primaire ;

— la notion d'héritage n'existe pas dans le modèle relationnel ;
— *etc*

Pour toutes ces raisons, il est fréquent d'avoir recours à des logiciels qui permettent de faire automatiquement la correspondance entre les modèles objet et relationnel. Ces logiciels sont appelés ORM (pour object relational mappers) et génèrent automatiquement les fonctions de traduction entre les deux modèles. En outre, ils apportent l'énorme avantage de ne pas nécessiter de modification explicite lorsque le logiciel évolue : les ORM mettent à jour leurs fonctions de traduction au fil des changements.

2 Utilisation via l'API `sqlite3`

Par rapport à la typologie dressée dans la section précédente, l'utilisation de `sqlite3` correspond à un usage dynamique ou statique et appelé. Le module `sqlite3` est l'une des nombreuses API disponibles en Python permettant de soumettre des requêtes SQL à un SGBD. Nous avons choisi ce module pour les raisons suivantes :

— il est simple mais permet de présenter la plupart des notions utilisées dans de telles API ;
— il est livré en standard avec Python ;
— il s'interface avec SQLite, qui est un SGBD disponible sous la forme d'une simple librairie (par opposition à un programme s'exécutant en mode client-serveur), ce qui facilite grandement le déploiement.

La documentation complète du module `sqlite3` est disponible à l'adresse http://docs.python.org/library/sqlite3.html.

2.1 Notion de connexion

La notion première des API d'accès à une base de données est celle de connexion. Cela tient notamment au fait que dans la plupart des cas on se trouve dans un mode client-serveur avec une connexion réseau vers une machine distante sur laquelle s'exécute le SGBD. Bien que `sqlite3` ne se fonde pas sur une architecture client-serveur, il conserve néanmoins la notion de connexion.

La connexion sert à initier les échanges avec la base de données. On indique typiquement à quelle base de données on souhaite accéder et on donne éventuellement ses identifiant et mot de passe pour la gestion des droits par le SGBD.

En `sqlite3`, il suffit d'indiquer le nom du fichier contenant la base de données (ou d'utiliser le nom spécial `:memory:` pour stocker la base de données en mémoire vive). L'obtention d'une connexion se fait en appelant la fonction `connect` avec comme paramètre le nom du fichier.

Une fois que l'on dispose d'un objet `Connection` renvoyé par la fonction précédente, on peut exécuter des instructions SQL par la méthode `execute` et gérer des transactions grâce aux méthodes :

— `commit()` ;
— `rollback()`.

Ainsi pour créer une table `T` qui sera stockée dans le fichier `/path/to/db`, on écrira la suite d'instructions montrée par le code Code 2.1.

Code 2.1 – Création d'une table depuis Python.

```
import sqlite3

connexion = sqlite3.connect('/path/to/db')
connexion.execute('CREATE TABLE T(X INT, Y VARCHAR(10))')
connexion.commit()
connexion.close()
```

Au sein des instructions SQL passées à la fonction `execute`, il est fréquent de vouloir faire figurer des valeurs issues du programme Python. Pour cela, on peut créer la chaîne représentant la commande SQL par concaténations successives ou en utilisant `format`. Il est également possible d'utiliser le caractère `?` dans la chaîne SQL pour indiquer des "trous" qui seront remplis par des valeurs passées à l'aide d'un tuple. Ainsi, le code Code 2.2 montre-t-il comment définir une fonction Python permettant d'ajouter un élément à la table précédemment créée.

Code 2.2 – Ajout de ligne depuis Python.

```
def ajoute(conn, x, y):
  conn.execute('INSERT INTO T VALUES (?, ?)', (x, y))
```

2.2 Notion de curseur

Lorsque l'on souhaite soumettre une requête renvoyant des résultats, il est d'usage d'utiliser un *curseur* qui permettra de parcourir les lignes renvoyées. Avec `sqlite3`, on obtient un curseur depuis une connexion en appelant la méthode `cursor()` sur cette dernière. L'objet `Cursor` alors renvoyé peut être utilisé pour exécuter des requêtes via la méthode `execute`.

Le code Code 2.3 montre comment utiliser un curseur pour exécuter une requête puis récupérer sous forme de liste de tuples l'ensemble des lignes correspondant au résultat de la requête. Ces lignes sont ensuite parcourues et affichées. La méthode `fetchall()` renvoie la liste de toutes les lignes ; il est également possible de consulter le résultat ligne par ligne en appelant successivement `fetchone()` jusqu'à obtenir une valeur `None`.

Code 2.3 – Requête SQL depuis Python.

```
def affiche_liste(conn):
  curseur = conn.cursor()
  curseur.execute('SELECT * FROM T')
  lignes = curseur.fetchall()
  for l in lignes:
    print(l[0], '->', l[1])
  curseur.close()
```

2.3 Notion de ligne

La notion de ligne (ou *row*) permet d'accéder aux éléments d'une ligne non plus par leur indice mais par leur nom dans la base SQL. Pour pouvoir utiliser cette possibilité, il faut commencer par configurer la connexion à l'aide de l'affectation suivante : `connexion.row_factory = sqlite3.Row`. Dès lors, les éléments renvoyés par les méthodes `fetchone()` et `fetchall()` sont des instances de `sqlite3.Row` qui se comportent à la fois comme des tuples (indices numériques) et des dictionnaires (indices selon les noms des colonnes). On peut alors écrire une nouvelle version de la fonction `liste` comme le montre le code Code 2.4.

Code 2.4 – Requête SQL depuis Python.

```
def affiche_liste(conn):
  curseur = conn.cursor()
  curseur.execute('SELECT X, Y as chaine FROM T')
  lignes = curseur.fetchall()
  for l in lignes:
    print(l['X'], '->', l['chaine'])
  curseur.close()
```

3 Utilisation via l'ORM `storm`

Nous avons présenté la notion d'ORM au paragraphe 1.3 et allons ici l'illustrer en Python à l'aide du logiciel STORM [1]. D'autres ORM ou interfaces d'accès à des SGBD depuis Python sont listées à l'adresse http://wiki.python.org/moin/HigherLevelDatabaseProgramming. En particulier, SQLAlchemy est une alternative populaire, notamment parce qu'elle est supportée comme *plugin* de la librairie Flask utilisée pour le développement de sites web.

Comme nous l'avons dit, un ORM se charge de générer le code qui permet de convertir les données entre leur représentation objet dans l'application et leur représentation relationnelle dans le SGBD. `storm`, comme la plupart des ORM permet de s'interfacer de manière transparente avec divers SGBD (avec une préférence pour les implémentations libres). Ainsi, par rapport aux implémentations libres citées plus haut, il est possible d'utiliser `storm` avec SQLite, MySQL ou PostgreSQL.

1. Disponible à l'adresse https://storm.canonical.com/ (ou https://github.com/cogini/storm/)

3.1 Notion de *store*

Les ORM reposent sur la notion de *store* (parfois traduit par entrepôt) qui permet de définir ce qui est effectivement stocké dans la base de données. En effet, seule une partie des objets créés et manipulés dans le programme est sauvegardée dans la base de données (on parle de données *persistentes*). Cela découle du fait que de nombreux objets ne sont créés que pour l'exécution courante de l'application et que l'on n'a aucun intérêt à sauvegarder leur état (*p. ex.* résultats intermédiaires, interface graphique, *etc*). Pire, stocker tous les objets dans la base serait une grossière erreur en termes de performances : l'ORM rend la sauvegarde transparente mais elle n'en reste pas moins coûteuse. Il faut donc sélectionner les instances que l'on souhaite sauvegarder. Cela se fait en les enregistrant dans un *store*.

Le code Code 3.1 montre comment obtenir le *store* lié à une base de données. Dans cet exemple, nous utilisons le SGBD SQLite. Cela se voit au paramètre passé pour obtenir la connexion : le début de la chaîne de caractères (avant le caractère "`:`") indique le SGBD auquel se connecter.

Code 3.1 – Connexion à une base de données et récupération du *store*.

```
1  import storm.locals
2
3  connexion = storm.locals.create_database('sqlite:/path/to/db')
4  store = storm.locals.Store(connexion)
```

Toutes les opérations qui sont faites sur le *store* se font dans des transactions. Il faut donc appeler les méthodes `commit` et `rollback` de la classe `Store` pour respectivement enregistrer et annuler les modifications effectuées. Le début d'une transaction n'est pas marqué explicitement, il s'agit soit de la création du *store*, soit du dernier appel à `commit` ou `rollback`.

3.2 Définition de classes et création d'instances

La définition d'une classe se fait de manière habituelle, en enrichissant la déclaration d'éléments relatifs à la base de données. On indique typiquement le nom de la table à utiliser, ainsi que les noms et types des colonnes. De plus, on indique quelle(s) colonne(s) compose(nt) la clef primaire s'il y en a une. Le code Code 3.2 montre comment définir une classe `Film`, reprenant l'exemple de la séance précédente.

Code 3.2 – Définition d'une classe et de la table associée.

```
1  import storm.locals
2
3  class Film:
4    __storm_table__ = 'Films'
5    ident = storm.locals.Int(primary = True)
6    annee = storm.locals.Int()
7    metteurEnScene = storm.locals.Unicode()
8    def __init__(self, ident, annee, metteurEnScene):
9      self.ident = ident
10     self.annee = annee
11     self.metteurEnScene = metteurEnScene
```

Il est alors possible de manipuler des instances de `Film` comme toutes les autres instances d'objets Python, et de les ajouter au *store* à l'aide de la méthode *add*. Pour savoir si une instance est sauvegardée dans un *store*, il suffit de comparer le résultat de `storm.locals.Store.of(instance)` à `None`. Le code Code 3.3 montre comment ajouter des instances au *store*. Les modifications effectuées sur les objets sont répercutées dans la base de données lors des appels à `commit`. À l'inverse, les appels à `rollback` défont les modifications effectuées sur les instances.

Code 3.3 – Utilisation du *store*.

```
le_parrain = Film(04143, 1972, 'Francis Ford Coppola')
scarface = Film(01345, 1985, 'Brian de Palma')

...

store.add(le_parrain)
store.add(scarface)
store.commit()

...

scarface.annee = 1983
store.commit()

...

le_parrain.annee = 1952
store.rollback()
```

3.3 Gestion des références

En Python, une instance peut être liée à une autre en conservant une référence. Dans une base de données, le mécanisme semblable est l'utilisation d'un système de clef primaire et de clef étrangère. Le code Code 3.4 montre comment définir une classe `Titre`, faisant référence à `Film`.

Code 3.4 – Références entre instances et entre tuples.

```
class Titre:
  __storm_table__ = 'Titres'
  ident = storm.locals.Int()
  langue = storm.locals.Unicode()
  titre = storm.locals.Unicode()
  film = storm.locals.Reference(ident, Film.ident)
  def __init__(self, ident, langue, titre):
    self.ident = ident
    self.langue = langue
    self.titre = titre
```

3.4 Requêtes

Comme nous l'avons vu, la gestion des instructions telles que `INSERT` ou `UPDATE` est automatique via le *store*. Si nous utilisons une base de données, c'est à la fois pour sauvegarder les données mais aussi afin de pouvoir effectuer facilement des recherches. Ces recherches s'effectuent par le biais du *store*.

La recherche la plus basique consiste à retrouver une instance donnée à partir de sa clef primaire. Cela se fait à l'aide de la méthode `get` d'un *store* qui prend deux paramètres :

— la classe de l'instance que l'on recherche ;
— la valeur de la clef primaire.

Ainsi, on pourra écrire `store.get(Film, 4143)` pour récupérer une référence sur l'instance correspondant au film "Le parrain".

Plus généralement, les requêtes de type `SELECT` se font quant à elles via la méthode `find` d'un *store* qui prend deux paramètres :

— la classe de l'instance que l'on recherche ;
— une expression représentant la condition de recherche.

Ainsi, on pourra écrire `store.find(Film, Film.ident == 4143)` pour récupérer la liste des films ayant pour identifiant `4143`. Pour trouver le titre en anglais du film, il suffit par exemple d'utiliser une jointure :

```
liste = store.find(Titre,
                   (Film.ident == 04143)
                   & (Film.ident == Titre.ident)
                   & (Titre.langue == 'EN'))
titre_anglais = liste[0].titre
```

4 Dépendance fonctionnelle et formes normales

Les notions exposées dans cette section ont pour objet d'améliorer la qualité d'une base de données en évitant les redondances et en s'assurant que les clefs primaires sont choisies avec soin. Il s'agit en conséquence d'assurer que le schéma de la base de données répond à des critères qui garantissent sa maintenabilité et sa cohérence sur le long terme.

4.1 Notion de dépendance fonctionnelle

La notion de dépendance fonctionnelle a été définie par Codd pour l'algèbre relationnelle. Soit une relation $R(a_1, \ldots, a_n)$, on dit qu'il existe une dépendance fonctionnelle de a_i vers a_j (notée $a_i \rightarrow a_j$) si à tout instant deux lignes de R qui ont même valeur pour a_i ont aussi même valeur pour a_j. Cette définition s'étend naturellement au cas où a_i et a_j ne désignent pas une unique colonne mais un ensemble de colonnes.

Dans la définition, il faut prêter une attention toute particulière aux mots "*à tout instant*". Il est impératif ne pas considérer que l'état actuel de la base de données, mais bien d'imaginer toutes les combinaisons de données que la base est censée accueillir. En effet, cette notion de dépendance fonctionnelle doit rester vraie dans le futur pour nous fournir les garanties de maintenabilité.

Si on note a_k des colonnes de R et A_k des ensembles de colonnes de R, on observe les propriétés suivantes (dites axiomes d'Armstrong) :

- **réflexivité :** si $A_i \subseteq A_j$ alors $A_j \rightarrow A_i$;
- **augmentation :** si $A_i \rightarrow a_j$ alors $A_i \bigcup \{a_k\} \rightarrow a_j$;
- **additivité :** si $A_i \rightarrow a_j$ et $A_k \rightarrow a_l$ alors $A_i \bigcup A_k \rightarrow \{a_j, a_l\}q$;
- **décomposabilité :** si $A_i \rightarrow \{a_j, a_k\}q$ alors $A_i \rightarrow a_j$ et $A_i \rightarrow a_k$;
- **transitivité :** si $A_i \rightarrow A_j$ et $A_j \rightarrow A_k$ alors $A_i \rightarrow A_k$.

Enfin, la notion de dépendance fonctionnelle nous propose une manière de décomposer une relation en plusieurs relations. Si l'on a une relation $R(a_i, a_j, a_k, \ldots, a_n)$ et que l'on a $a_i \rightarrow a_j$ alors on peut décomposer la relation R en deux relations $R1$ et $R2$:

- $R1(\underline{a_i}, a_j) = \Pi_{a_i, a_j}(R)$;
- $R2(a_i, a_k, \ldots, a_n) = \Pi_{a_i, a_k, \ldots, a_n}(R)$.

Cette décomposition garantit que l'on a pas perdu d'information, car on observe que $R = R1 \bowtie R2$.

4.2 Notion de forme normale

Les formes normales prescrivent, de manière croissante, des propriétés qui doivent être observées sur le schéma de la base de données pour garantir sa qualité. Lorsque le schéma ne respecte pas une propriété, il faut le modifier, typiquement en appliquant la décomposition que nous venons de voir au paragraphe précédent.

4.3 Première forme normale

Le schéma d'une base de données est en première forme normale (notée *1FN*) si :

- chaque valeur d'attribut est atomique.

Dans notre exemple, on peut considérer que la relation $Films$ ne respecte pas cette première forme normale car $MetteurEnScene$ n'est pas atomique mais composé d'un prénom et d'un nom. Pour passer en première forme normale, nous devrons donc remplacer la colonne actuelle par un identifiant de metteur en scène et créer une table $MetteursEnScenes$ pour conserver les correspondances entre ces identifiants et leurs noms et prénoms relatifs.

4.4 Deuxième forme normale

Le schéma d'une base de données est en deuxième forme normale (notée *2FN*) si :

- il est en première forme normale ;
- **et** tout attribut n'appartenant pas à la clef primaire ne dépend pas que d'une partie de la clef.

La deuxième forme normale vérifie donc que les attributs hors de la clef en dépendent totalement.

Comme la première forme normale contraint les attributs à être atomiques, la vérification se limite aux tables dont la clef primaire est composée. Dans notre exemple, la relation $Titres$ possède une clef composée et est bien en deuxième forme normale puisque l'on ne peut déterminer un titre si l'on ne possède que l'identifiant du film ou que la langue.

4.5 Troisième forme normale

Le schéma d'une base de données est en troisième forme normale (notée *3FN*) si :

- il est en deuxième forme normale ;
- **et** tout attribut n'appartenant pas à la clef primaire ne dépend pas d'un attribut ne faisant pas partie de la clef.

La troisième forme normale vérifie donc que les attributs hors de la clef dépendent totalement et uniquement de celle-ci.

Dans notre exemple, nous n'avons pas à vérifier que la relation $Titres$ est en troisième forme normale, puisqu'elle ne possède qu'un attribut en dehors de la clef. Dans la relation $Films$, nous devons vérifier que $MetteurEnScene$ ne dépend pas de l'année et vice-versa. C'est le cas, même s'il existe une contrainte (plus ou moins forte) portant sur le lien entre l'année et la vie du réalisateur. Notre relation est donc en troisième forme normale.

4.6 Autres formes normales

Il existe d'autres formes normales plus élaborées qui servent à garantir des propriétés non plus seulement sur le schéma de la base de données, mais sur les relations qui seront calculées à partir d'opérateurs de l'algèbre relationnelle. Bien que ces formes normales puissent avoir un intérêt, en pratique on se contente souvent d'un schéma relationnel en troisième forme normale.

On considère donc généralement qu'une base de données est normalisée si son schéma est en troisième forme normale. L'atomicité des valeurs garantit que l'on a bien décomposé le problème selon des domaines élémentaires. Les contraintes sur les clefs garantissent que l'on évite toute redondance d'information ; plutôt que de prendre le risque d'une redondance (qui poserait des problèmes de maintenabilité, puisqu'il faudrait modifier toutes les occurrences simultanément), on préfère décomposer une relation sachant que l'on a la garantie que l'information n'est pas perdue et sera retrouvée au prix d'une jointure.

4.7 Schéma révisé de notre exemple

Comme nous l'avons vu au fil de cette section, pour que le schéma de notre base de données soit normalisé, il faut introduire une nouvelle relation contenant les informations sur les metteurs en scène. On aboutit donc au schéma présenté à la figure Figure 4.1.

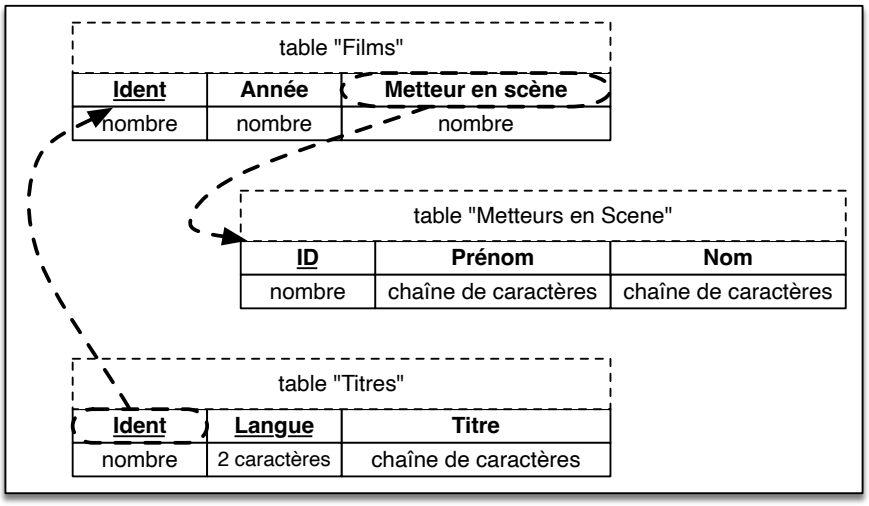

FIGURE 4.1 – Schéma normalisé.

5 Modélisation d'une base de données

Nous présentons brièvement dans cette section une méthode de modélisation d'une base de données. Une telle activité consiste à construire le schéma d'une base de données en partant des informations dont on dispose sur un domaine d'application. Les éléments que nous présentons ne sont que des guides de modélisation et ne garantissent pas que leur application mènera toujours à la même modélisation pour un problème donné.

5.1 Le dictionnaire des données

Le premier élément couramment utilisé lors de la modélisation d'un domaine d'application est le dictionnaire des données. Il consiste à obtenir par la lecture, l'interview ou l'audit le lexique le plus complet possible du domaine d'application. Typiquement, pour cerner les limites d'une base de données, on examine les processus qui ont cours dans une organisation.

Le but du dictionnaire de données est de déterminer le lexique du domaine. En plus de l'examen des processus actuels, il est souvent nécessaire de s'interroger sur les requêtes auxquelles la base de données à construire devra répondre. Dans cette phase préliminaire de recensement des besoins et des termes, il faut être prudent quant à de possibles homonymes.

Si nous souhaitons mettre en place une base de données pour gérer une filmothèque, il semble évident que nous devrons gérer au minimum les éléments suivants :

- des films, avec leurs informations techniques (titre, année, couleur, langue, durée, *etc*) ;
- des personnes, en lien avec les films (réalisateurs, scénaristes, acteurs, *etc*) ;
- des studios, en lien avec les films.

Par notre façon de lister les éléments, nous voyons émerger deux concepts qui vont être utilisés lors de la modélisation : les entités et les associations.

5.2 Entités et associations

Les entités représentent des éléments du domaine d'application et les associations représentent des liens entre ces entités. Pour représenter les entités et les associations, on utilise le plus souvent un graphe semblable au diagramme de classes d'UML. La figure Figure 5.1 donne un exemple de diagramme entités-associations.

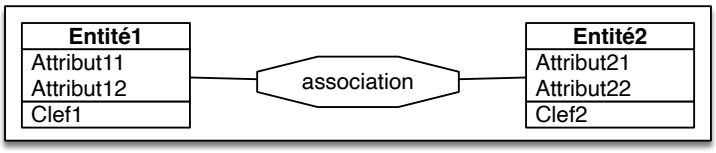

FIGURE 5.1 – Exemple de diagramme entités-associations.

On construit le diagramme entités-associations en partant du dictionnaire de données. On y identifie les objets manipulés et leurs attributs. On identifie dans chaque entité les clefs en utilisant le concept de dépendance fonctionnelle déjà évoqué. Dans le cadre de notre filmothèque, on peut imaginer obtenir les entités présentées à la figure Figure 5.2.

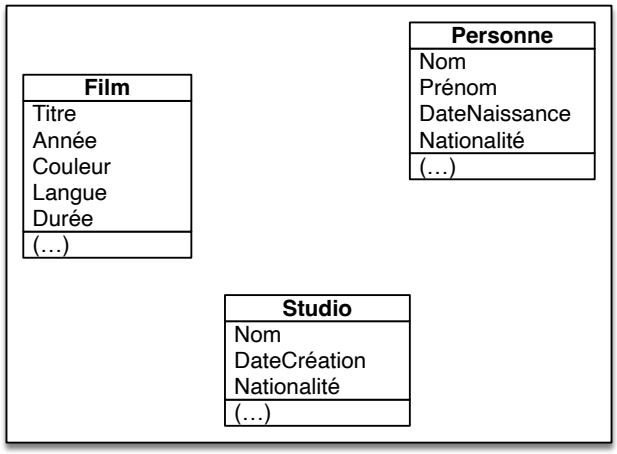

FIGURE 5.2 – Filmothèque : entités seules.

L'étape suivante consiste à identifier les associations entre les entités. La plus simple est l'association entre un studio et un film, que nous allons identifier par un verbe : "produit". L'association entre une personne et un film est un peu plus complexe car elle implique une notion d'activité. Nous allons appeler cette association "participe" et allons indiquer sur le schéma que cette association possède un attribut nommé "activité". Il est d'usage d'employer des substantifs pour les entités et les attributs mais des verbes pour les associations. Nous obtenons alors le diagramme entités-associations correspondant à la figure Figure 5.3.

Il reste une dernière étape pour obtenir un modèle entités-associations complet. Il faut établir pour chaque trait reliant une association et une entité quelle est sa cardinalité. Les cardinalités sont les mêmes que dans un diagramme de classes UML, à savoir :

- 0..1 : au plus un lien ;
- 1..1 : un et un seul lien ;
- 0..n : un nombre arbitraire de liens ;
- 1..n : au moins un lien.

Nous devons établir les cardinalités pour les quatre liens, soit répondre aux questions suivantes :

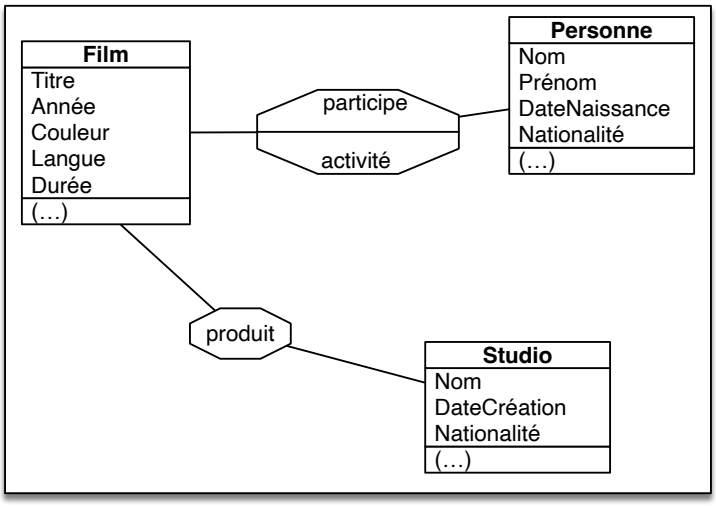

FIGURE 5.3 – Filmothèque : entités et associations.

1. combien un studio produit-il de films ?
2. par combien de studios un film est-il produit ?
3. combien de personnes participent-elles à un film ?
4. combien de fois une personne participe-t-elle à un film ?

auxquelles nous apportons respectivement les réponses :

1. un studio produit un nombre quelconque de films ;
2. un film est produit par un unique studio (discutable) ;
3. un nombre arbitraire mais non nul de personnes participent à un film ;
4. une personne peut participer un nombre quelconque de fois à un film (*p. ex.* combinaison de l'écriture et de la réalisation).

Nous aboutissons donc au diagramme entités-associations définitif montré à la figure Figure 5.4.

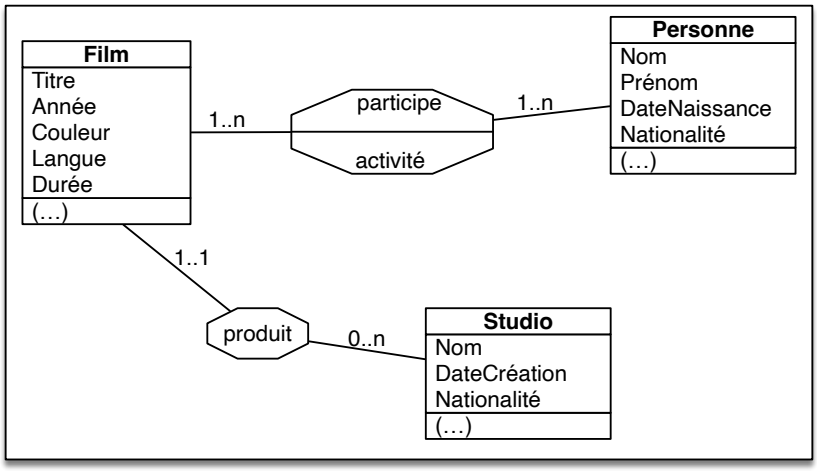

FIGURE 5.4 – Filmothèque : diagramme complet (entités et associations avec cardinalités).

5.3 Traduction en tables

Une fois le diagramme entités-associations défini, il ne reste plus qu'à le traduire en tables. Cela se fait de la manière suivante :

- chaque entité donne lieu à une table ;
- lorsqu'une association porte une cardinalité 0..1 ou 1..1, on la traduit par l'ajout d'une clef étrangère dans l'entité (l'attribut pouvant être égal à `NULL` ssi la cardinalité est 0..1) ;
- lorsqu'une association porte des cardinalités 0..n et 1..n, on la traduit par une nouvelle table.

La table ajoutée dans le dernier cas suit les règles suivantes : ses attributs sont la concaténation des clefs primaires des entités mises en relation (qui constituent la clef composite) et des possibles attributs de l'association.

Dans le cadre de notre exemple, si l'on choisit de distribuer des identifiants arbitraires comme clefs primaires pour les différentes entités on obtient le schéma suivant :

```
CREATE TABLE Studio
  (No_studio ... NOT NULL,
   Nom ... NOT NULL,
   DateCreation ... NOT NULL,
   Nationalite ... NOT NULL,
   PRIMARY KEY (No_studio))

CREATE TABLE Film
  (No_film ... NOT NULL,
   Titre ... NOT NULL
   Couleur ... NOT NULL,
   Langue ... NOT NULL,
   Duree ... NOT NULL,
   Produit_par ... NOT NULL,
   PRIMARY KEY (No_film),
   FOREIGN KEY (Produit_par) REFERENCES Studio(No_studio))

CREATE TABLE Personne
  (No_personne ... NOT NULL,
   Nom ... NOT NULL,
   Prenom ...  NOT NULL,
   DateNaissance ...  NOT NULL,
   Nationalite ... NOT NULL,
   PRIMARY KEY (No_personne))

CREATE TABLE Participe
  (No_film ... NOT NULL,
   No_personne ... NOT NULL,
   Activite ... NOT NULL,
   PRIMARY KEY (No_film, No_personne, Activite),
   FOREIGN KEY (No_film) REFERENCES Film(No_film),
   FOREIGN KEY (No_personne) REFERENCES Personne(No_personne))
```

L'ultime étape consiste à vérifier que le schéma obtenu est bien en troisième forme normale, ce qui est effectivement le cas dans notre exemple si l'on suppose que les valeurs stockées dans les types de données que nous avons omis sont atomiques.

6 NoSQL

6.1 Émergence des bases de données NoSQL

Les bases de données dites NoSQL (signifiant, selon les auteurs, soit "*No SQL*" soit "*Not Only SQL*") sont apparues au milieu des années 2000 en réponse à un certain nombre d'insuffisances des bases de données SQL. Les principaux reproches faits aux bases de données SQL sont :

- une opération centrale, la jointure, qui est d'un coût prohibitif dans certains cas (en effet, la jointure entre deux tables implique d'examiner l'ensemble des combinaisons dont le nombre est le produit des cardinalités des tables jointes) ;
- une gestion *sous-optimale* des accès concurrents (en effet, du fait notamment des jointures, une modification de la base de données implique de placer en attente d'autres modifications concurrentes) ;
- une gestion difficile de la distribution, *i. e.* de la capacité à répartir les informations sur plusieurs machines (en effet, la jointure implique de centraliser les informations des tables considérées) ;
- une notion de schéma que certains trouvent trop rigide (en effet, notamment dans le cadre de données *bruitées* ou dont le format évolue souvent, on souhaiterait bénéficier de plus de souplesse dans la structuration des données) ;
- un problème d'impédance entre le modèle relationnel et les langages, le plus souvent objets, utilisés pour se connecter à la base de données.

Bien entendu, ces problèmes et/ou limitations du modèle relationnel sont bien connus depuis très longtemps. Si leur importance a conduit à l'émergence des bases NoSQL depuis seulement une petite vingtaine d'années, c'est parce que ces problèmes ont pris littéralement une autre dimension depuis l'apparition du *big data*. En particulier, l'explosion de la taille des données rend l'utilisation de jointures strictement impossible dans certains cas. En outre, ces énormes bases de données sont souvent consultées et mises à jour par de nombreux processus concurrents, ce qui aggrave les problèmes de performance.

Les bases de données NoSQL ne prétendent pas proposer un modèle en tous points supérieur au modèle SQL. Elles se contentent de manière plus pragmatique de proposer des compromis différents sur les questions clefs de :

- la distribution ;
- la disponibilité ;
- les performances.

Enfin, il est important de noter que le terme NoSQL regroupe de nombreuses approches très différentes, contrairement au terme SQL qui correspond à un seul et unique modèle (qui a en outre le statut de standard). Les sections suivantes présentent les principales familles de bases de données NoSQL.

6.2 Bases de données orientées documents

Les bases de données orientées documents sont probablement les bases de données NoSQL les plus largement utilisées aujourd'hui, notamment via des systèmes tels que MongoDB [1] et CouchDB [2].

Comme leur nom l'indique, le principe de ces bases de données est de stocker dans les tables des documents, un document étant semblable aux structures JSON ou XML (*cf.* séance sur l'interopérabilité). De tels documents sont des structures arborescentes qui organisent un système de propriétés associant clefs et valeurs. Par exemple, le document suivant :

```
{
  "auteur": {
    "prenom": "Nicolas",
    "nom": "Machiavel"
  },
  "titre": {
    "lang": "fr",
    "intitule": "Le prince"
  }
}
```

représente un livre qui est défini par deux propriétés, `auteur` et `titre`. La première est elle-même composée de deux sous-propriétés `prenom` et `nom`, tandis que la seconde est composée de deux sous-propriétés `lang` et `intitule`.

Dans les bases de données orientées documents, les documents tels que le précédent remplacent conceptuellement les tuples des bases de données SQL. Là où les données SQL sont *à plat*, les données sous forme de documents sont hiérarchiques.

Un second point différenciant nettement ces bases de données des bases SQL est qu'il n'existe en général pas de notion de schéma. Ainsi, les différents éléments stockés dans une même table ou collection, n'ont pas nécessairement la même structure. L'absence de schéma entraîne, assez logiquement, la disparition de la plupart des contraintes d'intégrité qui peuvent être définies dans les bases de données. De fait, seule la notion de clef primaire est conservée, avec le même rôle que dans les bases de données SQL. Elle est généralement distribuée automatiquement si elle n'est pas spécifiée lors de la création d'un enregistrement.

6.3 Bases de données orientées clefs-valeurs

Les bases de données clefs-valeurs peuvent être vues comme une simplification des bases de données orientées documents. L'intérêt de cette simplification n'est pas tant de proposer un modèle plus simple à appréhender pour le programmeur, qu'un modèle qui permette d'obtenir de meilleures performances au détriment de l'expressivité.

La différence essentielle avec les bases orientées documents est la disparition de la dimension hiérarchique des documents, ne laissant donc plus qu'un ensemble de propriétés sous la forme d'associations clef-valeur. Ainsi, l'exemple de la section précédente serait représenté de la manière suivante :

1. http://www.mongodb.org
2. http://couchdb.apache.org

```
{
  "auteur_prenom": "Nicolas",
  "auteur_nom": "Machiavel",
  "titre_lang": "fr",
  "titre_intitule": "Le prince"
}
```

Les bases de données clefs-valeurs sont plus simples à implémenter que les bases orientées documents, et proposent de meilleures performances. Les données sont également légèrement plus simples à distribuer dans ce modèle, dont les éléments atomiques sont de plus petite taille. Comme dans les bases orientées documents, la notion de schéma est généralement absente.

De fait, le modèle des bases de données clefs-valeurs est proche de celui des bases de données relationnelles. L'absence de schéma, et surtout de contraintes d'intégrité telles que les clefs étrangères, fait toute la différence en évitant des opérations coûteuses qui mettraient en œuvre plusieurs tables.

6.4 Bases de données orientées graphes

Les bases de données orientées graphes ne sont actuellement que peu utilisées, mais sont celles dont la croissance est la plus forte. Comme leur nom l'indique, le principe de ces bases de données est de stocker des graphes, tels que celui représenté par la figure Figure 6.1.

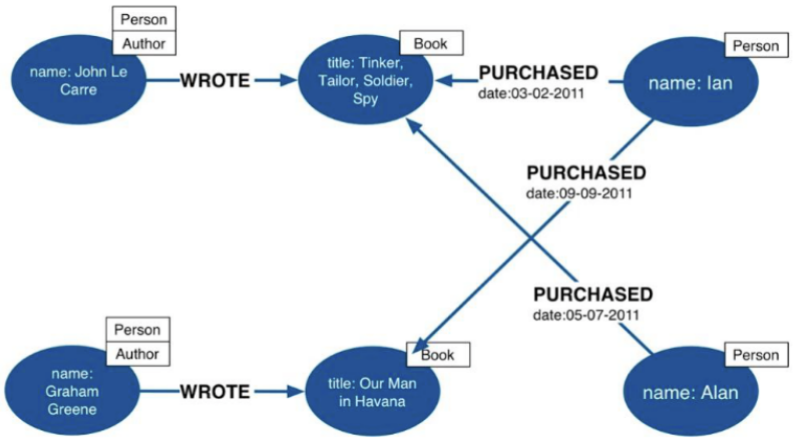

FIGURE 6.1 – Exemple de graphe (source : neo4j.com).

Une base de données graphes permet le stockage de graphes, donc de nœuds et d'arcs qui peuvent être étiquetés, comme le montre la figure par des propriétés (par exemple, `title` ou `date` dans l'exemple). Dans le cas de l'exemple, issu de la base de données Neo4j [3], il est également possible de donner une *classe* à chaque nœud, qui en retour impose la présence de certaines propriétés.

La structure des données gérées par les bases orientées graphes est totalement différente des autres bases de données évoquées. On peut au plus comparer les étiquettes d'un nœud ou arc à un système clefs-valeurs. Le principal avantage des bases orientées graphes est de permettre notamment d'écrire des requêtes portant sur des chemins du graphe de manière native, alors que cela nécessiterait des développements spécifiques sur les autres systèmes de bases de données.

3. https://neo4j.com

6.5 Accès aux bases de données NoSQL depuis un langage de programmation

La plupart des bases de données NoSQL ne définissent pas, contrairement au modèle relationnel, un langage propre d'interrogation de la base. Elles se contentent de proposer des APIs permettant de les utiliser depuis un langage de programmation. Il est important de noter que les modèles orientés documents et clefs-valeurs peuvent être simplement représentés dans un langage tel que Python par un système de dictionnaires (ou d'attributs d'instances) imbriqués. Cela signifie en pratique que ces systèmes de bases de données ne souffrent pas d'un problème d'impédance.

6.6 Comparaison SQL / NoSQL

Éléments de base

Il est important de rappeler que SQL est un standard, alors que chaque système NoSQL est différent des autres (même s'il existe des familles au sein de NoSQL). Il est ainsi facile de passer d'une implémentation SQL à une autre, à très faible coût.

En ce qui concerne les moyens d'accéder aux données stockées depuis un langage de programmation, on notera principalement que :

- SQL est un langage déclaratif alors que les système NoSQL utilisent des APIs ;
- en général, les systèmes NoSQL sont de *plus bas niveau*, ce qui en pratique donne davantage de contrôle et de meilleures performances au prix d'une moindre expressivité.

Théorème CAP

Lorsque l'on cherche à caractériser le comportement et les caractéristiques d'un système de bases de données, on est intéressé par en particulier par trois dimensions :

- *Consistency* (cohérence) ;
- *Availability* (disponibilité) ;
- *Partition tolerance* (tolérance à la partition/distribution).

La cohérence est le fait que la base de données réponde toujours avec la dernière version des données, en particulier lorsqu'elles sont réparties sur plusieurs machines.

La disponibilité est le fait de répondre sans délai à une interrogation de la base de données, quel que soit en particulier le nombre d'accès concurrents.

La tolérance à la partition/distribution est le fait que la base de données puisse être stockée non pas sur une seule machine, mais sur un ensemble de machines.

Ce que dit le *théorème* CAP, c'est qu'un système de base de données ne peut donner satisfaction que sur deux des trois dimensions. Pour s'en convaincre, de manière intuitive, il suffit de considérer que si l'on souhaite par exemple avoir simultanément la cohérence et la partition, il faut imaginer un mécanisme de propagation des modifications entre les différentes machines qui nécessitera un certain temps et ne permettra pas une disponibilité optimale.

L'intérêt du théorème, outre son résultat *d'impossibilité*, est qu'il permet de placer voire classifier les systèmes de bases de données selon ces trois dimensions. Les bases de données SQL accordent la priorité absolue à la cohérence. Les bases de données NoSQL effectuent d'autres compromis. Par exemple, MongoDB choisit de donner la priorité à la disponibilité et à la partition, au détriment de la cohérence.

Il est donc crucial d'avoir une idée précise des opérations à effectuer sur les données stockées afin de savoir quel système de base de données choisir. Par exemple, pour une application bancaire, la cohérence est vraisemblablement la propriété la plus importante tandis que pour un moteur de recherche la propriété la plus importante est la disponibilité ou la partition.

Annexes

À retenir

Concernant l'utilisation de SQL depuis Python, il est utile de connaître les différents modes d'appels, ainsi que les abstractions sur lesquelles ils reposent. Il est important de comprendre les raisons pour lesquelles des ORMs sont développés.

Concernant la modélisation de bases de données, les notions de dépendance fonctionnelle et de forme normale sont utiles, de même que les schémas entités-associations qu'il peut être intéressant de mettre en parallèle des diagrammes UML.

Concernant les bases de données NoSQL, il est important de comprendre pourquoi elles sont apparues, ainsi que les problèmes qu'elles cherchent à résoudre. Il peut être intéressant d'avoir une idée, même sommaire, des critères d'arbitrages entre SQL et NoSQL.

Références

— "Bases de données - Concepts, utilisation et développement" (Jean-Luc Hainaut, Dunod) propose une introduction complète aux divers aspects des bases de données relationnelles.
— "Algèbre relationnelle - Guide pratique de conception d'une base de données relationnelle normalisée" (Michelle Clouse, ENI éditions) propose une vision complète de l'algèbre relationnelle et de la modélisation.
— http://docs.python.org/library/sqlite3.html regroupe la documentation du module `sqlite3`
— http://storm.canonical.com/Manual regroupe la documentation de Storm