Bases de données

PTSI Lycée Eiffel

28 février 2014

Introduction

Pour ce troisième chapître, nous allons mettre de côté notre apprentissage de Pyhton pour nous intéresser à un sujet fondamental dans l'informatique actuelle : les bases de données. La gestion efficace d'énormes quantités de données est une problématique essentielle pour tous ceux qui travaillent en lien avec Internet (songez au fonctionnement d'un moteur de recherche comme Google par exemple), mais aussi, à plus petite échelle, pour les futurs ingénieurs que vous serez : comment traiter de façon efficace des données numériques issues de batteries de tests et accéder le plus rapidement possible aux informations pertinentes ? Les bases de données sont en soi un concept très simple : il s'agit d'organiser des données de façon à en optimiser le traitement, c'est-à-dire en gros les mises-à-jour et les recherches. Le but de ce chapître n'est pas de devenir un spécialiste technique des bases de données (nous n'apprendrons pas vraiment comment fonctionne une bases de données « en interne »), mais plutôt d'en devenir un utilisateur expert.

La première utilité des bases de données est très simple : stocker des données. Certes, mais on dispose déjà pour cela sur notre cher ordinateur d'un système de fichiers qui, utilisé correctement, est un outil assez efficace dans ce domaine. Le fonctionnement d'une base de données est subtilement différent. En gros, le système de fichiers se contente de gérer l'organisation des données (et encore, de façon assez simpliste). Si on souhaite simplement accéder aux données en question, on le fera en « ouvrant » le fichier, c'est-à-dire en accédant aux données du fichier à l'aide d'un logiciel externe. De même pour toutes les mises-à-jour de données. Quant aux recherches au sein du système de fichiers, elles sont quasiment inexistantes puisqu'on ne peut en gros faire que des recherches portant sur le nom du fichier.

Définition 1. Une **base de données** est une structure gérant la collecte, le stockage, les mise à jour et l'accès à un (gros) ensemble de données. Cette structure permet en particulier d'optimiser les mises à jour et les recherches, de sécuriser le système de données, et de gérer les priorités d'accès et les éventuels conflits en cas de travail à plusieurs sur les mêmes données

Remarque 1. En fait, la base de données à proprement parler n'est constituée que des données, et le système complet devrait être appelé plus proprement **système de gestion de bases de données** (SGBD en abrégé), mais on fait fréquemment la confusion entre les deux. Nous continuerons à utiliser le terme générique base de données tout au long du chapître.

Travailler avec des bases de données peut se faire de bien des façons différentes. On peut en gros distinguer les différents types d'utilisation suivants :

- utilisateur « naïf » : se contente d'effectuer des recherches dans la base de données.
- utilisateur « actif » : effectue des recherche mais également des mises à jour (création, modification, destruction de données) sur la base.
- administrateur : gère (et éventuellement écrit) les programmes sous-jacents dans la structure de la base de données.
- concepteur : décide de l'organisation de la base de données avant son implémentation informatique.

Nous nous contenterons cette année d'essayer de maîtriser, dans un premier temps, le rôle de concepteur de bases de données (qui est tout à fait théorique, on peut très bien effectuer le travail correspondant « sur papier »), puis celui d'utilisateur actif. Nous laisserons par contre de côté la plupart des aspects techniques afférents au travail de l'administrateur. Autrement dit, nul besoin de savoir programmer dans ce chapître, les seuls éléments de langage informatique que nous apprendrons seront ceux permettant d'effectuer des requêtes (en gros des recherches) dans notre base de données.

1 Construction d'une bases de données : modèle entité/associations

L'étape de conception d'une base de données est primordiale. En effet, une fois la structure de la base créée, et un grand nombre de données insérées, il est extrêmement délicat de la modifier. Il faut donc bien réfléchir à l'organisation des données avant même de créer la base. Pour cela, on dispose de plusieurs façons de représenter les choses (ce qu'on appelle des modèles), nous en étudierons deux dans ce cours. Pour vous donner une base de réflexion, prenons l'exemple suivant (qui nous servira dans tout le chapître : vous décrochez un petit boulot de stagiaire pour l'été prochain chez Alloc**é (pas de pub !), et vous êtes en charge de la base de données du site. Bien évidemment, vous n'aurez pas à concevoir vous-même la base, mais imaginez le travail nécessaire pour organiser intelligemment des centaines de milliers de données : quels sont les éléments qu'on a envie de connaître pour chaque film, chaque acteur, chaque réalisteur ? Comment les relie-t-on entre eux ? Si on veut que la base de données puisse nous sortir facilement, par exemple, la liste de tous les westerns des années 50 dont l'un des trois rôles principaux est tenu par une femme, comment fera-t-on ? Pour ce dernier point, c'est la base de données qui gère, mais encore faut-il qu'elle soit conçue de façon rationnelle. Mais avant de nous lancer dans le cinéma, donnons un exemple de liste de données (je ne parle volontairement pas ici de base de donnée) dans un autre domaine, qui va nous fournir un bon exemple de ce qu'il ne faut surtout pas faire, et justifier par ce biais l'abandon d'un simple système de fichiers pour une structure mieux organisée.

Année	Vainqueur	Nationalité	Finaliste	Nationalité	Sets
2013	Nadal	Espagnol	Ferrer	Espagnol	3
2012	Nadal	Espagnol	Djokovoc	Serbe	4
2011	Nadal	Espagnol	Federer	Suisse	4
2001	Kuerten	Brésilien	Corretja	Espagnol	4
1999	Agassi	Américain	Medvedev	Ukrainien	5
1983	Noah	Français	Wilander	Suédois	3
2013	S.Williams	Américaine	Sharapova	Russe	2

Comme les plus sportifs d'entre vous l'auront compris, il s'agit d'une liste partielle de finales du tournoi de tennis de Roland-Garros. La liste pourrait bien sûr être beaucoup plus étendue, mais on peut déjà constater sur ce maigre échantillon les problèmes que va poser cette organisation :

- il y a une redondance manifeste d'informations : inutile de préciser huit fois que Nadal est espagnol, une seule suffirait. Cela risque d'ailleurs de poser des problèmes en cas de mise à jour : si Nadal décide soudain de se faire naturaliser qatari, il faudra bien le signaler sur toutes les lignes où il apparait.
- le classement des lignes n'est pas évident. J'ai choisi de mettre les Dames à la fin (ce n'est pas très galant) mais j'aurais pu faire autrement. Et si je cherche des informations sur la finale Hommes 2013, comment savoir quelle ligne est la bonne puisqu'il y en a deux qui sont numérotées ainsi ? Bien sûr, c'est un problème très mineur sur une liste aussi courte que celle-ci, mais imaginez une vraie base de données contenant le même genre d'informations pour tous les tournois de la saison, il faudra très vite organiser les choses un peu mieux.
- autres soucis de mise à jour : si je décide de supprimer une édition pour une raison ou pour une autre, par exemple 1999 parce qu'Agassi a avoué s'être dopé et que je n'en veux plus dans

ma liste, je risque de détruire en même temps d'autres informations que je souhaiterais garder (le pauvre Medvedev qui n'y est pour rien va disparaitre du même coup).
- et si je veux insérer mon tennisman préféré, Jo-Wilfried Tsonga, dans la liste ? Ah ben je ne peux pas, il n'est jamais allé en finale de Roland-Garros !

Bref, ça ne va pas du tout. Une façon assez logique de résoudre le problème serait de présenter non pas une seule liste de données, mais deux : une pour les finales, avec l'année, le nombre de sets et les deux protagonistes, et une autre contenant une liste de joueurs et joueuses avec toutes les données personnelles que l'on souhaite avoir sur eux. Il ne restera plus alors qu'à essayer de faire un lien entre les joueurs apparaissant dans la liste des finales, et la liste des joueurs. C'est exactement le genre de choses que va nous permettre de réaliser le premier modèle que nous allons étudier, le modèle Entité/Association.

Définition 2. Une **entité** est un objet, une personne, un concept, etc, identifiable et pertinent dans le cadre de notre base de données.

Dans notre exemple cinématographique, une entité peut ainsi très bien être un film, un réalisateur, un festival, ou bien d'autres choses. Le concept reste assez vague, mais l'idée est qu'une entité pourra correspondre à une ligne d'un tableau dans notre base de données. Le principe de l'identification est extrêmement important : il faut pouvoir identifier de façon certaine et unique chaque entité. On devra pour cela disposer d'une caractéristique appelée **clé primaire** qui doit être unique. Par exemple, dans une liste de films, il serait dangereux d'utiliser le titre du film comme clé primaire, car on peut imaginer deux films différents ayant le même titre (et cela va compliquer le fonctionnement de la base). Pour éviter ce problème, on prendra très souvent comme clé primaire un identificant artificiel, souvent un numéro d'identification.

Une entité contiendra également en général plusieurs **attributs**, caractéristiques enregistrées dans la base de données (pour nos films, on peut imaginer comme attributs le titre, le réalisateur, la date de sortie, la durée, le genre, et quantité d'autres), qui prendront leur valeurs dans des **domaines** spécifiés à l'avance (ainsi, le titre d'un film sera une chaîne de caractères, sa durée un entier etc. On commence à se rapprocher de la notion de variables typées dans un langage de programmation).

Une entité est **définie** par son nom, ses attributs, et sa clé primaire (qui est en général un des attributs). On la représentera de la façon suivante :

Nom
Clé primaire
Attribut2
Attribut3
⋮

Remarque 2. Les plus attentifs d'entre vous auront remarqué que j'entretenais une certaine confusion au sujet de la notion d'entité. Je vous ai dit un peu plus haut qu'une pouvait être un film ou un réalisateur, alors que dans la définition ultérieure, une entité semble plutôt être un concept abstrait général, comme « film » ou « réalisateur », et pas un cas particulier de ce concept. En fait, on parle de **type d'entité** pour désigner le concept général, l'entité en étant bien un cas particulier. On dit également qu'un film particulier sera une **instance** du type d'entité « Film » (ce que j'abrégerai souvent, et très improprement, en parlant d'instance de l'entité « Film »). Un exemple plus concret de (type d') entité pour finir :

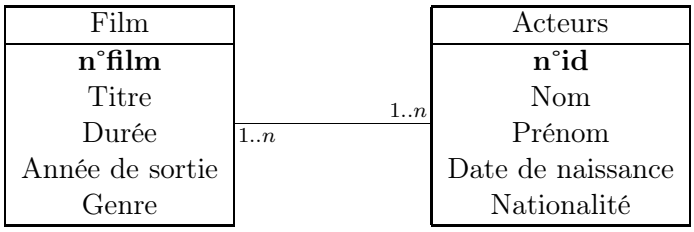
Acteurs
n°id
Nom
Prénom
Date de naissance
Nationalité

Définition 3. Une **association** est un lien entre deux entités.

Remarque 3. Voila encore une définition bien floue. Techniquement, une association entre deux entités E_1 et E_2 est simplement un sous-ensemble de $E_1 \times E_2$. Ainsi, si on dispose d'une entité Film et d'une entité Acteurs, on pourra créer une association « Joue » qui relie à un film tous les acteurs ayant un rôle dans le film.

Définition 4. La **cardinalité** d'une association est constituée de deux couples d'entiers (i, j), représentant le minimum et le maximum d'éléments de E_1 associés à un élément donné de E_2 (et vice-versa).

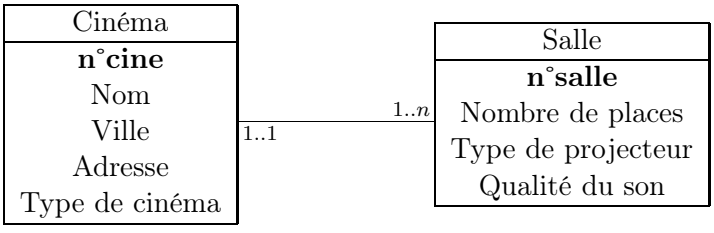
Lorsqu'on réalise un schéma d'une base de données dans le modèle Entité/Association, les entités sont représentées par des rectangles, comme on l'a déjà vu, et les associations par de simples segments reliant les deux entités. On indique aux extrêmités du segment les deux couples constituant la cardinalité sous la forme $i..j$. Dans le cas où il n'y a pas de maximum, on notera un n à la place du j. Ainsi, une association de type 1..1 associe à tout élément de la première entité un unique élément de la deuxième, alors qu'une association de type $0..n$ peut associer à un même élément plusieurs éléments, ou même aucun. si on reprend notre exemple initial sur Roland-Garros, avec une entité Finales et une entité Joueurs, on les relierait par une association de type 2..2 dans le sens Finales ⇒ Joueurs (deux joueurs, ni moins ni plus, pour chaque finale), et de type $0..n$ dans le sens Joueurs ⇒ Finales (un joueur peut ne jamais avoir atteint la finale, et au contraire avoir participé à plusieurs finales). Pour reprendre un exemple plus cinématographique, on peut désormais relier notre entité Film et notre entité Acteurs et le représenter de la façon suivante (on peut se poser des questions existentielles concernant les cardinalités : un film peut-il ne pas avoir d'acteurs ? Un acteur peut-il ne jouer dans aucun film ? Il est important de répondre à ces question au moment de la conception de la base sous peine d'avoir de gros problèmes ensuite si on tombe sur un cas ne rentrant pas dans le cadre prévu) :



Le modèle que nous venons de présenter est très simple mais suffisant pour schématiser raisonnablement des bases de données pas trop complexes. On peut ajouter une ou deux précisions pour compléter un peu la description que nous venons de faire :

Définition 5. Une entité donnée est une **entité faible** d'une autre entité si elle est en lien de cardinalité 1..1 (dans le sens entité faible ⇒ entité forte) et 1..n avec cette dernière.

Le plus simple pour comprendre ce concept est de voir ce qui se passe sur un exemple : dans notre base de données cinéma, on crée une entité Cinéma visant à compiler la liste des complexes cinématographiques, et une entité Salle qui se restreindra à une seule salle à la fois. On a une association logique entre les deux entités (l'appartenance de la salle à un cinéma donné) qu'on peut classiquement représenter ainsi :



Cette présentation est toutefois peu adaptée, notamment en ce qui concerne l'identification des salles, puisqu'on doit alors numéroter les salles (toutes les salles de tous les cinémas) de façon unique, et donc indépendante du cinéma de rattachement. Il serait plus cohérent d'avoir une numérotation des salles internes à chhaque cinéma. On peut régler ce problème en rajoutant dans l'entité Salle un attribut n°cine (identique à celui présent dans l'entité Cinéma) et considérant que la clé primaire de l'entité Salle est constituée du couple (n°cine, n°salle). On peut alors représenter le lien de façon particulière sur notre schéma pour indiquer cette caractéristique « entité faible ». Nous ne le ferons pas en général, mais nous garderons cette notion à l'esprit dans notre prochaine partie.

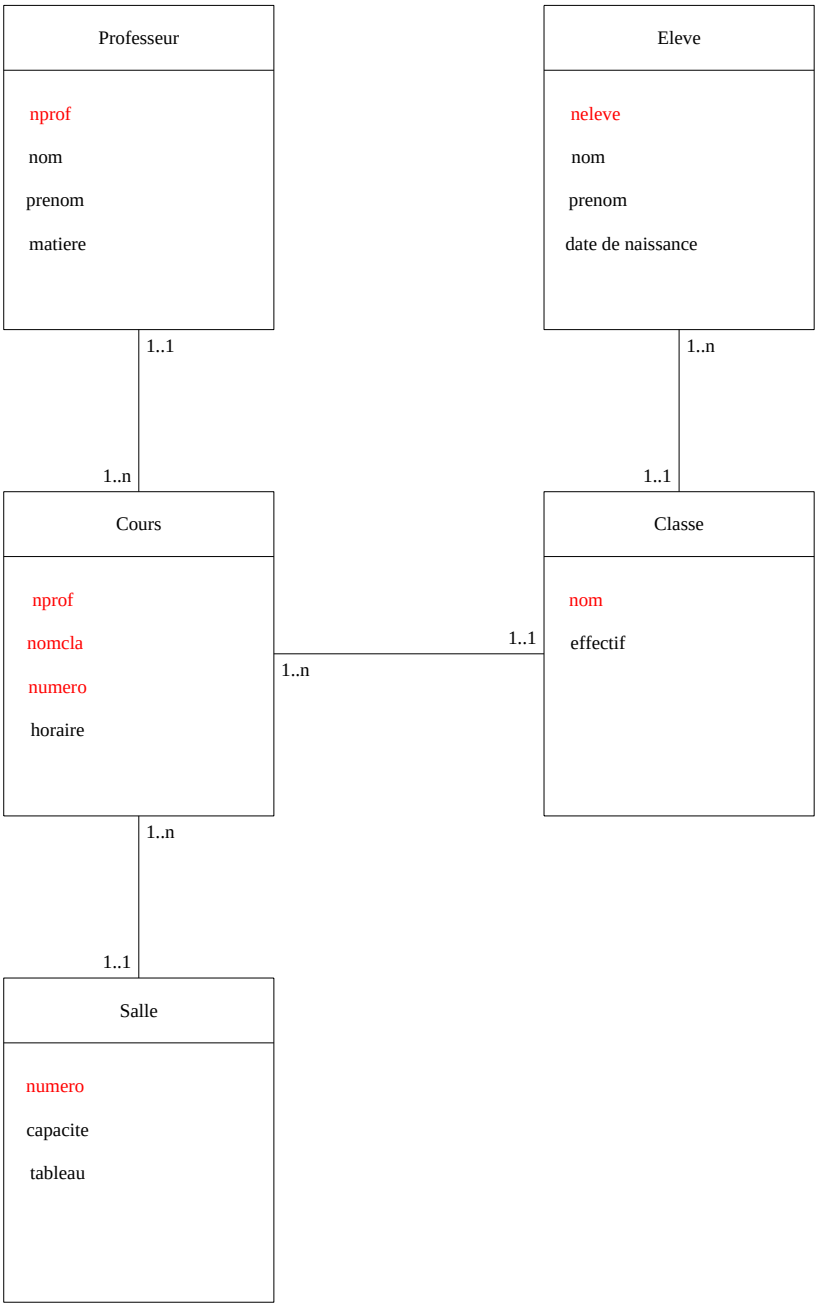
Une autre question qu'on peut naturellement se poser : est-il possible de créer des associations entre trois entités (ou plus) au lieu de deux ? Oui, bien sûr ! Mais le schéma va devenir compliqué, notamment au niveau de la représentation des cardinalités. Il est en fait plus pertinent dans ce cas de créer une nouvelle entité correspondant à l'association. Nous verrons un cas concret un peu plus bas (dans ce cas, en général, la clé primaire de cette nouvelle entité sera constitué du regroupement de toutes les clés primaires des entités qu'elle relie).

Malgré tous nos efforts, le modèle Entité/Association reste assez limité, notamment parce qu'il impose de faire énormément de choix au moment de la conception. On peut très représenter un lien comme une association ou comme une entité à part, sans que l'une des deux possibilités ne soit manifestement supérieure. Par ailleurs, il ne fait pas apparaitre les **contraintes d'intégrité**, c'est-à-dire les liens « logiques » entre les entités ou leurs attributs. Dans l'exemple que nous allons développer ci-dessous, rien n'interdit dans notre schéma de créer deux cours avec des classes différentes et des professeurs différents, mais ayant lieu dans la même salle à la même heure !

Pour terminer notre étude de ce premier modèle, essayons de créer un schéma Entité/Association représentant (sans trop rentrer dans les détails) l'organisation du lycée Eiffel. On peut raisonnablement penser penser aux entités suivantes :

- Professeur (commençons par le plus important), avec comme attributs nom, prenom, matiere (on se restrein volontairement pour l'exemple à un nombre d'attributs limité).
- Eleve, avec comme attributs nom, prenom, date de naissance.
- Classe, avec comme attributs nom, effectif.
- Salle, avec comme attributs numero, capacite, tableau (type de tableau présent dans la salle).

Il faudra bien sûr ajouter des numéros d'identification un peu partout pour servir de clés primaires, et surtout des associations. Il peut paraitre logique de lier les professeurs et les élèves aux classes, mais ce n'est pas forcément si pertinent que cela, car il faudra de toute façon créer une association « Cours » qui reliera simultanément un professeur, une salle, et une classe. En tant qu'association ternaire, on va plutôt la transformer directement en entité. Du coup, on reliera directement les profs, les salles et les classes à l'entité Cours, et seulement les élèves (en tant qu'entité faible si on admet qu'on est dans un cas simple où un élève appartient à une et une seule classe) aux classes. On peut mettre comme attributs à l'entité Cours l'horaire du cours, auquel on ajoute tout simplement les trois clés des entités Professeur, Classe et Salle, ce triplet constituant la clé primaire de l'entité Cours. On a déjà un schéma assez satisfaisant, qu'on peut représenter de la façon suivante :



Les attributs nprof et neleve représentent des numéros d'identification, les clés primaires sont comme vous l'aurez déviné mises en rouge.

2 Modèle relationnel

Le but de ce nouveau modèle est de remplacer le modèle Entité/Association par quelque chose de plus proche de ce qu'on va réellement intégrer informatiquement parlant (en gros une liste de tableaux), notamment en imposant une seule structure, la **relation** (aussi appelée directement **table**). Plus d'associations entre les tables, tout doit être directement intégré dans la structure de la table.

2.1 Description du modèle

Définition 6. Une **relation** est définie par son nom et par une liste d'attributs prenant leurs valeurs dans des domaines spécifiés à l'avance.

Bon, de qui se moque-t-on dans ce cours ? La définition ci-dessus est un copier-coller honteux de celle d'entité donnée un peu plus haut. Eh bien oui, il n'y a pas vraiment de différence fondamentale, même si on ne les manipulera pas tout à fait de la même façon. Pour ne pas trop entretenir la confusion, on va tout de même noter nos relation un peu différemment, sous la forme Relation(attribut1 : domaine1, attribut2 : domaine2, ...). C'est ce qu'on appelle le **schéma** de la relation (la relation elle-même étant le tableau et pas l'objet abstrait caché derrière). Par exemple, on peut définir une relation Film(titre : varchar, real : varchar, annee : integer). Les types de données varchar et integer seront définis plus loin, même si vous devez pouvoir deviner ce que ça représente sans trop de problèmes. Une **instance** de la relation sera une ligne du tableau correspondant (donc un élément de $D_1 \times D_2 \times \ldots$, en notant D_i le domaine dans lequel l'attribut numéro i prend ses valeurs). Un tel élément est également appelé **tuple** de la relation.

Une base de données est alors simplement constituée d'une accumulation de relations. Je vous sens venir avec vos questions impertinentes : mais où sont passés les liens entre les différentes entités qu'on appelle plus entités mais relations mais qui sont quand même la même chose ? Ne vous inquiétez pas, ça vient, mais en attendant, demandons-nous déjà ce qu'est devenu le concept de clé primaire. C'est très simple, il est transporté tel quel, la **clé primaire** d'une relation étant un sous-ensemble de sa liste d'attributs. On représentera bien sûr une relation complète (et pas simplement son schéma) sous forme de tableau, comme dans l'exemple suivant (toute inexactitude dans les données restranscrites étant totalement fortuite), pour la relation Film(n°id,titre,real,annee,genre) (on se permettra souvent de ne pas préciser les types de données dans un schéma de relation) :

Film

1	Titanic	Cameron	1997	film comique
2	Avatar	Cameron	2009	film d'horreur
3	Blanche-Neige	Disney	1937	film expérimental
4	Snatch	Ritchie	2000	western

Plutôt que de donner des dizaines de définitions floues supplémentaires, contentons-nous maintenant d'expliquer et d'illustrer les quelques techniques permettant de passer d'un schéma Entité/Association d'une base de données à un schéma relationnel cohérent et solide :

- chaque entité devient une relation, les attributs restant identiques, et la clé primaire étant en général le premier attribut de la relation (si on donne le schéma de la relation, on indiquera toujours la clé primaire en gras).
- une association de cardinalité 1..1 1..n ne donnera pas lieu à une nouvelle relation, mais on imposera à la clé primaire de la première entité d'apparaitre comme attribut de la seconde. Ainsi, l'association entre un film et son réalisateur imposera d'indiquer la clé primaire du réalisateur (typiquement un numéro d'identifiant) dans la table Film (notre table précédente est donc incorrecte, le nom du réalisateur ne devrait pas apparaître, car ce n'est pas un bon choix de clé primaire, son numéro d'identification servira de toute façon de lien vers la table des réalisateurs où toutes les informations le concernant seront disponibles).
- dans le cas particulier d'une entité faible, la clé primaire de la relation correspondant à l'entité faible sera le couple constitué de la clé de l'entité forte et de celle de l'entité faible (par exemple (**n°cine**,**n°salle**) pour reprendre notre exemple précédent, on aura alors dans la table Salle deux premières colonnes donnant le numéro du cinéma et celui de la salle, ce qui semble assez logique).
- dans le cas d'associations plus complexes (de cardinalité 1..n 1..n notamment), on créera tout simplement une relation pour l'ssociation, intégrant en guise de clé primaire toutes les clés

primaires des entités qu'elle relie. Donnons tout de suite un exemple où, en plus de notre relation film déjà décrite plus haut (mais modifiée pour remplacer le nom du réalisateur par un numéro d'identifiant), on ajoute une relation Réalisateurs(**n°real**,nom,prenom,nationalite), une relation Acteurs(**n°acteur**,nom,prenom,date de naissance), et une relation correspondant à une association, la relation Rôle(**n°id**,**n°acteur**,role) reliant un acteur à un film dans lequel il joue en précisant le rôle joué.

Film

n°id	titre	n°real	annee	genre
1	Titanic	1	1997	film comique
2	Avatar	1	2009	film d'horreur
3	Blanche-Neige	2	1937	film expérimental
4	Snatch	3	2000	western

Réalisateurs

n°real	nom	prenom	nationalite
1	Cameron	James	américain
2	Disney	Walt	américain
3	Ritchie	Guy	grand-breton

Acteurs

n°acteur	nom	prenom	date de naissance
1	Di Caprio	Leonardo	11/11/1974
2	Winslet	Kate	05/10/1975
3	Mathy	Mimie	08/07/1957
4	Poil	Rou	28/04/1981

Rôle

n°id	n°acteur	role
1	1	mauvais nageur
1	2	croqueuse de diamants
3	3	Grincheux
3	4	Prof

2.2 Algèbre relationnelle et requêtes

Maintenant que nous savons construire une base de données bien organisée, il reste à apprendre à l'interroger pour y effectuer des recherches (ce qu'on appellera des **requêtes** dans le jargon des bases de données). Nous ferons cet apprentissage en deux temps, en présentant d'abord un langage mathématique pour présenter ces requêtes, puis un langage informatique. Allons-y pour les maths (je vous rassure, même si le terme algèbre est présente dans le titre de ca paragraphe, le plus compliqué dans ce que nous ferons, ce sera les notations un peu étranges).

Définition 7. L'**algèbre relationnelle** est un ensemble d'opérations possibles sur les relations d'une base de données, visant à répondre à des requêtes effectuées sur la base.

Définition 8. La **sélection** d'une relation R suivant la condition F extrait de la table R les lignes vérifiant la condition F. Elle est notée de façon abrégée $\sigma_F(R)$.

Exemple : En reprenant notre mini-base de données cinématographique créée à la section précédente, l'opération $\sigma_{n^\circ real=1}(Film)$ va nous renvoyer la table suivante :

1	Titanic	1	1997	film comique
2	Avatar	1	2009	film d'horreur

Définition 9. La **projection** de la relation R suivant les attributs A_1, ..., A_i extrait de la table R les colonnes correspondants aux attributs spécifiés. On la note $\Pi_{A_1,...,A_i}(R)$.

Exemple : L'opération $\Pi_{titre,genre}(Film)$ va nous donner la table suivante :

Titanic	film comique
Avatar	film d'horreur
Blanche-Neige	film expérimental
Snatch	western

Ces deux opérations sont les plus naturelles mais sont insuffisantes pour des recherches plus complexes sur une base de données, ce qu'on appelle les **recherches croisées** (recherches faisant intervenir des informations issue de plusieurs tables). Si on veut par exemple simplement afficher la liste des titres de films avec le nom et le prénom de leur réalisateur, une sélection ou une projection ne peut pas suffire.

Définition 10. Le **produit cartésien** des relations R_1 et R_2 est constitué de lignes de la forme $x \times y$, où x parcourt R_1 et y parcourt R_2. On le note $R_1 \times R_2$.

Exemple : Si on effectue le produit de nos relations Film et Réalisateurs, on va obtenir le passionnant résultat suivant :

n°id	titre	F.n°real	annee	genre	R.n°real	nom	prenom	nationalite
1	Titanic	1	1997	film comique	1	Cameron	James	américain
1	Titanic	1	1997	film comique	2	Disney	Walt	américain
1	Titanic	1	1997	film comique	3	Ritchie	Guy	grand-breton
2	Avatar	1	2009	film d'horreur	1	Cameron	James	américain
2	Avatar	1	2009	film d'horreur	2	Disney	Walt	américain
2	Avatar	1	2009	film d'horreur	3	Ritchie	Guy	grand-breton
3	Blanche-Neige	2	1937	film expérimental	1	Cameron	James	américain
3	Blanche-Neige	2	1937	film expérimental	2	Disney	Walt	américain
3	Blanche-Neige	2	1937	film expérimental	3	Ritchie	Guy	grand-breton
4	Snatch	3	2000	western	1	Cameron	James	américain
4	Snatch	3	2000	western	2	Disney	Walt	américain
4	Snatch	3	2000	western	3	Ritchie	Guy	grand-breton

Une première remarque : il y avait un **conflit d'attributs** entre nos deux tables (un même nom d'attributs est utilisé dans les deux tables, et pour cause puisqu'il représente la même donnée !), ce qui force pour faire le produit à renommer les attributs correspondants pour préciser leur provenance. Par défaut, on les appelle R_1.attribut et R_2.attribut, en abrégeant le nom des relations à leurs initiales (comme on l'a fait ici) si cela suffit à lever l'ambiguïté. Certes, mais le principal problème du produit, c'est tout simplement qu'il ne sert à rien ! Du moins en tant que tel, car il présente quand même l'intérêt de regrouper les données de deux tables (il effectue de fait un produit cartésien ordinaire au sens mathématique du terme). Pour le rendre intéressant, il suffit en fait de le composer avec une opération de sélection. On peut composer (au sens mathématique encore une fois) deux opérations quelconques, mais le type décrit juste avant est tellement fréquent qu'on lui donne un nom spécifique.

Définition 11. Une **jointure** de deux relations est la composée de leur produit cartésien par une sélection. On note la jointure de R_1 et de R_2 avec la condition F sous la forme abrégée $R_1 \bowtie_F R_2$.

Exemple : Rien n'empêche évidemment de composer encore une jointure, par exemple par une projection (cas très courant également). Dans notre base de données, l'opération $\Pi_{titre,nom,prenom}(Film \bowtie_{F.n^\circ real=R.n^\circ real} Ralisateurs)$ va nous afficher ceci :

Titanic	Cameron	James
Avatar	Cameron	James
Blanche-Neige	Disney	Walt
Snatch	Ritchie	Guy

Définition 12. Si deux relations R_1 et R_2 ont le même schéma de relation, on peut effectuer deux opérations supplémentaires :

- l'**union** $R_1 \cup R_2$ qui regroupe les lignes de R_1 et R_2 non identiques en une seule relation.
- la **différence** $R_1 - R_2$ qui sélectionne les lignes de R_1 n'apparaissant pas dans R_2.

Avec les opérations décrites dans ce paragraphe, on peut théoriquement effectuer toutes les recherches imaginables dans une base de données.

3 Quelques éléments de SQL

L'algèbre relationnelle est un outil théorique très puissant, mais une fois installé devant nos chers ordinateurs, il est peu probable qu'on nous demande de rédiger nos requêtes dans ce langage. J'ai donc une mauvaise nouvelle à vous annoncer : il va falloir apprendre les rudiments d'un nouveau langage informatique. En contrepartie, deux bonnes nouvelles pour le prix d'une : la première, c'est que le langage en question est nettement plus simple à appréhender q'un langage de programmation comme Python ; la seconde, c'est qu'à peu près toutes les bases de données du monde utilisent un langage de requêtes commun, le **SQL** (**S**tructured **Q**uery **L**angage), qui vous servira donc toute votre vie une fois que vous en aurez compris les bases. Une dernière remarque avant de se plonger dans le vif du sujet : la plupart des gestionnaires de bases de données modernes (et même moins modernes, par exemple celui que vous manipulerez en TP) disposent d'interfaces graphiques simplifiées pour la gestion de la base (en gros des menus cliquables), mais l'objectif de ce cours est de vous apprendre à faire les choses « correctement » sans raccourcis (de toute façon, l'interface en question restranscrira chacune de vos demandes en langage SQL avant de l'envoyer à la base).

3.1 Types en SQL

Un point commun entre SQL et les langages de programmation est la présence de types de données, permettant tout simplement de préciser le format que va contenir chaque « case » dans nos tables de données (ce qui correspond au domaine des attributs dans le schéma de la relation). Il existe quantité de types distincts en SQL, mais connaître les principaux est largement suffisant (notons que SQL ne contient que des types simples, une « variable » dans une base ne peut contenir qu'une seule et unique donnée).

Liste non exhaustive de types de données en SQL.

- INTEGER : nombre entier.
- FLOAT : nombre décimal.
- VARCHAR(M) : chaîne de caractère de longueur maximale M (à préciser obligatoirement).
- DATE : date (au format anglo-saxon aaaa-mm-jj).
- YEAR : année (cas particulier d'INTEGER).

Notons en passant que, suivant les conventions courantes, nous noterons en lettres capitales tous les mots réservés du langage SQL dans tous els exemples du cours, le reste étant en minuscules.

3.2 Création d'une table

On ne peut pas dire que les noms de commandes soient particulièrement difficiles à comprendre en SQL, comme le prouve notre exemple, puisque la commande de création de table est CREATE TABLE. Il faut bien évidemment donner quelques précisions, qui correspondent simplement au schéma de la relation qu'on est en train de créer : le nom de la table, et, entre parenthèses, la liste des attributs avec leur domaine séparés par des virgules.
Exemple : On reprend notre éternel exemple de base de donnée cinématographique. On créera par exemple une table correspondant à notre relation Film via la commande :

```
CREATE TABLE Film   (n°id INTEGER,
                     titre VARCHAR(100),
                     n°real INTEGER,
                     annee INTEGER,
                     genre VARCHAR(20))
```

Il existe énormément de possibilités de moduler ce genre de définition pour y ajouter des précisions. Sans chercher le moins du monde à être complet (le but n'est pas de vous transformer en experts du SQL, mais simplement que vous sachiez écrire correctement les requêtes les plus fréquentes, et soyez capables de créer sans trop de difficultés votre propre base de données), en voici quelques-unes :

- On peut imposer à un des attributs de prendre une valeur en ajoutant NOT NULL après le type de données (si on ne le fait pas, la case correspondante pourra rester vide dans la base de données).
- On peut également imposer une valeur par défaut égale à « machin » via l'ajout DEFAULT 'machin' (s'il s'agit d'une chaîne de caractères).
- On peut préciser qu'un attribut est une clé primaire soit en mettant une option (comme ci-dessus, après le type) PRIMARY KEY derrière l'attribut, soit en ajoutant une dernière ligne de la forme PRIMARY KEY (attribut), ce qui sera notamment utile si la clé primaire est constituée de deux attributs ou plus.
- Sans qu'il s'agisse d'une clé primaire, on peut imposer l'unicité des valeurs prises par un attribut par l'option UNIQUE (même fonctionnement que PRIMARY KEY).
- On peut imposer des conditions sur les valeurs prises par un attribut via la commande CHECK. Par exemple, dans notre table Film, on pourra préciser sur l'avant-dernière ligne : annee INTEGER CHECK (annee BETWEEN 1895 AND 2020) pour éviter les saisies de valeurs aberrantes. De même on peut préciser sur la dernière ligne genre VARCHAR(20) CHECK (genre in ('comedie','film de zombie','dessin anime')) si on souhaite imposer une liste de genres dès le départ (utile si plusieurs personnes gèrent les mises à jour pour avoir une cohérence des dénominations).

Enfin, *last but not least*, il existe un concept essentiel dont je ne vous ai pas encore parlé dans ce cours, qui va nous permettre de créer des liens effectifs entre nos tables et d'imposer des contraintes d'intégrité, celui de **clé secondaire**. Il s'agit tout simplement d'un attribut d'une table qui correspond à la clé primaire d'une autre table (ce qui arrivera à chaque fois qu'on restranscrit dans un schéma relationnel une association du modèle initial Entité/Association). On indique ce genre de chose par la commande :

FOREIGN KEY (attribut) REFERENCES table

à l'intérieur de la création de la table où notre attribut n'est pas clé primaire (et la table indiquée après le REFERENCES est donc celle où il est clé primaire). On peut ajouter des précisions concernant cette clé étrangère, et notamment ce qui se passe si on effectue une mise-à-jour dans notre table (doit-on la répercuter dans l'autre ?). Les options les plus fréquemment utilisées seront ON DELETE SET NULL (si on détruit une donnée, on garde les données qui lui sont reliées, mais en « vidant » la case correspondant à l'information supprimée) et ON DELETE CASCADE (qui supprime en

cascade toutes les données reliées à une donnée supprimée). Par exemple, avec nos tables Film et Réalisateurs, on aura à jouter une ligne dans la création de la table Film, qui ressemblera plutôt à ceci après tous les ajouts effectués :

```
CREATE TABLE Film   (n°id INTEGER PRIMARY KEY,
                    titre VARCHAR(100) NOT NULL,
                    n°real INTEGER,
                    annee INTEGER
                        CHECK (annee BETWEEN 1895 AND 2020),
                    genre VARCHAR(20)
                        CHECK genre in ('comedie','film de zombie','dessin anime')
                    FOREIGN KEY (n°real) REFERENCES Réalisateurs
                        ON DELETE SET NULL)
```

3.3 Mises à jour

Une fois les tables de notre base de données créées, il reste encore à les remplir, puis à mettre à jour les données si besoin. Plutôt qu'un long discours théorique, nous nous contenterons de données quelques exemples faisant apparaitre les mots-clés les plus importants du langage SQL dans ce domaine, la syntaxe étant de toute façon très naturelle.

- on peut modifier le schéma d'une table à l'aide de la commande ALTER TABLE, ainsi que des commandes ADD (pour ajouter une colonne), DROP (destruction de colonne), MODIFY (pour changer les options de l'attribut, y compris son type, ce qui est toutefois dangereux si la table contient déjà des données). Ainsi, pour ajouter un attribut nationalité à notre table Réalisateur, on pourra taper la commande :

  ALTER TABLE Réalisateurs ADD nationalite VARCHAR(20)

- pour insérer des données dans une table, opération ô combien courante, on utilisera la commande INSERT INTO suivie du mot-clé VALUES. Ainsi pour remplir la première ligne de notre table Film :

  INSERT INTO Film VALUES (1,'Titanic',1,1997,'comedie')

  Si on ne souhaite donner de valeurs qu'à certains attributs, on précise les attributs entre parenthèses derrière le nom de la table (le reste aura la valeur NULL).
- pour détruire des données dans une table, on utilise la syntaxe DELETE FROM table WHERE condition (une condition étant en général une égalité, on reviendra dessus en décrivant les sélections un peu plus bas).
- pour mettre à jour des données existantes, on dispose de la commande UPDATE table SET maj WHERE condition. Il n'est pas inspensable de préciser une condition si on veut que toute la table soit affectée, et maj indique la mise-à-jour, en général une modification d'un ou plusieurs attributs. Par exemple, pour remplacer la nationalité « américain » par « états-unien » pour tous les réalisateurs concernés, on effectuera :

  UPDATE Réalisateurs SET nationalite='états-unien' WHERE nationalite='américain'

3.4 Opérations

Il ne reste plus qu'à décrire le plus important, la traduction des opérations de l'algèbre relationnelle dans le langage SQL. Le mot-clé qui sera, de très loin, le plus utilisé, est SELECT, qui servira à la fois à effectuer des sélections (au sens de l'algèbre relationnelle) et des projections (attention à ne pas se tromper). La syntaxe de base est la suivante : SELECT attribut1, attribut2 FROM table WHERE condition. Le liste des attributs n'a bien sûr aucune raison dêtre limitée à deux, et on peut par ailleurs effectuer dans un SELECT des opérations sur les attributs sélectionnés (ce qui peut

notamment avoir un sens en cas d'attributs numériques). Le choix des attributs correspond donc à une projection, alors que la condition située après le WHERE correspond à une sélection. Si on souhaite effectuer une simple projection, on supprime purement et simplement le WHERE. Si on souhaite au contraire effectuer une simple sélection, on remplace les attributs par *, ce qui revient à prendre tous les attributs. Par exemple :

SELECT * FROM Film WHERE annee BETWEEN 1950 AND 1959

affichera toutes les lignes de la table Film correspondant aux films des années 50. Si on ne souhaite que les titres de ces films, on remplace l'étoile par Titre. Si on souhaite trier l'affichage selon les valeurs prises par un attribut, on ajoutera une option ORDER BY attribut. Si on souhaite imposer plusieurs conditions, on les sépare simplement par le mot-clé AND.

Remarque 4. Dans le cas d'attributs prenant comme valeurs des chaînes de caractères, on peut rajouter aux classiques égalités des conditions plus intéressante comme « la chaîne commence par 'la' », ce qui s'écrira WHERE attribut LIKE 'la%' (SQL considère % comme pouvant être remplacé par n'importe quelle chaîne de caractères). Si on souhaite imposer en plus que la chaîne de caractères contient cinq caractères, on écrira WHERE attribut LIKE 'la___' (chaque _ peut être remplacé par un caractère quelconque, mais pas plusieurs).

Pour effectuer des jointures, rien de plus simple puisqu'on utilise encore et toujours la commande SELECT ! Il suffit en effet d'indiquer plusieurs tables derrière le FROM pour effectuer une requête croisée, la condition après le WHERE faisant en général intervenir des attributs issus des différentes tables. S'il y a un conflit d'attribut, on utilisera la syntaxe plus lourde table.attribut pour désigner les attributs posant problème. Si on veut raccourcir la syntaxe, on indique un nom alternatif (par exemple l'initiale) pour la table après le FROM. Je suis d'accord, un exemple sera plus simple :

SELECT titre, nom, prenom FROM Film, Réalisateurs WHERE Film.n°real=Réalisateurs.n°real

va afficher la liste des films de la base de données avec le nom et le prénom de leur réalisateur (la syntaxe SQL est finalement très proche de ce qu'on avait dans l'algèbre relationnelle pour les jointures). En un peu plus court,

SELECT titre, nom, prenom FROM Film F, Réalisateurs R WHERE F.n°real=R.n°real

effectue exactement la même chose (on a simplement renommé Film en F et Réalisateurs en R en passant). Techniquement, on peut d'ailleurs effectuer des requêtes croisées faisant intervenir deux fois la même table (pour obtenir des couples d'éléments d'une table), en la renommant de deux façons différentes. Ainsi, si on souhaite afficher une liste des couples de films ayant le même réalisateur (pourquoi pas !), on pourra taper la commande :

SELECT F1.titre, F2.titre FROM Film F1, Film F2 WHERE F1.n°real=F2.n°real

Les autres opérations de l'algèbre relationnelle (union, intersection, différence) se font tout simplement à l'aide des commandes UNION, INTERSECT et EXCEPT. Il faut bien sûr les appliquer à des tables ayant le même schéma pour que cela ait un sens (non, pas d'exemples pour ça, on ne s'en servira que rarement). Il est par ailleurs tout à fait possible de combiner des conditions ou même de mettre des SELECT à l'intérieur d'un WHERE, mais nous nous contenterons de cas relativement sommaires pour ce cours. À vous de réfléchir au cas pas cas aux possibilités offertes par les quelques commandes que je vous ai présentées.

Tout de même, un mot avant de conclure sur les **fonctions d'agrégation**, qui permettent de ressortir à l'issue d'une recherche non pas une table, mais le résultat d'une opération effectuée sur l'ensemble des valeurs de la table (en général limitée à une colonne). On appliquera en général ces fonctions à des attributs prenant des valeurs numériques (sous peine d'obtenir des résultats peu interprétables), et les principales fonctions disponibles sont COUNT (qui compte simplement le nombre de lignes), AVG (moyenne des valeurs prises par l'attribut), MAX, MIN (j'ai besoin d'expliquer pour ces deux-là ?) et SUM (somme des valeurs ; les plus réveillés d'entre vous constateront l'inutilité profonde de la commande AVG qui peut toujours être obtenue en divisant SUM par COUNT). Ainsi :

SELECT COUNT(titre) FROM Film WHERE n°real=1

va simplement afficher le nombre de films stockés dans la base de données dont le réalisateur porte le numéro 1 (james Cameron dans notre exemple). Ou encore :

SELECT titre, MAX(annee) FROM Film WHERE n°real=1

va faire ... une erreur SQL ! Si vous pensiez que cette requête vous sortirait le titre et l'année du film le plus récent du père Cameron dans la table, vous vous trompez, car il y a une incohérence dans la requête entre le SELECT titre (qui ressort une table) et le SELECT MAX(annee) (qui ne sort qu'une valeur). Il faudra donc se débrouiller autrement.