Cours de bases de données - Modèles et langages

Version 1.0

Philippe Rigaux

mars 16, 2017

Table des matières

Contents : Le document que vous commencez à lire fait partie de l'ensemble des supports d'apprentissage proposés sur le site http://www.bdpedia.fr. Il fait partie du cours consacré aux bases de données relationnelles, divisé en deux parties :

- La version en ligne du support "Modèles et langages" est accessible à http://sql.bdpedia.fr, la version imprimable (PDF) est disponible à http://sql.bdpedia.fr/files/cbd-sql.pdf.
- La version en ligne du support "Aspects système" est accessible à http://sys.bdpedia.fr, la version imprimable (PDF) est disponible à http://sys.bdpedia.fr/files/cbd-sys.pdf.

Deux autres cours, aux contenus proches, sont également disponibles :

- Un cours sur les bases de données documentaires et distribuées à http://b3d.bdpedia.fr.
- Un cours sur les applications avec bases de données à http://orm.bdpedia.fr

Reportez-vous à http://www.bdpedia.fr pour plus d'explications.

CHAPITRE 1

Introduction

Ce support de cours s'adresse aux étudiants en premier cyle (IUT) ou second cycle universitaire (Licence, Master) et formations apparentées. Il propose un ensemble de chapitres consacrés aux principes et à la mise en œuvre de bases de données relationnelles, ainsi qu'à la pratique des Systèmes de Gestion de Bases de Données (SGBD). Il couvre plus particulièrement les *modèles et langages* des bases de données. Cette partie couvre tout d'abord la conception et la définition d'un schéma relationnel correct et complet, comprenant des tables, des contraintes, des vues, etc. Elle décrit ensuite l'algèbre relationnelle et SQL, ainsi que l'intégration de SQL avec un langage de programmation comme le C.

Une seconde partie, consacrée aux *aspects systèmes* présente les techniques internes utilisées par les SGBD relationnels pour stocker efficacement les données et évaluer des requêtes. Elle couvre la représentation physique, l'indexation, l'optimisation et comprend également une introduction aux problèmes de concurrence d'accès, dont la connaissance est nécessaire aux développeurs d'applications basées sur des SGBD. Cette seconde partie est disponible séparément sur le site le site http://sys.bdpedia.fr.

La première partie est accessible aux étudiants suivant un cours d'introduction aux bases de données et ne demande que peu de pré-requis. La second partie est plus avancée et nécessite de bonnes bases en structures de données et algorithmique.

Le premier chapitre est une (rapide) présentation de tous les thèmes présentés en détails dans ce cours. On peut le lire comme une mise en perspective générale de l'ensemble du document.

CHAPITRE 2

Conception d'une base de données

Ce chapitre est consacré la démarche de *conception* d'une base relationnelle. Elle se déroule en général selon un processus de conception comportant plusieurs étapes. Nous nous concentrons dans ce chapitre sur la modélisation, et sur la représentation de cette modélisation avec une notation très répandue, dite *entité / association*. Au préalable, nous allons discuter de l'objectif final, qui est d'obtenir un *schéma* de la base de données conforme au besoin et ne présentant par d'anomalie.

2.1 Schéma d'une base de données

Le schéma d'une base relationnelle est constitué d'un ensemble de schémas de tables. Le schéma d'une table à son tour consiste – pour l'essentiel – en un nom (de table) et un ensemble de noms d'attributs.

La conception d'un schéma relationnel ne présente pas de difficulté technique. On constate en pratique qu'elle demande une certaine expérience, notamment parce que les conseils d'usage, assez abstraits, deviennent beaucoup plus clairs quand on s'est trompé une ou deux fois et qu'on a constaté les conséquences de ses erreurs. Le schéma de données constitue en effet les fondations d'une application Le schéma n'est jamais figé : il évoluera avec l'application. Mais la partie existante d'un schéma est toujours difficile à remettre en cause sans entraîner une réécriture non négligeable.

Dans ce qui suit, nous illustrons, à partir d'exemples, les obstacles à éviter dans la conception d'un schéma, l'objectif à atteindre et les méthodes disponibles.

2.1.1 Qualité d'un schéma relationnel

Voici un exemple de schéma, avec une notation très simplifiée, que nous allons utiliser pour discuter de la notion centrale de "bon" et "mauvais" schéma. On veut créer une base de données représentant des films, avec des informations comme le titre, l'année, le metteur en scène, etc. On part d'un schéma rassemblant ces informations ("MES" est l'acronyme de "Metteur en Scène") :

```
Film(titre, annee, prenomMes, nomMES, anneeNaiss)
```

Un tel schéma permet de représenter les données de manière complète, en en évitant autant que possible les anomalies ? Regardons un exemple de contenu de la table décrite par ce schéma.

titre	année	nomMES	prénomMES	annéeNais
Alien	1979	Scott	Ridley	1943
Vertigo	1958	Hitchcock	Alfred	1899
Psychose	1960	Hitchcock	Alfred	1899
Kagemusha	1980	Kurosawa	Akira	1910
Volte-face	1997	Woo	John	1946
Pulp Fiction	1995	Tarantino	Quentin	1963
Titanic	1997	Cameron	James	1954
Sacrifice	1986	Tarkovski	Andrei	1932

Même pour une information aussi simple, il est facile d'énumérer tout un ensemble de problèmes potentiels. Tous ou presque découlent d'un grave défaut de la table ci-dessus : *il est possible de représenter la même information plusieurs fois*, ou, pour employer un mot que nous retrouverons souvent, *il y a redondance de l'information*.

Anomalies lors d'une insertion

Rien n'empêche de représenter plusieurs fois le même film. Pire : il est possible d'insérer plusieurs fois le film *Vertigo* en le décrivant à chaque fois de manière différente, par exemple en lui attribuant une fois comme réalisateur Alfred Hitchcock, puis une autre fois John Woo, etc.

La bonne question consiste d'ailleurs à se demander ce qui distingue deux films l'un de l'autre, et à quel moment on peut dire que la même information a été répétée. Peut-il y avoir deux films différents avec le même titre par exemple ? Si la réponse est non (?), alors on devrait pouvoir assurer qu'il n'y a pas deux lignes dans la table avec la même valeur pour l'attribut *titre*. Si la réponse est oui (ce qui semble raisonnable), il reste à déterminer quel est l'ensemble des attributs qui permet de caractériser de manière unique un film ou, à défaut, de créer un tel *identifiant* artificiellement. C'est une notion centrale sur laquelle nous revenons de manière approfondie.

Anomalies lors d'une modification

La redondance d'information entraîne également des anomalies de mise à jour. Supposons que l'on modifie l'année de naissance de Hitchcock pour la ligne *Vertigo* et pas pour la ligne *Psychose*. On se retrouve alors avec des informations incohérentes. Les mêmes questions que précédemment se posent d'ailleurs. Jusqu'à quel point peut-on dire qu'il n'y a qu'un seul réalisateur nommé Hitchcock, et qu'il ne doit donc y avoir qu'une seule année de naissance pour un réalisateur de ce nom ?

Anomalies lors d'une destruction

On ne peut pas supprimer un film sans supprimer du même coup son metteur en scène. Si on souhaite, par exemple, ne plus voir le film *Titanic* figurer dans la base de données, on va effacer du même coup les informations sur James Cameron.

2.1.2 Schémas normalisés

Que déduire de ce qui précède ? Tout d'abord qu'il existe des schémas avec de bonnes propriétés, et d'autres qui souffrent de défauts de conception. Ensuite, que nous avons besoin d'aller plus loin qu'une simple énumération d'attributs et énoncer des *contraintes* et des *règles* qui nous indiquent plus précisément les liens qui caractérisent les données.

Le modèle relationnel nous propose un outil précieux pour répondre à ces questions : la *normalisation*. Un schéma normalisé présente des caractéristiques formelles qu'il est possible d'évaluer. La normalisation nous garantit l'absence de défaut (et notamment de redondance) tout en préservant l'intégralité de l'information représentée.

La théorie du modèle relationnel a développé une construction mathématique solide pour qualifier les propriétés d'un schéma d'une part, et décomposer un schéma dénormalisé en schéma normalisé d'autre part. Le second aspect s'appuie malheureusement sur une démarche peu utilisable en pratique pour des raisons sur lesquelles nous reviendront plus loin. Le premier en revanche donne un éclairage très précis sur ce qu'est un bon schéma relationnel. Il est intéressant de consacrer quelques instants (voire un peu plus si vous voulez vous donner tous les moyens d'éviter des ennuis ultérieurs) à la définition des schémas normalisés.

2.1.3 La notion de dépendance fonctionnelle

Le principal concept est celui de dépendance fonctionnelle, qui fournit une construction de base pour élaborer les contraintes dont nous avons besoin pour caractériser nos données et leurs liens. Il s'énonce comme suit. Prenons un schéma de relation *R*, *S* un *sous-ensemble* d'attributs de *R*, et *A* un attribut quelconque de *R*.

Définition : dépendance fonctionnelle

On dit que *A dépend fonctionnellement* de *S* (ce que l'on note $S \rightarrow A$) quand, pour toute paire (l_1, l_2) de lignes de *R*, l'égalité de l_1 et de l_2 sur *S* implique l'égalité sur *A*.

Informellement, on peut raisonner ainsi : "Si je connais *S*, alors je connais *A*". Par, exemple, si je prends l'ensemble des attributs suivants :

```
(nom, prenom, noSS, dateNaissance, adresse, email)
```

je peux chercher les dépendances fonctionnelles et trouver celles-ci :

— $email \rightarrow nom, prenom, noSS, dateNaissance, adresse$
— $noSS \rightarrow email, nom, prenom, dateNaissance, adresse$

J'ai donc reconnu que la connaisance d'un email détermine la connaissance des autres attributs, et de même pour le numéro de sécurité sociale. On peut avoir des dépendances fonctionnelles où la partie gauche comprend plusieurs attributs. Par exemple, pour les attributs suivants :

```
noEtudiant, noCours, annee, note
```

on trouvera la dépendance fonctionnelle suivante :

— $noEtudiant, noCours, annee \rightarrow note$

La connaissance d'un étudiant, d'un cours et d'une année détermine la note obtenue.

Les dépendances fonctionnelles fournissent un outil pour analyser la qualité d'un schéma relationnel. Prenons le cas d'un système permettant d'évaluer des manuscrits soumis à un éditeur. Voici deux schémas possibles pour représenter les rapports d'évaluation d'un manuscrit (on va supposer qu'il existe un seul expert par manuscrit).

— **Schéma 1**
  — Manuscrit (id_manuscrit, titre, id_expert, nom, adresse, commentaire)}
— **Schéma 2**
  — Manuscrit (id_manuscrit, titre, id_expert, commentaire)
  — Expert (id_expert, nom, adresse)

Il est très instructif de réfléchir à ces deux schémas, à leurs différences et à leurs avantages et inconvénients respectifs.

Si l'on raisonne en termes de dépendance fonctionnelle (que nous noterons DF à partir de maintenant), on constate que dans le schéma de la relation *Expert(id_expert, adresse)*, il existe une dépendance fonctionnelle entre `id_expert` et `adresse`. En d'autre termes, si on trouve dans la base deux lignes avec l'identifiant d'expert 128, l'adresse pour ces deux lignes doit être la même. Une DF énonce donc une *contrainte* sur le contenu de la base de données : la préservation des DFs garantit l'absence d'incohérence (le même expert, mais deux adresses différentes et donc contradictoires).

La question suivante est bien sûr : pourquoi devrait-on représenter plusieurs fois la même information et s'imposer la charge de vérifier la cohérence ? Là encore, en nous appuyant sur les dépendances fonctionnelles, nous allons pouvoir définir la notion essentielle de *clé* qui nous permettra d'éviter les redondances.

Définition : clé

Une clé (d'une relation *R*) est un sous-ensemble *minimal C* des attributs tel que tout attribut de *R* dépend fonctionnellement de *C*.

L'attribut `id_expert` est une clé de la relation *Expert* dans le schéma 2. Dans le schéma 1, l'attribut `id_manuscrit` est une clé de *Manuscrit*. Notez que tout attribut de la relation dépend *aussi* de la paire `(id_manuscrit,id_expert)`, sans que cette paire soit une clé puisqu'elle n'est pas *minimale* (il existe un sous-ensemble strict qui est lui-même clé).

Et maintenant nous pouvons définir ce qu'est un schéma de relation normalisé.

Définition : schéma normalisé

Un schéma de relation *R* est *normalisé* quand, dans toute dépendance fonctionnelle $S \rightarrow A$ sur les attributs de *R*, *S* est une clé.

Considérons à nouveau notre schéma 1 :

```
Manuscrit (id_manuscrit, id_expert,  nom, commentaire, adresse_expert)
```

Il n'est pas normalisé car il existe une dépendance fonctionelle `id_expert` $\rightarrow$ `nom,adresse_expert`, alors que l'attribut `id_expert` n'est pas une clé. Il existe une version intuitive de cette constatation abstraite : la table `Manuscrit` contient des informations qui ne sont pas *directement* liées à la notion de manuscrit. La présence d'informations indirectes est une source de redondance et donc d'anomalies.

L'essentiel de ce qu'il faut comprendre est énoncé (certes de manière *très* condensée) dans ce qui précède. On veut obtenir des relations normalisées car il et facile de montrer que la dénormalisation entraîne toutes sortes d'anomalies au moment où la base est mise à jour. De plus, si *R* est une relation de clé *C*, deux lignes de *R* ayant les même valeurs pour *C* auront par définition les mêmes valeurs pour les autres attributs et seront donc parfaitement identiques. Il est donc inutile (et nuisible) d'autoriser cette situation : on fera en sorte que la valeur d'une clé soit *unique* pour l'ensemble des lignes d'une relation. En résumé on veut des schémas de relation normalisés et dotés d'une clé unique bien identifiée. Cette combinaison interdit toute redondance.

Note : Les théoriciens qui ont étudié ces notions ont défini plusieurs formes de normalisation. Celle présentée ici est dite "troisième forme normale" (3FN).

2.1.4 La décomposition d'un schéma

Etant donné un schéma et ses dépendances fonctionnelles, nous savons déterminer s'il est normalisé. Peut-on aller plus loin et produire *automatiquement* un schéma normalisé à partir de l'ensemble des attributs et de leurs contraintes (les DFs) ? En principe oui, en pratique pas vraiment.

Regardons d'abord le principe avec un exemple illustrant la normalisation d'un schéma relationnel par un processus de décomposition progressif. On veut représenter l'organisation d'un ensemble d'immeubles locatifs en appartements, et décrire les informations relatives aux propriétaires des immeubles et aux occupants de chaque appartement. Voici un premier schéma de relation :

```
Appart(id_appart, surface, id_immeuble, nb_etages, date_construction)
```

La clé est `id_appart`. Cette relation est-elle normalisée ? Non, car l'identifiant de l'immeuble détermine fonctionnellement le nombre d'étages et la date de construction. On a donc une dépendance $id\_immeuble \rightarrow nb\_tages, date\_construction$ dont la partie gauche n'est pas la clé `id_appart`. En pratique, une telle relation dupliquerait le nombre d'étages et la date de construction autant de fois qu'il y a d'appartements dans un immeuble. Le bon schéma en l'occurrence serait la décomposition en deux relations :

```
Appart(id_appart, surface, id_immeuble)
Immeuble (id_immeuble, nb_etages, date_construction)
```

Supposons qu'un immeuble puisse être détenu par plusieurs propriétaires, et considérons la seconde relation suivante, :

```
Proprietaire(id_appart, id_personne, quote_part)
```

Est-elle normalisée ? Oui car l'unique dépendance fonctionnelle est

$$id\_appart, id\_personne \rightarrow quote\_part$$

Un peu de réflexion suffit à se convaincre que ni l'appartement, ni le propriétaire ne déterminent à eux seuls la quote-part. Seule l'association des deux permet de donner un sens à cette information, et la clé est donc le couple `(id_appart,id_personne)`. Maintenant considérons l'ajout du nom et du prénom du propriétaire dans la relation.

```
Proprietaire(id_appart, id_personne, prenom, nom, quote_part)
```

La dépendance fonctionnelle $id\_personne \rightarrow prnom, nom$ indique que cette relation n'est pas normalisée. Le bon schéma est :

```
Proprietaire(id_appart, id_personne, quote_part)
Personne(id_personne, prenom, nom)
```

Si, en revanche, on décide qu'il ne peut y avoir qu'un propriétaire pour un appartement et inversement, la quote-part devient inutile, une nouvelle dépendance fonctionnelle $id\_personne \rightarrow id\_appart$ apparaît, et la relation avant décomposition est bien normalisée, avec pour clé `id_personne`.

Voyons pour finir le cas des occupants d'un appartement, avec la relation suivante.

```
Occupant(id_personne, nom, prenom, id_appart, surface)
```

On mélange clairement des informations sur les personnes, et d'autres sur les appartements. Plus précisément, la clé est le couple `(id_personne,id_appart)`, mais on a les dépendances suivantes :

- $id\_personne \rightarrow prnom, nom$
- $id\_appart \rightarrow surface$

Un premier réflexe pourrait être de décomposer en deux relations `Personne(id_personne,prénom,nom)` et `Appart (id_appart,surface)`. Toutes deux sont normalisées, mais on perd alors une information importante, et même essentielle : le fait que telle personne occupe tel appartement. Il faut donc impérativement conserver une relation correspondant à la clé initiale :`Occupant (id_personne,id_appart)`. D'où le schéma final :

```
Immeuble (id_immeuble, nb_etages, date_construction)
Proprietaire(id_appart, id_personne, quote_part)
Personne (id_personne, prenom, nom)
Appart (id_appart, surface)
Occupant (id_personne, id_appart)
```

Ce schéma, obtenu par décompositions successives, présente la double propriété

- de ne pas avoir perdu d'information par rapport à la version initiale ;
- de ne contenir que des relations normalisées.

Important : L'absence de perte d'information est une notion qui est survolée ici mais qui est de fait essentielle. Une décomposition est sans perte d'information s'il est possible, par des opérations, de reconstruire la relation initiale, avant décomposition. Ces opérations, et notamment la jointure, seront présentés dans les chapitres suivants.

Cet exemple est une démonstration d'une approche quasiment algorithmique pour obtenir un schéma normalisé à partir d'un ensemble d'attributs initial. Cette approche est malheureusement inutilisable en pratique à cause d'une difficulté que nous avons cachée : les dépendances fonctionnelles présentes dans notre schéma ont été artificiellement créées par ajout d'identifiants pour les immeubles, les occupants et les appartements. En pratique, de tels identifiants n'existent

pas si on n'a pas au préalable déterminé les "entités" présentes dans le schéma : *Immeuble*, *Occupant*, et *Appartement*. En d'autres termes, l'exemple qui précède s'appuie sur une forme de tricherie : on a normalisé un schéma dans lequel on avait guidé à l'avance la décomposition.

2.1.5 Une approche pratique

Reprenons notre table des films pour nous confronter à une situation réaliste. Rappelons les quelques attributs considérés.

```
(titre, annee, prenomMes, nomMES, anneeNaiss)
```

La triste réalité est qu'on ne trouve aucune dépendance fonctionnelle dans cet ensemble d'attribut. Le titre d'un film ne détermine rien puisqu'il y a évidemment des films différents avec le même titre, Eventuellement, la paire *(titre, année)* pourrait déterminer de manière univoque un film, mais un peu de réflexion suffit à se convaincre qu'il est très possible de trouver deux films différents avec le même titre la même année. Et ainsi de suite : le nom du réalisateur ou même la paire *(prénom, nom)* sont des candidats très fragiles pour définir des dépendances fonctionnelles. En fait, on constate qu'il est très rare en pratique de trouver des DFs "naturelles" sur lesquelles on peut solidement s'appuyer pour définir un schéma.

Il nous faut donc une démarche différente consistant à créer *artificiellement* des DFs parmi les ensembles d'attributs. La connaissance des identifiants d'appartement, d'immeuble et de personne dans notre exemple précédent correspondait à une telle création artificielle : tous les attributs de, respectivement, *Immeuble*, *Appartement* et *Personne* dépendent fonctionnellement, par construction, de leurs identifiants respectifs, ajoutés au schéma.

Comment trouve-t-on ces identifiants ? Par une démarche essentiellement différente de celle partant "de bas en haut", consistant à prendre un tas d'attributs et à les regrouper en relations. Au contraire, on part de groupes d'attributs appartenant à une même *entité*, à laquelle on ajoute un attribut-identifiant. La démarche consiste à :

- déterminer les "entités" (immeuble, personne, appartement, ou film et réalisateur) pertinents pour l'application ;
- définir une méthode *d'identification* de chaque entité ; en pratique on recourt à la définition d'un *identifiant* artificiel (il n'a aucun rôle descriptif) qui permet d'une part de s'assurer qu'une même "entité" est représentée une seule fois, d'autre part de référencer une entité par son identifiant.
- préserver le lien entre les entités sans introduire de redondance, par un mécanisme de référence basé sur la clé.

Voici une illustration informelle de la méthode, que nous reprendrons ensuite de manière plus détailée avec la notation Entité/association.

Commençons par les deux premières étapes. On va d'abord distinguer la table des films et la table des réalisateurs. Ensuite, on va ajouter à chaque table un attribut spécial, l'identifiant, désigné par `id`.

Important : Le choix de l'identifiant est un sujet délicat. On peut arguer en effet que l'identifiant devrait être recherché dans les attributs existant, au lieu d'en créer un artificiellement. Pour des raisons qui tiennent à la rareté/fragilité des DFs "naturelles", la création de l'identifiant artificiel est la seule réellement applicable et satisfaisante dans tous les cas.

On obtient le résultat suivant.

id	titre	année
1	Alien	1979
2	Vertigo	1958
3	Psychose	1960
4	Kagemusha	1980
5	Volte-face	1997
6	Pulp Fiction	1995
7	Titanic	1997
8	Sacrifice	1986

La table des films.

id	titre	année	
101	Scott	Ridley	1943
102	Hitchcock	Alfred	1899
103	Kurosawa	Akira	1910
104	Woo	John	1946
105	Tarantino	Quentin	1963
106	Cameron	James	1954
107	Tarkovski	Andrei	1932

La table des réalisateurs

Premier progrès : il n'y a maintenant plus de redondance dans la base de données. Le réalisateur Hitchcock, par exemple, n'apparaît plus qu'une seule fois, ce qui élimine les anomalies de mise à jour évoquées précédemment.

Important : On peut noter que, par construction, les schémas obtenus sont normalisés.

Il reste à représenter le lien entre les films et les metteurs en scène, sans introduire de redondance. Maintenant que nous avons défini les identifiants, il existe un moyen simple pour indiquer quel est le metteur en scène qui a réalisé un film : associer l'identifiant du metteur en scène au film. L'identifiant sert alors de *référence* à l'entité. On ajoute un attribut `idRéalisateur` dans la table *Film*, et on obtient la représentation suivante.

id	titre	année	idRéalisateur
1	Alien	1979	101
2	Vertigo	1958	102
3	Psychose	1960	102
4	Kagemusha	1980	103
5	Volte-face	1997	104
6	Pulp Fiction	1995	105
7	Titanic	1997	106
8	Sacrifice	1986	107

Cette représentation est correcte. La redondance est réduite au minimum puisque seule l'identifiant du metteur en scène a été déplacé dans une autre table. Pour peu qu'on s'assure que cet identifiant ne change *jamais*, cette redondance n'induit aucun effet négatif. On peut vérifier que les anomalies que nous avons citées ont disparu.

D'une part, il n'y a plus de redondance des attributs descriptifs, donc toute mise à jour affecte l'unique occurrence de la donnée à modifier. D'autre part, on peut détruire un film sans affecter les informations sur le réalisateur. Ce gain dans la qualité du schéma n'a pas pour contrepartie une perte d'information. Il est en effet facile de voir que l'information initiale (autrement dit, *avant* la décomposition en deux tables) peut être reconstituée intégralement. En prenant un film, on obtient l'identifiant de son metteur en scène, et cette identifiant permet de trouver *l'unique* ligne dans la table des réalisateurs qui contient toutes les informations sur ce metteur en scène. Ce processus de reconstruction de l'information, dispersée dans plusieurs tables, peut s'exprimer avec les opérations relationnelles, et notamment la jointure.

Il reste à appliquer une méthode systématique visant à aboutir au résultat ci-dessus, et ce même dans des cas beaucoup plus complexes. Cela étant, *aucune* méthode ne produit automatiquement un résultat correct, puisqu'elle repose sur un processus d'analyse et d"identification des entités dont rien ne peut garantir la validité. On ne peut jamais se dispenser de réfléchir et de comprendre les données que l'on veut modéliser. Il faut, toujours, repérer les *entités* et leur attribuer un identifiant, et gérer correctement les références (ou *associations*) entre ces entités. Entités, associations, sont les concepts essentiels. D'où le modèle présenté dans ce qui suit.

2.2 Le modèle E/A

Le modèle Entité/Association (E/A) propose essentiellement une notation pour soutenir la démarche de conception de schéma présentée précédemment. La notation E/A a pour caractéristiques d'être simple et suffisamment puissante pour modéliser des structures relationnelles. De plus, elle repose sur une représentation graphique qui facilite considérablement sa compréhension.

Le modèle E/A souffre également de nombreuses insuffisances : la principale est de ne proposer que des *structures*. Il n'existe pas d'opération permettant de manipuler les données, et pas (ou peu) de moyen d'exprimer des contraintes. Un autre inconvénient du modèle E/A est de mener à certaines ambiguités pour des schémas complexes.

2.2.1 Le schéma de la base *Films*

La présentation qui suit est délibérement axée sur l'utilité du modèle E/A dans le cadre de la conception d'une base de données. Ajoutons qu'il ne s'agit pas directement de *concevoir* un schéma E/A (voir un cours sur les systèmes d'information), mais d'être capable de le comprendre et de l'interpréter. Nous reprenons l'exemple d'une base de données décrivant des films, avec leur metteur en scène et leurs acteurs, ainsi que les cinémas où passent ces films. Nous supposerons également que cette base de données est accessible sur le Web et que des internautes peuvent noter les films qu'ils ont vus.

La méthode permet de distinguer les *entités* qui constituent la base de données, et les *associations* entre ces entités. Un schéma E/A décrit l'application visée, c'est-à-dire une *abstraction* d'un domaine d'étude, pertinente relativement aux objectifs visés. Rappelons qu'une abstraction consiste à choisir certains aspects de la réalité perçue (et donc à éliminer les autres). Cette sélection se fait en fonction de certains *besoins*, qui doivent être précisément définis, et rélève d'une démarche d'analyse qui n'est pas abordée ici.

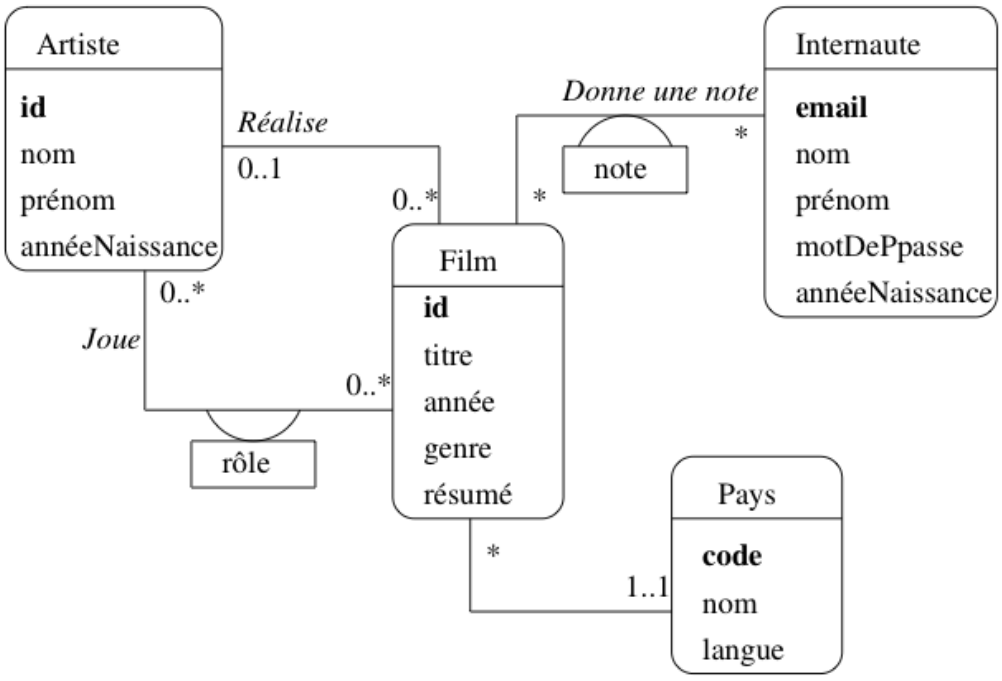

Fig. 2.1 – Le schéma E/A des films

Par exemple, pour notre base de données *Films*, on n'a pas besoin de stocker dans la base de données l'intégralité des informations relatives à un internaute, ou à un film. Seules comptent celles qui sont importantes pour l'application.

Voici le schéma décrivant cete base de données *Films* (figure Fig. 2.1). Sans entrer dans les détails pour l'instant, on distingue

— des *entités*, représentées par des rectangles, ici *Film*, *Artiste*, *Internaute* et *Pays* ;
— des *associations* entre entités représentées par des liens entre ces rectangles. Ici on a représenté par exemple le fait qu'un artiste *joue* dans des films, qu'un internaute *note* des films, etc.

Chaque entité est caractérisée par un ensemble d'attributs, parmi lesquels un ou plusieurs forment l'identifiant unique (en gras). Il est essentiel de dire ce qui caractérise de manière unique une entité, de manière à éviter la redondance d'information. Comme nous l'avons préconisé précédemment, un attribut non-descriptif a été ajouté à chaque entité, indépendamment des attributs "descriptifs". Nous l'avons appelé **id** pour *Film* et *Artiste*, **code** pour le pays. Le nom de l'attribut-identifiant est peu important, même si la convention **id** est très répandue.

Seule exception : les internautes sont identifiés par un de leurs attributs descriptifs, leur adresse de courrier électronique. Même s'il s'agit en apparence d'un choix raisonnable (unicité de l'email pour identifier une personne), ce cas nous permettra d'illustrer les problèmes qui peuvent quand même se poser.

Les associations sont caractérisées par des *cardinalités*. La notation 0..* sur le lien *Réalise*, du côté de l'entité *Film*, signifie qu'un artiste peut réaliser plusieurs films, ou aucun. La notation 0..1 du côté *Artiste* signifie en revanche qu'un film ne peut être réalisé que par au plus un artiste. En revanche dans l'association *Donne une note*, un internaute peut noter plusieurs films, et un film peut être noté par plusieurs internautes, ce qui justifie l'a présence de 0..* aux deux extrêmités de l'association.

Le choix des cardinalités est *essentiel*. Ce choix est aussi parfois discutable, et constitue donc l'un des aspects les plus délicats de la modélisation. Reprenons l'exemple de l'association *Réalise*. En indiquant qu'un film est réalisé par *un seul* metteur en scène, on s'interdit les – pas si rares – situations où un film est réalisé par plusieurs personnes. Il ne sera donc pas possible de représenter dans la base de données une telle situation. Tout est ici question de choix et de compromis : est-on prêt en l'occurrence à accepter une structure plus complexe (avec 0..* de chaque côté) pour l'association *Réalise*, pour prendre en compte un nombre minime de cas ?

Les cardinalités sont notées par deux chiffres. Le chiffre de droite est la *cardinalité maximale*, qui vaut en général 1 ou . *Le chiffre de gauche est la cardinalité minimale. Par exemple la notation 0..1 entre *Artiste* et *Film* indique qu'on s'autorise à ne pas connaître le metteur en scène d'un film. Attention : cela ne signifie pas que ce metteur en scène n'existe pas. Une base de données, telle qu'elle est décrite par un schéma E/A, ne prétend pas donner une vision exhaustive de la réalité. On ne doit surtout pas chercher à *tout* représenter, mais s'assurer de la prise en compte des besoins de l'application.

La notation 1..1 entre *Film* et *Pays* indique au contraire que l'on doit toujours connaître le pays producteur d'un film. On devra donc interdire le stockage dans la base d'un film sans son pays.

Les cardinalités minimales sont moins importantes que les cardinalités maximales, car elles ont un impact limité sur la structure de la base de données et peuvent plus facilement être remises en cause après coup. Il faut bien être conscient de plus qu'elles ne représentent qu'un choix de conception, souvent discutable. Dans la notation UML que nous présentons ici, il existe des notations abrégées qui donnent des valeurs implicites aux cardinalités minimales :

— La notation * est équivalente à 0..* ;
— la notation 1 est équivalente à 1..1 .

Outre les propriétés déjà évoquées (simplicité, clarté de lecture), évidentes sur ce schéma, on peut noter aussi que la modélisation conceptuelle est totalement indépendante de tout choix d'implantation. Le schéma de la figure *Le schéma E/A des films* (page 12) ne spécifie aucun système en particulier. Il n'est pas non plus question de type ou de structure de données, d'algorithme, de langage, etc. En principe, il s'agit donc de la partie la plus stable d'une application. Le fait de se débarrasser à ce stade de la plupart des considérations techniques permet de se concentrer sur l'essentiel : que veut-on stocker dans la base ?

Une des principales difficultés dans le maniement des schémas E/A est que la qualité du résultat ne peut s'évaluer que par rapport à une demande qui est difficilement formalisable. Il est donc souvent difficile de mesurer (en fonction de quels critères et quelle métrique ?) l'adéquation du résultat au besoin. Peut-on affirmer par exemple que :

— *toutes* les informations nécessaires sont représentées ?
— qu'un film ne sera *jamais* réalisé par plus d'un artiste ?

Il faut faire des choix, en connaissance de cause, en sachant toutefois qu'il est toujours possible de faire évoluer une base de données, quand cette évolution n'implique pas de restructuration trop importante. Pour reprendre les exemples ci-dessus, il est facile d'ajouter des informations pour décrire un film ou un internaute ; il serait beaucoup plus difficile de modifier la base pour qu'un film passe de un, et un seul, réalisateur, à plusieurs. Quant à changer l'identifiant de la table *Internaute*, c'est une des évolutions les plus complexes à réaliser. Les cardinalités et le choix des clés font vraiment partie des aspects décisifs des choix de conception.

2.2.2 Entités, attributs et identifiants

Il est difficile de donner une définition très précise des entités. Les points essentiels sont résumés ci-dessous.

Définition : Entités

On désigne par *entité* toute unité d'information *identifiable* et *pertinente* pour l'application.

La notion d'unité d'information correspond au fait qu'une entité ne peut pas se décomposer sans perte de sens. Comme nous l'avons vu précédemment, *l'identité* est primordiale. C'est elle qui permet de distinguer les entités les unes des autres, et donc de dire qu'une information est redondante ou qu'elle ne l'est pas. Il est indispensable de prévoir un moyen technique pour pouvoir effectuer cette distinction entre entités au niveau de la base de données : on parle *d'identifiant* ou (dans un contexte de base de données) de *clé*. Reportez-vous au début du chapitre pour une définition précise de cette notion.

La pertinence est également importante : on ne doit prendre en compte que les informations nécessaires pour satisfaire les besoins. Par exemple :

— le film *Impitoyable* ;
— l'acteur *Clint Eastwood* ;

sont des entités pour la base *Films*.

La première étape d'une conception consiste à identifier les entités utiles. On peut souvent le faire en considérant quelques cas particuliers. La deuxième est de regrouper les entités en ensembles : en général on ne s'intéresse pas à un individu particulier mais à des groupes. Par exemple il est clair que les films et les acteurs constituent des ensembles distincts d'entités. Qu'en est-il de l'ensemble des réalisateurs et de l'ensemble des acteurs ? Doit-on les distinguer ou les assembler ? Il est certainement préférable de les assembler, puisque des acteurs peuvent aussi être réalisateurs.

Attributs

Les entités sont caractérisées par des *attributs* (ou *propriétés*) : le titre (du film), le nom (de l'acteur), sa date de naissance, l'adresse, etc. Le choix des attributs relève de la même démarche d'abstraction qui a dicté la sélection des entités : il n'est pas nécéssaire de donner exhaustivement tous les attributs d'une entité. On ne garde que ceux utiles pour l'application.

Un attribut est désigné par un *nom* et prend sa valeur dans un domaine comme les entiers, les chaînes de caractères, les dates, etc. On peut considérer un nom d'atribut A comme une fonction définie sur un ensemble d'entités E et prenant ses valeurs dans un domaine A pour une entité E.

Selon cette définition un attribut prend une valeur et une seule. On dit que les attributs sont *atomiques*. Il s'agit d'une restriction importante puisqu'on s'interdit, par exemple, de définir un attribut *téléphones* d'une entité *Personne*, prenant pour valeur *les* numéros de téléphone d'une personne. Cette restricion est l'un des inconvénients du modèle relationnel, qui mène à la multiplication des tables par le mécanisme de normalisation dérit en début de chapitre. Pour notre exemple, il faudrait par exemple définir une table dédiée aux numéros de téléphone et associée aux personnes.

Note : Certaines méthodes admettent l'introduction de constructions plus complexes :

— les *attributs multivalués* sont constitués d'un *ensemble* de valeurs prises dans un même domaine ; une telle construction permet de résoudre le problème des numéros de téléphones multiples ;

— les *attributs composés* sont constitués par agrégation d'autres atributs ; un attribut *adresse* peut par exemple être décrit comme l'agrégation d'un code postal, d'un numéro de rue, d'un nom de rue et d'un nom de ville.

Cette modélisation dans le modèle conceptuel (E/A) doit pouvoir être transposée dans la base de données. Certains systèmes relationnels (PostgreSQL par exemple) autorisent des attributs complexes. Une autre solution est de recourir à d'autres modèles, semi-structurés ou objets.

Nous nous en tiendrons pour l'instant aux attributs atomiques qui, au moins dans le contexte d'une modélisation orientée vers un SGBD relationnel, sont suffisants.

2.2.3 Types d'entités

Il est maintenant possible de décrire un peu plus précisément les entités par leur *type*.

Définition : Type d'entité

Le type d'une entité est composé des éléments suivants :

— son nom ;
— la liste de ses attributs avec, – optionnellement – le domaine où l'attribut prend ses valeurs : les entiers, les chaînes de caractères ;
— l'indication du (ou des) attribut(s) permettant d'identifier l'entité : ils constituent la *clé*.

On dit qu'une entité *e* est une *instance* de son type *E*. Enfin, un ensemble d'entités $\{e_1, e_2, \ldots e_n\}$, instances d'un même type E est une *extension* de E.

Rappelons maintenant la notion de clé, pratiquement identique à celle énoncée pour les shémas relationnels.

Définition : clé

Soit E un type d'entité et A l'ensemble des attributs de E. Une *clé* de E est un sous-ensemble *minimal* de A permettant d'identifier de manière unique une entité parmi n'importe quelle extension de E.

Prenons quelques exemples pour illustrer cette définition. Un internaute est caractérisé par plusieurs attributs : son email, son nom, son prénom, la région où il habite. L'adresse mail constitue une clé naturelle puisqu'on ne trouve pas, en principe, deux internautes ayant la même adresse électronique. En revanche l'identification par le nom seul paraît impossible puisqu'on constituerait facilement un ensemble contenant deux internautes avec le même nom. On pourrait penser à utiliser la paire `(nom,prénom)`, mais il faut utiliser avec modération l'utilisation d'identifiants composés de plusieurs attributs. Quoique possible, elle peut poser des problèmes de performance et complique les manipulations par SQL.

Il est possible d'avoir plusieurs clés candidates pour un même ensemble d'entités. Dans ce cas on en choisit une comme *clé principale* (ou primaire), et les autres comme clés secondaires. Le choix de la clé (primaire) est déterminant pour la qualité du schéma de la base de données. Les caractéristiques d'une bonne clé primaire sont les suivantes :

— elle désigne sans ambiguité une et une seule entité dans toute extension ;
— sa valeur est connue pour toute entité ;
— on ne doit jamais avoir besoin de la modifier ;
— enfin, pour des raisons de performance, sa taille de stockage doit être la plus petite possible.

Il est très difficile de trouver un ensemble d'attributs satisfaisant ces propriétés parmi les attributs descriptifs d'une entité. Considérons l'exemple des films. Le choix du titre pour identifier un film serait incorrect puisqu'on aura affaire un jour ou l'autre à deux films ayant le même titre. Même en combinant le titre avec un autre attribut (par exemple l'année), il est difficile de garantir l'unicité.

Le choix de l'adresse électronique (email) pour un internaute semble respecter ces conditions, du moins la première (unicité). Mais peut-on vraiment garantir que l'email sera connu au moment de la création de l'entité ? De plus, il

semble clair que cette adresse peut changer, ce qui va poser de gros problèmes puisque la clé, comme nous le verrons, sert à référencer une entité. Changer l'identifiant de l'entité implique donc de changer *aussi* toutes les références. La conclusion s'impose : ce choix d'identifiant est un mauvais choix, il posera à terme des problèmes pratiques.

Insistons : la seule solution saine et générique consiste à créer un identifiant artificiel, indépendant de tout autre attribut. On peut ainsi ajouter dans le type d'entité *Film* un attribut `id`, corespondant à un numéro séquentiel qui sera incrémenté au fur et à mesure des insertions. Ce choix est de fait le meilleur, dès lors qu'un attribut ne respecte pas les conditions ci-dessus (autrement dit, toujours). Il satisfait ces conditions : on peut toujours lui attribuer une valeur, il ne sera jamais nécessaire de la modifier, et elle a une représentation compacte.

On représente graphiquement un type d'entité comme sur la figure *Représentation des types d'entité* (page 16) qui donne l'exemple des types *Internaute* et *Film*. L'attribut (ou les attributs s'il y en a plusieurs) formant la clé sont en gras.

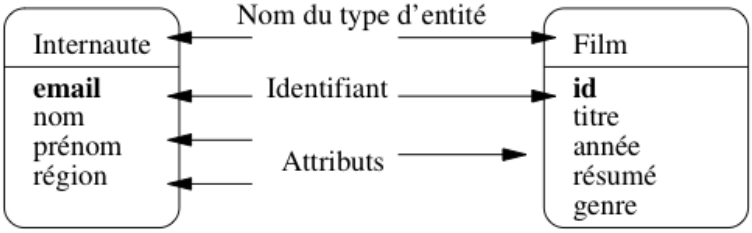

Fig. 2.2 – Représentation des types d'entité

Il est important de bien distinguer *types d'entités* et *entités*. La distinction est la même qu'entre entre *type* et *valeur* dans un langage de programmation, ou *schéma et *base* dans un SGBD, comme nous le verrons.

2.2.4 Associations binaires

La représentation (et le stockage) d'entités indépendantes les unes des autres est de peu d'utilité. On va maintenant décrire les *relations* (ou *associations*) entre des ensembles d'entités.

Définition : association

Une association binaire entre les ensembles d'entités E_1 et E_1 est un ensemble de couples (e_1, e_2), avec $e_1 \in E_1$ et $e_2 \in E_2$.

C'est la notion classique, ensembliste, de relation. On emploie plutôt le terme d'association pour éviter toute confusion avec le modèle relationnel. Une bonne manière d'interpréter une association entre des ensembles d'entités est de faire un petit graphe où on prend quelques exemples, les plus généraux possibles.

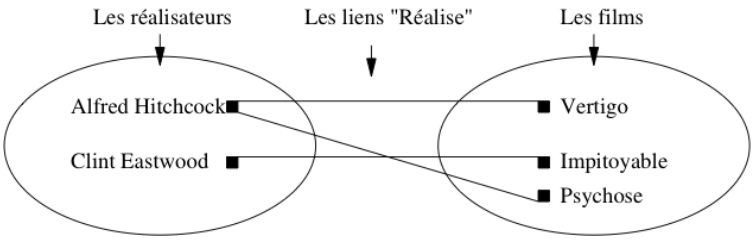

Fig. 2.3 – Association entre deux ensembles

Prenons l'exemple de l'association représentant le fait qu'un réalisateur met en scène des films. Sur le graphe de la figure *Association entre deux ensembles* (page 16) on remarque que :

— certains réalisateurs mettent en scène plusieurs films ;
— inversement, un film est mis en scène par au plus un réalisateur.

La recherche des situations les plus générales possibles vise à s'assurer que les deux caractéristiques ci-dessus sont vraies dans tout les cas. Bien entendu on peut trouver *x %* des cas où un film a plusieurs réalisateurs, mais la question se pose alors : doit-on modifier la structure de notre base, pour *x %* des cas. Ici, on a décidé que non. Encore une fois on ne cherche pas à représenter la réalité dans toute sa complexité, mais seulement la partie de cette réalité que l'on veut stocker dans la base de données.

Ces caractéristiques sont essentielles dans la description d'une association entre des ensembles d'entités.

Définition : cardinalités

Soit une association (E_1, E_2) entre deux types d'entités. La cardinalité de l'association pour $E_i, i \in \{1, 2\}$, est une paire $[min, max]$ telle que :

— Le symbole *max* (cardinalité maximale) désigne le nombre *maximal* de fois où une une entité e_i peut intervenir dans l'association.
En général, ce nombre est 1 (au plus une fois) ou n (plusieurs fois, nombre indeterminé), noté par le symbole *.
— Le symbole *min* (cardinalité minimale) désigne le nombre *minimal* de fois où une une entité e_i peut intervenir dans l'association.
En général, ce nombre est 1 (au moins une fois) ou 0.

Les cardinalités maximales sont plus importantes que les cardinalités minimales ou, plus précisément, elles s'avèrent beaucoup plus difficiles à remettre en cause une fois que le schéma de la base est constitué. On décrit donc souvent une association de manière abrégée en omettant les cardinalités minimales. La notation * en UML, est l'abréviation de *0..**, et *1* est l'abréviation de *1..1*. On caractérise également une association de manière concise en donnant les cardinalités maximales aux deux extrêmités, par exemple *1 :** (association de un à plusieurs) ou ** :** (association de plusieurs à plusieurs).

Les cardinalités minimales sont parfois désignées par le terme *contraintes de participation*. La valeur 0 indique qu'une entité peut ne pas participer à l'association, et la valeur 1 qu'elle doit y participer.

Insistons sur le point suivant : *les cardinalités n'expriment pas une vérité absolue, mais des choix de conception*. Elles ne peuvent être déclarés valides que relativement à un besoin. Plus ce besoin sera exprimé précisément, et plus il sera possible d'appécier la qualité du modèle.

Il existe plusieurs manières de noter une association entre types d'entités. Nous utilisons ici la notation de la méthode UML. En France, on utilise aussi couramment – de moins en moins... – la notation de la méthode MERISE que nous ne présenterons pas ici.

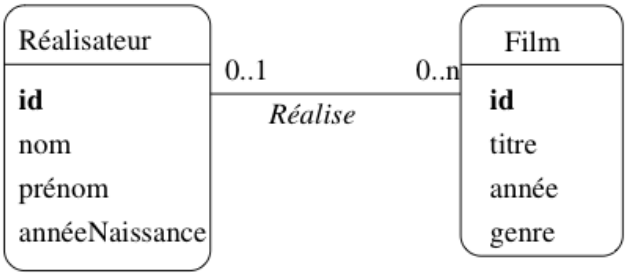

Fig. 2.4 – Représentation de l'association

Dans la notation UML, on indique les cardinalités aux deux extrêmités d'un lien d'association entre deux types d'entités T_A et T_B. Les cardinalités pour T_A sont placées à l'extrémité du lien allant de T_A à T_B et les cardinalités pour

T_B sont l'extrémité du lien allant de T_B à T_A.

Pour l'association entre *Réalisateur* et *Film*, cela donne l'association de la figure *Représentation de l'association* (page 17). Cette association se lit *Un réalisateur réalise zéro, un ou plusieurs films*, mais on pourrait tout aussi bien utiliser la forme passive avec comme intitulé de l'association *Est réalisé par* et une lecture *Un film est réalisé par au plus un réalisateur*. Le seul critère à privilégier dans ce choix des termes est la clarté de la représentation.

Prenons maintenant l'exemple de l'association *(Acteur, Film)* représentant le fait qu'un acteur joue dans un film. Un graphe basé sur quelques exemples est donné dans la figure *Association (Acteur,Film)* (page 18). On constate tout d'abord qu'un acteur peut jouer dans plusieurs films, et que dans un film on trouve plusieurs acteurs. Mieux : Clint Eastwood, qui apparaissait déjà en tant que metteur en scène, est maintenant également acteur, et dans le même film.

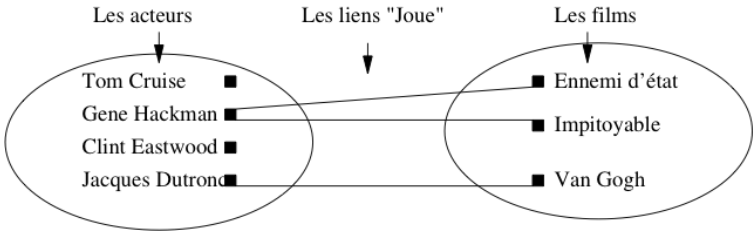

Fig. 2.5 – Association *(Acteur,Film)*

Cette dernière constatation mène à la conclusion qu'il vaut mieux regrouper les acteurs et les réalisateurs dans un même ensemble, désigné par le terme plus général "Artiste". On obtient le schéma de la figure *Association entre Artiste et Film* (page 18), avec les deux associations représentant les deux types de lien possible entre un artiste et un film : il peut jouer dans le film, ou le réaliser. Ce "ou" n'est pas exclusif : Eastwood joue dans *Impitoyable*, qu'il a aussi réalisé.

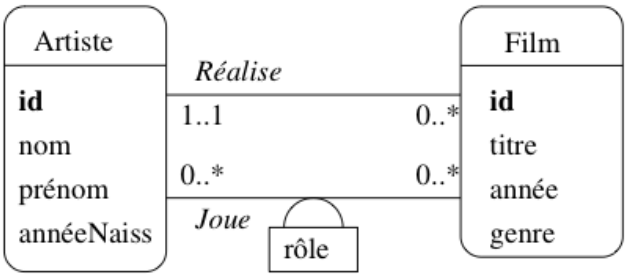

Fig. 2.6 – Association entre *Artiste* et *Film*

Dans le cas d'associations avec des cardinalités multiples de chaque côté, on peut avoir des attributs qui ne peuvent être affectés qu'à l'association elle-même. Par exemple l'association *Joue* a pour attribut le rôle tenu par l'acteur dans le film (figure *Association entre Artiste et Film* (page 18)).

Rappelons qu'un attribut ne peut prendre qu'une et une seule valeur. Clairement, on ne peut associer `rôle` ni à *Acteur* puisqu'il a autant de valeurs possibles qu'il y a de films dans lesquels cet acteur a joué, ni à *Film*, la réciproque étant vraie également. Seules les associations ayant des cardinalités multiples de chaque côté peuvent porter des attributs.

Quelle est la clé d'une association ? Si l'on s'en tient à la définition, une association est un *ensemble* de couples, et il ne peut donc y avoir deux fois le même couple (parce qu'on ne trouve pas deux fois le même élément dans un ensemble). On a donc :

Définition : Clé d'une association

La clé d'une association (binaire) entre un type d'entité E_1 et un type d'entité E_2 est la paire constituée de la clé c_1 de E_1 et de la clé c_2 de E_2.

En pratique cette contrainte est souvent trop contraignante car on souhaite autoriser deux entités à être liées plus d'une fois dans une association. Imaginons par exemple qu'un internaute soit amené à noter à plusieurs reprises un film, et que l'on souhaite conserver l'historique de ces notations successives. Avec une association binaire entre *Internaute* et *Film*, c'est impossible : on ne peut définir qu'un seul lien entre un film donné et un internaute donné.

Le problème est qu'il n'existe pas de moyen pour distinguer des liens multiples entre deux mêmes entités. Le seul moyen pour effectuer une telle distinction est d'introduire une entité discriminante, par exemple la date de la notation. On obtient alors une association ternaire dans laquelle on a ajouté un type d'entité *Date* (figure *Ajout d'une entité Date pour conserver l'historique des notations* (page 19)).

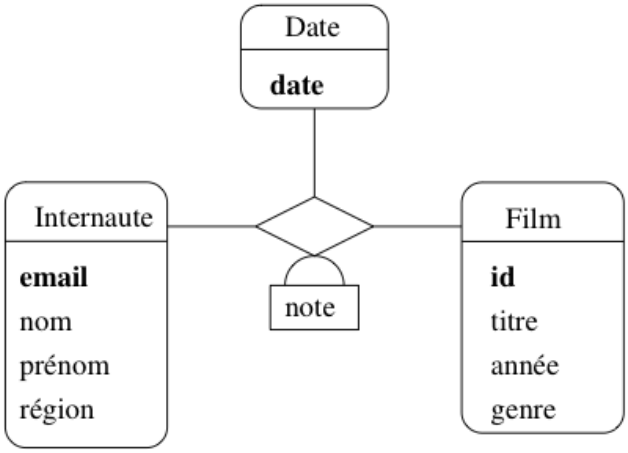

Fig. 2.7 – Ajout d'une entité *Date* pour conserver l'historique des notations

Un lien de cette association réunit donc une entité *Film*, une entité *Internaute* et une entité *Date*. On peut identifier un tel lien par un triplet `(id,email,date)` constitué par les clés des trois entités constituant le lien.

Comme le montre la figure *Graphe d'une association ternaire* (page 19), il devient alors possible, pour un même internaute, de noter plusieurs fois le même film, pourvu que ce ne soit pas à la même date. Réciproquement un internaute peut noter des films différents le même jour, et un même film peut être noté plusieurs fois à la même date, à condition que ce ne soit pas par le même internaute.

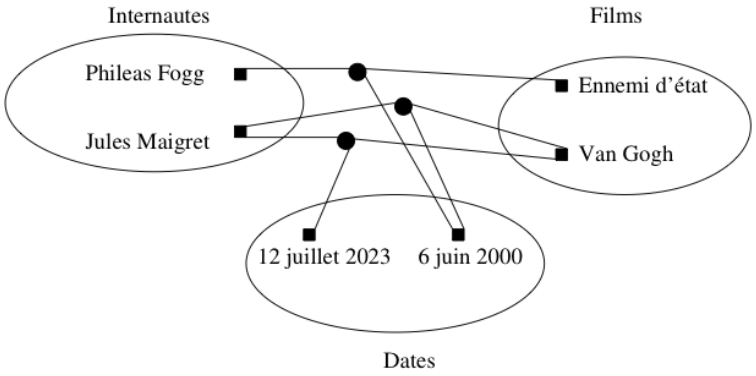

Fig. 2.8 – Graphe d'une association ternaire

Même si cette solution est correcte, elle présente l'inconvénient d'introduire une entité assez artificielle, *Date*, qui

porte peu d'information et vient alourdir le schéma. En pratique on s'autorise une notation abrégée en ajoutant un attribut `date` dans l'association, et en le soulignant pour indiquer qu'il fait partie de la clé, *en plus du couple des clés des entités* (voir figure *Notation abrégée d'une association avec un type d'entité Date* (page 20)).

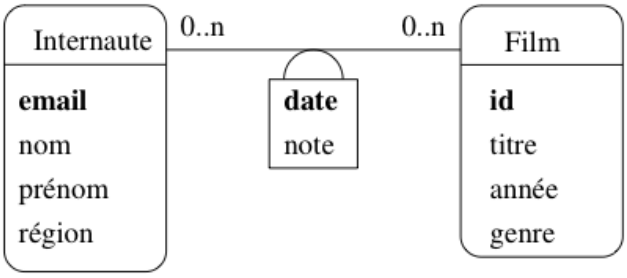

Fig. 2.9 – Notation abrégée d'une association avec un type d'entité *Date*

Nous reviendrons plus longuement sur les associations ternaires par la suite.

2.2.5 Entités faibles

Jusqu'à présent nous avons considéré le cas d'entités *indépendantes* les unes des autres. Chaque entité, disposant de son propre identifiant, pouvait être considérée isolément. Il existe des cas où une entité ne peut exister qu'en étroite association avec une autre, et est identifiée relativement à cette autre entité. On parle alors *d'entité faible*.

Prenons l'exemple d'un cinéma, et de ses salles. On peut considérer chaque salle comme une entité, dotée d'attributs comme la capacité, l'équipement en son Dolby, ou autre. Il est diffcilement imaginable de représenter une salle sans qu'elle soit rattachée à son cinéma. C'est en effet au niveau du cinéma que l'on va trouver quelques informations générales comme l'adresse de la salle.

Il est possible de représenter le lien en un cinéma et ses salles par une association classique, comme le montre la figure *Modélisations possibles du lien Cinéma-Salle* (page 21).a. La cardinalité 1..1 force la participation d'une salle à un lien d'association avec un et un seul cinéma. Cette représentation est correcte, mais présente un inconvénient : on doit créer un identifiant artificiel `id` pour le type d'entité *Salle*, et numéroter toutes les salles, *indépendamment du cinéma auquel elles sont rattachées*.

On peut considérer qu'il est beaucoup plus naturel de numéroter les salles par un numéro interne à chaque cinéma. La clé d'identification d'une salle est alors constituée de deux parties :

— la clé de *Cinéma*, qui indique dans quel cinéma se trouve la salle ;
— le numéro de la salle au sein du cinéma.

En d'autres termes, l'entité *Salle* ne dispose pas d'une identification absolue, mais d'une identification *relative* à une autre entité. Bien entendu cela force la salle a toujours être associée à un et un seul cinéma.

La représentation graphique des entités faibles avec UML est illustrée dans la figure *Modélisations possibles du lien Cinéma-Salle* (page 21).b. La salle est associée au cinéma avec une association qualifiée par l'attribut `no` qui sert de discriminant pour distinguer les salles au sein d'un même cinéma. Noter que la cardinalité du côté *Cinéma* est implicitement 1..1.

L'introduction d'entités faibles est un subtilité qui permet de capturer une caractéristique intéressante du modèle. Elle n'est pas une nécessité absolue puisqu'on peut très bien utiliser une association classique. La principale différence est que, dans le cas d'une entité faible, on obtient une identification composée qui peut être plus pratique à gérer, et peut également rendre plus faciles certaines requêtes. On touche ici à la liberté de choix qui est laissée, sur bien des aspects, à un "modeleur" de base de données, et qui nécessite de s'appuyer sur une expérience robuste pour apprécier les conséquences de telle ou telle décision.

La présence d'un type d'entité faible B associé à un type d'entité A implique également des contraintes fortes sur les créations, modifications et destructions des instances de A car on doit toujours s'assurer que la contrainte est valide.

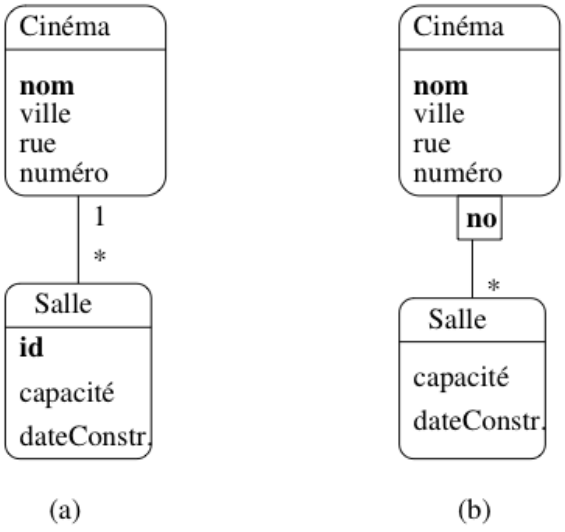

Fig. 2.10 – Modélisations possibles du lien Cinéma-Salle

Concrètement, en prenant l'exemple de *Salle* et de *Cinéma*, on doit mettre en place les mécanismes suivants :

— quand on insère une salle dans la base, on doit toujours l'associer à un cinéma ;
— quand un cinéma est détruit, on doit aussi détruire toutes ses salles ;
— quand on modifie la clé d'un cinéma, il faut répercuter la modification sur toutes ses salles.

Réfléchissez bien à ces mécanismes pour apprécier le sucroît de contraintes apporté par des variantes des associations. Parmi les impacts qui en découlent, et pour respecter les règles de destruction/création énoncées, on doit mettre en place une stratégie. Nous verrons que les SGBD relationnels nous permettent de spécifier de telles stratégies.

2.2.6 Associations généralisées

On peut envisager des associations entre plus de deux entités, mais elles sont plus difficiles à comprendre, et surtout la signification des cardinalités devient beaucoup plus ambigue. La définition d'une association n-aire est une généralisation de celle des associations binaires.

Définition : associations *n*-aires

Une association *n*-aire entre *n* types d'entités $E_1, E_2, \ldots E_n$ est un ensemble de *n*-uplets $(e_1, e_2, \ldots, e_n)$ où chaque e_i appartient à E_i.

Il n'existe pas en principe pas de limite sur le degré d'une association. En pratique on ne va jamais au-delà d'une association entre trois entités qui est déjà assez difficile à interpréter.

Nous allons prendre l'exemple d'une association permettant de représenter la projection de certains films dans des salles à certains horaires. Il s'agit d'une association ternaire entre les types d'entités *Film*, *Salle* et *Horaire* (figure *Association ternaire représentant les séances* (page 22)). Chaque instance de cette association lie un film, un horaire et une salle. La figure *Graphe d'une association ternaire* (page 22) montre quelques-unes de ces instances.

Bien que, jusqu'à présent, une association ternaire puisse être considérée comme une généralisation directe des associations binaires, en réalité de nouveaux problèmes sont soulevés.

Tout d'abord les cardinalités sont, implicitement, 0..*. Il n'est pas possible de dire qu'une entité ne participe qu'une fois à l'association. Il est vrai que, d'une part la situation se présente rarement, d'autre part cette limitation est due à la notation UML qui place les cardinalités à l'extrémité opposée d'une entité.

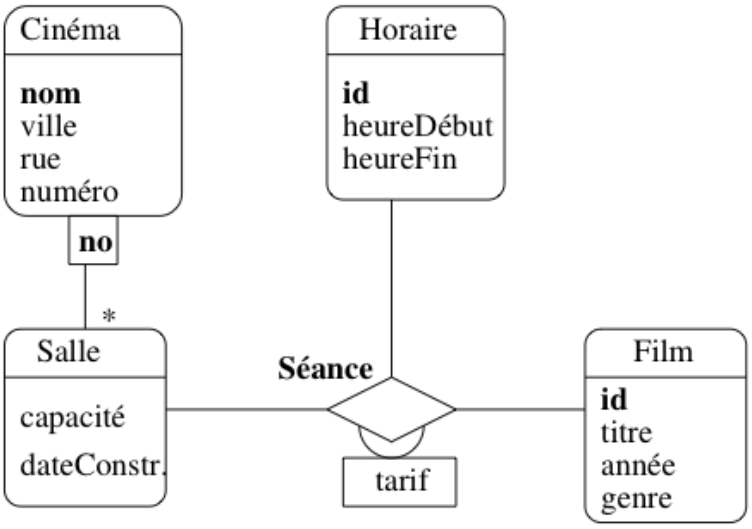

Fig. 2.11 – Association ternaire représentant les séances

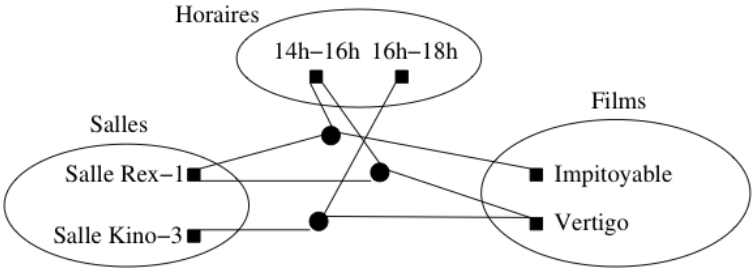

Fig. 2.12 – Graphe d'une association ternaire

Plus problématique en revanche est la détermination de la clé. Qu'est-ce qui identifie un lien entre trois entités ? En principe, la clé est le triplet constitué des clés respectives de la salle, du film et de l'horaire constituant le lien. On aurait donc le n-uplet *[nomCinéma, noSalle, idFilm, idHoraire]*. Une telle clé est assez volumineuse, ce qui risque de poser des problèmes de performance. De plus elle ne permet pas d'imposer certaines contraintes comme, par exemple, le fait que dans une salle, pour un horaire donné, il n'y a qu'un seul film. Comme le montre la figure *Graphe d'une association ternaire* (page 22), il est tout à fait possible de créer deux liens distincts qui s'appuient sur le même horaire et la même salle.

Ajouter une telle contrainte, c'est signifier que la clé de l'association est en fait constitué de *[nomCinéma, noSalle, idHoraire]*. C'est donc un sous-ensemble de la concaténation des clés, ce qui semble rompre avec la définition donnée précédemment. On peut évidemment compliquer les choses en ajoutant une deuxième contrainte similaire, comme connaissant le cinéma, le film et l'horaire, je connais la salle. Il faut ajouter une deuxième clé *[idFilm,idHoraire]*. Il n'est donc plus possible de déduire automatiquement la clé comme on le faisait dans le cas des associations binaires. Plusieurs clés deviennent possibles : on parle de *clé candidates*.

Les associations de degré supérieur à deux sont difficiles à manipuler et à interpréter. Il est *toujours* possible de remplacer cette association par un type d'entité. Pour cela on suit la règle suivante :

Règle

Soit A une association entre les types d'entité $\{E_1, E_2, \ldots, E_n\}$. La transformation de A en type d'entité s'effectue en trois étapes :

- On attribue un identifiant autonome à A.
- On crée une association A_i de type '1 :n' entre A et chacun des A_i *La contrainte minimale, du côté de A, est toujours à 1*.

L'association précédente peut être transformée en un type d'entité *Séance*. On lui attribue un identifiant `idSéance`, et des associations '1..*' avec *Film*, *Horaire* et *Salle*. Voir figure *L'association Séance transformée en entité* (page 23).

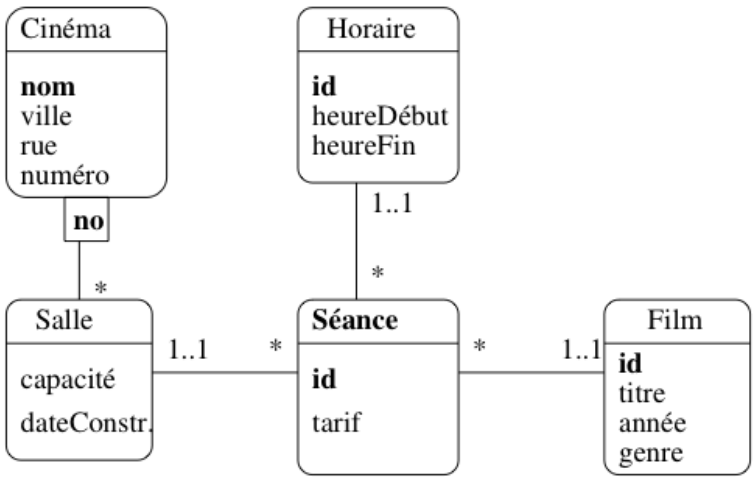

Fig. 2.13 – L'association *Séance* transformée en entité

2.2.7 Bilan

Le modèle Entité/Association est *simple* et *pratique*.

- Il n'y a que 3 concepts : *entités*, *associations* et *attributs*.
- Il est approprié à une représentation graphique intuitive, même s'il existe beaucoup de conventions.

— Il permet de modéliser rapidement des structures pas trop complexes.

Il présente malheureusement plusieurs limitations, qui découlent du fait que beaucoup de choix de conceptions plus ou moins équivalents peuvent découler d'une même spécification, et que la spécification elle-même est dans la plupart du cas informelle et sujette à interprétation.

Un autre inconvénient du modèle E/A reste sa pauvreté : il est difficile d'exprimer des contraintes d'intégrité, des structures complexes. Beaucoup d'extensions ont été proposées, mais la conception de schéma reste en partie matière de bon sens et d'expérience. On essaie en général :

— de se ramener à des associations entre 2 entités : au-delà, on a probablement intérêt a transformer l'association en entité ;
— d'éviter toute redondance : une information doit se trouver en un seul endroit ;
— enfin – et surtout – de privilégier la simplicité et la lisibilité, notamment en ne représentant que ce qui est strictement nécessaire.

La mise au point d'un modèle engage fortement la suite d'un projet de développement de base de données. Elle doit s'appuyer sur des personnes expérimentées, sur l'écoute des prescripteurs, et sur un processus par itération qui identifie les ambiguités et cherche à les résoudre en précisant le besoin correspondant. Dans le cadre des bases de données, le modèle E/A est utilisé dans la phase de conception. Il permet de spécifier la structure des informations qui vont être contenues dans la base et d'offrir une représentation abstraite indépendante du modèle logique qui sera choisi ensuite.

Signalons pour finir que les SGBD fournissent des outils de modélisation étroitement associés à la production et à la maintenance de la base. C'est le cas par exemple de *MySQLWorkbench* que vous pouvez librement récupérer et tester (https://www.mysql.fr/products/workbench/).

CHAPITRE 3

Le modèle relationnel

Un *modèle de données* définit un mode de représentation des données selon trois composantes :

— Des *structures*.
— Des *contraintes*.
— Des *opérations*.

Les deux premières relèvent du *Langage de Définition de Données* (LDD) dans un Système de Gestion de Bases de Données (SGBD). Le LDD est utilisé pour décrire le *schéma* d'une base de données. La troisième composante (opérations) est la base du *Langage de Manipulation de Données* (LMD) dont le représentant le plus célèbre est SQL.

Dans le contexte des bases de données, la principale qualité d'un modèle de données est d'être indépendant de la représentation physique. Cette indépendance permet de séparer totalement les tâches respectives des administrateurs de la base, chargés de l'optimisation de ses performances, et des développeurs d'application ou utilisateurs finaux qui n'ont pas à se soucier de la manière dont le système satisfait leurs demandes.

Le modèle relationnel offre une totale indépendance entre les représentations logique et physique. Ce chapitre présente la partie du modèle relative à la définition et à la création des tables.

3.1 Définition d'un schéma relationnel

Un des grands avantages du modèle relationnel est sa très grande simplicité. Il n'existe en effet qu'une seule structure, la *relation*. Une relation peut simplement être représentée sous forme de *table* (les deux termes seront employés comme des synonymes). Une table a un *nom* (*Film*) et se compose d'un ensemble de colonnes désignées par un *nom d'attribut*. Dans chaque colonne on trouve des valeurs d'un certain *domaine* (chaînes de caractères, nombres). Enfin chaque ligne (ou *nuplet*) correspond à une entité (ici des films).

id	titre	année	genre
1	Alien	1979	Science-Fiction
2	Vertigo	1958	Suspense
3	Psychose	1960	Suspense
4	Kagemusha	1980	Drame
5	Volte-face	1997	Policier
6	Pulp Fiction	1995	Policier
7	Titanic	1997	Drame
8	Sacrifice	1986	Drame

Un schéma relationnel est constitué d'un ensemble de *schémas de relations* qui décrivent, à l'aide des élements présentés informellement ci-dessus (domaines, attributs, noms de relation) le contenu d'une relation. Le schéma de cette table est donc :

```
Film (titre: string, année: number, genre : string)
```

Il existe un langage de définition pour créer une relation dans un SGBDR, mais nous nous contenterons pour l'instant de cette description simplifiée. Voici maintenant quelques précisions sur la terminologie introduite ci-dessus.

Domaines

Un *domaine de valeurs* est un ensemble d'instances d'un type élémentaire. Exemple : les entiers, les réels, les chaînes de caractères, etc. La notion de type *élémentaire* s'oppose à celle de type *structuré* : il n'est pas possble en relationnel de placer dans une cellule un graphe, une liste, un enregistrement. En d'autres termes le système de types est figé et fourni par le système.

Note : C'est une limitation que l'on a cherché à lever par l'extension des systèmes à de nouveaux types. Ces SGBD extensibles ne sont pas présentés ici.

Attributs

Les *attributs* nomment les colonnes d'une relation. Il servent à la fois à indiquer le contenu de cette colonne, et à la référencer quand on effectue des opérations. Un attribut est toujours associé à un *domaine*. Un même nom d'un attribut peut apparaître dans plusieurs schémas distincts.

Schéma de relation

Un schéma est simplement un nom suivi de la liste des attributs, chaque attribut étant associé à un type. La syntaxe est donc :

$$R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$$

où les A_i sont les noms d'attributs et les D_i les types.

On peut trouver dans un schéma de relation plusieurs fois le même type, mais une seule fois un même nom d'attribut. Le type peut être omis en phase de définition. Il est d'ailleurs reltivement facile de changer le type d'un attribut sur une base existante.

Instance d'une relation

Une *instance d'une relation R*, ou simplement *relation* se définit mathématiquement comme un sous-ensemble *fini* du produit cartésien des domaines des attributs de R. Rappelons que le produit cartésien $D_1 \times \ldots \times D_n$ entre des domaines $D_1, \ldots, D_n$ est l'ensemble de *tous* les nuplets $(v_1, \ldots, v_n), v_i \in D_i$.

Un des fondements du modèle relationnel est la théorie des ensembles et la notion de relation dans le modèle correspond strictement au concept mathématique dans cette théorie. Une (instance de) relation se représente sous forme de *table*, et on emploie le plus souvent ces deux termes comme des synonymes.

La définition d'une table comme un ensemble (au sens mathématique) a quelques conséquences importantes :

- *l'ordre des lignes est indifférent* car il n'y a pas d'ordre dans un ensemble ; conséquence pratique : le résultat d'une requête appliquée à une table ne dépend pas de l'ordre des lignes dans la table ;
- *on ne peut pas trouver deux fois la même ligne* car il n'y a pas de doublons dans un ensemble ;
- il n'y a pas de "cellule vide" dans la table, donc toutes les valeurs de tous les attributs sont toujours connues.

Dans la pratique les choses sont un peu différentes pour les doublons et les cellules vides, comme nous le verrons plus loin.

Clé d'une table

Une clé d'une table est un sous-ensemble des attributs qui permet d'identifier chaque ligne de manière unique (cf. le chapitre *Conception d'une base de données* (page 5)). Comme on a vu que deux lignes sont toujours différentes, l'ensemble de tous les attributs est lui-même une clé mais il est indispensable de trouver au moins un sous-ensemble satisfaisant la condition. Le choix de la clé, qui s'effectue au moment de la conception, est très important pour la qualité du schéma. Choisir d'identifier un film par son titre comme nous l'avons envisagé dans l'exemple précédent n'est pas un très bon choix. Reportez-vous sur ce sujet dans le chapitre *Conception d'une base de données* (page 5) consacré à la modélisation.

nuplets

Un *nuplet* d'une relation $R(D_1, \ldots, D_n)$ est une liste de *n* valeurs $(v_1, \ldots, v_n)$ où chaque v_i est la valeur d'un attribut A_i de type D_i. Exemple :

```
('Cyrano', 1992, 'Rappeneau')
```

Un nuplet est représenté par une ligne dans une relation sous forme de table. En principe, on connaît les valeurs de tous les attributs du nuplet.

Note : La terminologie "nuplet" vient directement de celle utilisée en théorie des ensembles. En pratique, on utilisera couramment le terme "ligne", et parfois "tuple" qui n'est pas reconnu par l'Académie.

Base de données

Une (instance de) base de données est un ensemble fini (d'instances) de relations. Le schéma de la base est l'ensemble des schémas des relations de cette base.

La création d'un schéma de base de données est simple une fois que l'on a déterminé toutes les tables qui constituent la base. En revanche le choix de ces tables est un problème difficile car il détermine en grande partie les caractéristiques et qualités de la base : performances, exactitude, exhaustivité, disponibilité des informations, etc. En pratique, on définit le schéma à l'aide d'un modèle de données *conceptuel*, puis on le transcrit en schéma relationnel. La technique la plus répandue consiste à partir d'un schéma Entité/Association. La section suivante donne les règles du processus de transformation, en s'appuyant sur l'exemple du schéma E/A, de la figure *Le schéma E/A des films* (page 27), discuté dans le chapitre *Conception d'une base de données* (page 5).

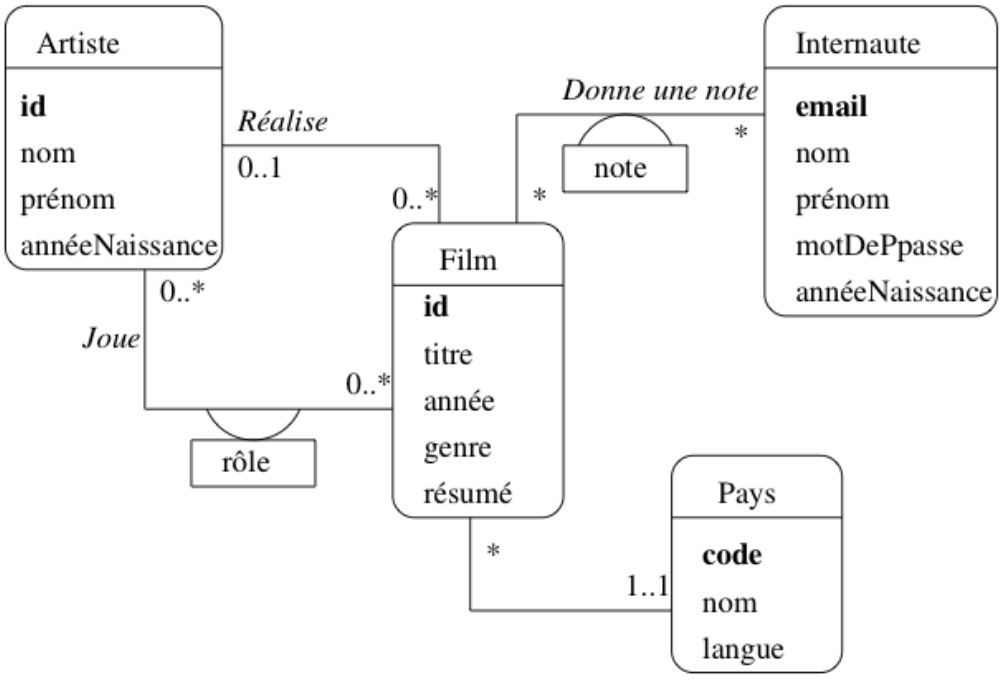

Fig. 3.1 – Le schéma E/A des films

3.2 Passage d'un schéma E/A à un schéma relationnel

On passe donc d'un modèle disposant de deux structures (entités et associations) à un modèle disposant d'une seule (tables). Logiquement, entités et associations seront donc toutes deux transformées en tables. La subtilité réside en fait dans la nécessité de préserver les *liens* existant explicitement dans un schéma E/A et qui semblent manquer dans le modèle relationnel. Dans ce dernier cas on utilise en fait un mécanisme de référence par valeur basé sur les *clés* des tables.

3.2.1 Types d'entités

Le schéma d'une table est constitué du nom de la table, suivi de la liste des attributs. Alors, pour chaque type d'entité du schéma E/A, on obtient le schéma d'une table :

— La table a le même nom que l'entité.
— Chaque propriété du type d'entité, y compris l'identifiant, devient un attribut de la table.
— L'identition est la clé de la table.

À partir du schéma E/A de la figure *Le schéma E/A des films* (page 27), on obtient les schémas de table suivants. La clé primaire est en gras.

— Film (**idFilm**, titre, année, genre, résumé)
— Artiste (**idArtiste**, nom, prénom, annéeNaissance)
— Internaute (**email**, nom, prénom, région)
— Pays (**code**, nom, langue)

Chaque entité sera donc représentée par une ligne (nuplet) dans la table. On peut remarquer que l'on a perdu pour l'instant tout lien entre les tables.

3.2.2 Associations de un à plusieurs

Soit une association de un à plusieurs (l s'agit ici des cardinalités *maximales*). Le passage au modèle relationnel suit les règles suivantes :

— On crée les tables R_A et R_B correspondant respectivement aux entités A et B, appliquant la première règle.
— L'identifiant de B devient un attribut de R_A.

L'idée est qu'une ligne (entité) de A *référence* la ligne (entité) de B qui lui est associée à l'aide d'un attribut qui contient la valeur de la clé de B. Cet attribut est la *clé étrangère* de B dans A.

Voici le schéma obtenu pour représenter l'association entre les types d'entité *Film*, *Artiste* et *Pays*. Les clés étrangères sont en italiques.

— Film (**idFilm**, titre, année, genre, résumé, *idArtiste*, *codePays*)
— Artiste (**idArtiste**, nom, prénom, annéeNaissance)
— Pays (**code**, nom, langue)

Comment nommer la clé étrangère ? Ici nous avons adopté une convention simple en concaténant `id` et le nom de la table référencée. On peut souvent faire mieux. Par exemple, dans le schéma de la table *Film*, le rôle précis tenu par l'artiste référencé dans l'association n'est pas induit par le nom `idArtiste`. L'artiste dans *Film* a un rôle de metteur en scène, mais il pourrait tout aussi bien s'agir du décorateur ou de l'accessoiriste : rien dans le nom de l'attribut ne le précise

On peut donner un nom plus explicite à l'attribut. Il n'est pas du tout obligatoire en fait que les attributs constituant une clé étrangère aient le même nom que ceux de le clé primaire auxquels ils se réfèrent. Voici le schéma de la table *Film*, dans lequel la clé étrangère pour le metteur en scène est nommée `idRéalisateur`.

— Film (**idFilm**, titre, année, genre, résumé, *idRéalisateur*)

Les tables ci-dessous montrent un exemple de la représentation des associations entre *Film* et *Artiste* d'une part, *Film* et *Pays* d'autre part (on a omis le résumé du film). Noter que l'on ne peut avoir qu'un artiste dont l'`id` est 102 dans la table *Artiste*, puisque l'attribut `idArtiste` ne peut prendre qu'une valeur. Cela correspond à la contrainte, identifiée pendant la conception et modélisée dans le schéma E/A de la figure *Le schéma E/A des films* (page 27), qu'un film n'a qu'un seul réalisateur.

En revanche rien n'empêche cet artiste 102 de figurer plusieurs fois dans la colonne `idRéalisateur` de la table *Film* puisqu'il n'y a aucune contrainte d'unicité sur cet attribut. On a donc bien l'équivalent de l'association un à plusieurs élaborée dans le schéma E/A.

id	nom	prénom	année
101	Scott	Ridley	1943
102	Hitchcock	Alfred	1899
103	Kurosawa	Akira	1910
104	Woo	John	1946
105	Tarantino	Quentin	1963
106	Cameron	James	1954
107	Tarkovski	Andrei	1932

Et voici la table des films. Remarquez que chaque valeur de la colonne `idRéalisateur` est l'identifiant d'un artiste.

id	titre	année	genre	idRéalisateur	codePays
1	Alien	1979	Science-Fiction	101	USA
2	Vertigo	1958	Suspense	102	USA
3	Psychose	1960	Suspense	102	USA
4	Kagemusha	1980	Drame	103	JP
5	Volte-face	1997	Policier	104	USA
6	Pulp Fiction	1995	Policier	105	USA
7	Titanic	1997	Drame	106	USA
8	Sacrifice	1986	Drame	107	FR

Note : les valeurs des clés primaires et étrangères sont complètement indépendantes l'une de l'autre. Nous avons identifié les films en partant de 1 et les artistes en partant de 101 pour des raisons de clarté, mais en pratique rien n'empêche de trouver une ligne comme :

(63, Gravity, 2014, SF, 63, USA)

Il n'y a pas d'ambiguité : le premier '63' est l'identifiant du film, le second est l'identifiant du réalisateur.

Et voici, pour compléter, la table des pays.

code	nom	langue
USA	Etats Unis	anglais
FR	France	français
JP	Japon	japonais

3.2.3 Associations avec type d'entité faible

Une entité faible est toujours identifiée par rapport à une autre entité. C'est le cas par exemple de l'association entre *Cinéma* et *Salle* (voir chapitre *Conception d'une base de données* (page 5)). Cette association est de type "un à plusieurs" car l'entité faible (une salle) est liée à une seule autre entité (un cinéma) alors que, en revanche, un cinéma peut être lié à plusieurs salles.

Le passage à un schéma relationnel est donc identique à celui d'une association 1-*n* classique. On utilise un mécanisme de clé étrangère pour référencer l'entité forte dans l'entité faible. La seule nuance est que la clé étrangère est une partie de l'identifiant de l'entité faible.

Regardons notre exemple pour bien comprendre. Voici le schéma obtenu pour représenter l'association entre les types d'entité *Cinéma* et *Salle*.

— Cinéma (**id**, nom, numéro, rue, ville)
— Salle (**idCinéma, no**, capacité)

On note que l'identifiant d'une salle est constitué de l'identifiant du cinéma et d'un numéro complémentaire permettant de distinguer les salles au sein d'un même cinéma. Mais l'identifiant du cinéma dans *Salle* est *aussi* une clé étrangère référençant une ligne de la table *Cinéma*. En d'autres termes, la clé étrangère est une partie de la clé primaire.

Cette modélisation simplifie l'attribution de l'identifiant à une nouvelle entité *Salle* puisqu'il suffit de reprendre l'identifiant du composé (le cinéma) et de numéroter les composants (les salles) relativement au composé. Il ne s'agit pas d'une différence vraiment fondamentale avec les associations *1-n* mais elle peut clarifier le schéma.

3.2.4 Associations binaires de plusieurs à plusieurs

Soit une association binaire *n-n* entre A et B. La transformation en un schéma relationnel se fait de la manière suivante :

- On crée les tables R_A* et R_B* correspondant respectivement aux entités A et B.
- On crée une table R_{AB} pour l'association. La clé de R_A* et la clé de R_B* deviennent des clés étrangères de R_{AB}.
- La clé de cette table est la concaténation des clés des tables R_A et R_B.
- Les propriétés de l'association deviennent des attributs de R_{AB}.

Toujours à partir du schéma des films, on obtient la table *Rôle* représentant l'association entre les films et les acteurs. Par application des règles, on obtient :

- Film (**idFilm**, titre, année, genre, résumé, *idRéalisateur*, *codePays*)
- Artiste (**idArtiste**, nom, prénom, annéeNaissance)
- Role (**idFilm, idActeur**, nomRôle)

De même, on obtient une table *Notation* pour représenter l'association entre un internaute et les films qu'il a notés.

- Film (**idFilm**, titre, année, genre, résumé, *idRéalisateur*, *codePays*)
- Internaute (**email**, nom, prénom, région)
- Notation (**email, idFilm**, note)

Pour bien comprendre le mécanisme de representation des entités et associations grâce aux clés primaires et étrangères, examinons les tables suivantes montrant un exemple de représentation de *Rôle*. On peut constater le mécanisme de référence unique obtenu grâce aux clés des tables. Chaque rôle correspond à un unique acteur et à un unique film. De plus on ne peut pas trouver deux fois la même paire `(idFilm,idActeur)` dans cette table (c'est un choix de conception qui découle du schéma E/A sur lequel nous nous basons). En revanche un même acteur peut figurer plusieurs fois (mais pas associé au même film), ainsi qu'un même film (mais pas associé au même acteur).

Voici tout d'abord la table des films.

id	titre	année	genre	idRéalisateur	codePays
20	Impitoyable	1992	Western	130	USA
21	Ennemi d'état	1998	Action	132	USA

Puis la table des artistes.

id	nom	prénom	année
130	Eastwood	Clint	1930
131	Hackman	Gene	1930
132	Scott	Tony	1930
133	Smith	Will	1968

En voici enfin la table des rôles, qui consiste ensentiellement en identifiants établissant des liens avec les deux tables précédentes. À vous de les décrypter pour comprendre comment toute l'information est représentée, et conforme aux choix de conception issus du schéma E/A. Que peut-on dire de l'artiste 130 par exemple ? Peut-on savoir dans quels films joue Gene Hackman ? Qui a mis en scène *Impitoyable* ?

idFilm	idArtiste	nomRôle
20	130	William Munny
20	131	Little Bill
21	131	Bril
21	133	Robert Dean

On peut donc remarquer que chaque partie de la clé de la table *Rôle* est elle-même une clé étrangère qui fait référence à une ligne dans une autre table :

— l'attribut `idFilm` fait référence à une ligne de la table *Film* (un film) ;
— l'attribut `idActeur` fait référence à une ligne de la table *Artiste* (un acteur) ;

Le même principe de référencement et d'identification des tables s'applique à la table *Notation*. Il faut bien noter que, par choix de conception, on a interdit qu'un internaute puisse noter plusieurs fois le même film, de même qu'un acteur ne peut pas jouer plusieurs fois dans un même film. Ces contraintes ne constituent pas des limitations, mais des décisions prises au moment de la conception sur ce qui est autorisé, et sur ce qui ne l'est pas.

Le processus de conception détaillé ci-dessus permet de décomposer toutes les informations d'une base de données en plusieurs tables dont chacune stocke un des ensembles d'entités. Les liens sont définis par un mécanisme de référencement basé sur les clés primaires et les clés étrangères. Ce mécanisme est à la fois simple et puissant, et bien le comprendre est nécessaire pour maîtriser la construction de bases de données qui ne demanderont par de réorganisation – nécessairement douloureuse – par la suite.

3.2.5 Associations ternaires

Dans le cas d'associations impliquant plus de deux entités, on atteint une des limites du modèle Entité/Association en matière de spécification de contraintes. En première approche, on peut appliquer la règle énoncée précédemment pour les associations binaires et la généraliser. On obtiendrait alors, pour l'association *Séance* :

— Cinéma (**idCinéma**, nomCinéma, numéro, rue, ville)
— Salle (**idCinéma, no**, capacité)
— Film (**idFilm**, titre, année, genre, résumé, *idRéalisateur*, *codePays*)
— Horaire (**idHoraire**, heureDébut, heureFin)
— Séance (**idFilm, idCinéma, noSalle, idHoraire**, tarif)

La table *Séance* a pour clé la concaténation des identifiants de chacune des entités composantes, ce qui donne une clé d'une taille assez importante, avec 4 attributs. La contrainte d'unicité portant sur l'ensemble des attributs, on s'aperçoit que rien n'interdit qu'une même salle présente deux films différents au même horaire. Si on souhaite éviter cette situation, la clé devient **(idCinéma, noSalle, idHoraire)**, et on ne respecte plus la règle de passage du schéma E/A vers le schéma relationnel.

En d'autres termes, en cas d'association entre plus de 2 entités, la clé de la table représentant l'association est un sous-ensemble de la concaténation des clés. Il faut se poser soigneusement la question de la (ou des) clé(s) au moment de la création de la table car elle(s) ne peu(ven)t plus être déduite(s) du schéma E/A. On parle parfois de *clé candidate*. On peut trouver plus de deux clés candidates : c'est le cas par exemple si on souhaite qu'un même cinéma ne présente pas deux fois le même film dans des salles différentes.

Ce type de contrainte est exprimable dans un schéma relationnel (voir, plus loi, la clause `unique`). Les exemples qui précèdent montrent que l'association de plus de deux entités introduit une liberté d'interprétation et de représentation des données qui peut se faire au dépend de leur cohérence. Il est en fait très recommandé de ne représenter dans le schéma E/A que des associations binaires, ce qui peut se faire dans le cas qui précède en transformant l'association ternaire en une entité *Séance* dotée de son propre identifiant. La plupart des modèles de données intégrés comme outils de conception aux SGBD ne permettent que des associations binaires.

3.2.6 Retour sur le choix des identifiants

Il est maintenant largement admis que la clé d'une table doit être un identifiant "neutre" contenant un entier auto-incrémenté à chaque insertion. En effet :

— La valeur de la clé doit être connue au moment de l'insertion.
— Chaque valeur de l'identifiant doit caractériser de manière unique une occurrence, ; le titre pour la table *Film* ou le nom pour la table *Acteur* ne sont clairement pas des bons choix.
— Si on utilise un *ensemble* de propriétés comme identifiant, la référence à une entité devient très lourde ; par exemple, la clé de *Cinéma* pourraît être **(nom, rue, ville)**.

— L'identifiant sert de référence externe *et ne doit donc jamais être modifié* (il faudrait répercuter les mises à jour sur les valeurs des clés étrangères, ce qui est très compliqué à garantir).

Il s'avère, sauf cas particulier, impossible de trouver une propriété dans une entité qui satisfait toutes ces conditions. Si on a une table *Personne*, par exemple, on peut mener la discussion suivante :

— Le numéro de sécurité sociale ? Oui, il identifie uniquement une personne ; est-ce qu'il ne change jamais ? C'est moins clair : que se passe-t-il si on a fait une erreur à la saisie par exemple. Est-il toujours connu au moment de l'insertion de l'entité ? Là, il semble très aventureux de répondre "oui" : quid des étrangers, des mineurs, de cas où le no SS est temporairement inconnu ? Ce n'est certainement pas un bon choix.

— L'adresse électronique (email) ? Oui c'est un identifiant unique, mais est-ce qu'il ne peut pas passer d'une personne à une autre ? En tout il est clair que les personnes changent d'email, ce qui rompt une des conditions ci-dessus.

En conclusion, la seule méthode dont on peut être sûr qu'elle ne présente pas de risque est l'ajout d'un attribut `id`, de type entier, dans lequel on place une valeur unique générée par le programme d'insertion.

Ce choix ne régle pas le risque de redondance (la même entité présente plusieurs fois) et même, d'une certaine manière, la maximalise. En effet, puisque l'identifiant est généré, une insertion répétée d'une même entité engendre de nouveaux identifiants sans provoquer de rejet. La redondance doit être gérée, mais indépendamment du choix de l'identifiant, par des *contraintes d'unicité* qui sont exprimables dans le langage de création des tables.

3.3 Définition d'un schéma SQL

Cette section présente le *langage de définition de données* (LDD) qui permet de spécifier le schéma d'une base de données relationnelle. Ce langage correspond à une partie de la norme SQL (*structured query language*), l'autre partie étant relative à la *manipulation des données* (LMD).

La définition d'un schéma comprend essentiellement deux parties : d'une part la description des *tables* et de leur contenu, d'autre part les *contraintes* qui portent sur les données de la base. La spécification des contraintes est souvent placée au second plan bien qu'elle soit en fait très importante : elle permet d'assurer, *au niveau de la base*, des contrôles sur l'intégrité des donnés qui s'imposent à toutes les applications accédant à cette base. Un dernier aspect de la définition d'un schéma, rapidement survolé ici, est la description de la représentation dite "physique", celle qui décrit l'organisation des données. Il est toujours possible de réorganiser une base, et on peut donc tout à fait adopter initialement l'organisation choisie par défaut pour le système.

3.3.1 Types SQL

La norme SQL ANSI propose un ensemble de types dont les principaux sont donnés dans le tableau ci-dessous. Ce tableau présente également la taille, en octets, des instances de chaque type, cette taille n'étant ici qu'à titre indicatif car elle peut varier selon les systèmes.

Type	Description	Taille
integer	Type des entiers relatifs	4 octets
smallint	idem	2 octets
bigint	idem	8 octets
float	Flottants simple précision	4 octets
double	Flottants double précision	8 octets
real	Flottant simple ou double	8 octets
numeric (*M, D*)	Numérique avec précision fixe.	*M* octets
decimal(*M, D*)	Idem.	*M* octets
char(*M*)	Chaînes de longueur fixe	*M* octets
varchar*(M*)	Chaînes de longueur variable	*L+1* avec $L \leq M$
bit varying	Chaînes d'octets	Longueur de la chaîne.
date	Date (jour, mois, an)	env. 4 octets
time	Horaire (heure, minutes, secondes)	env. 4 octets
datetime	Date et heure	8 octets
year	Année	2 octets

Types numériques exacts

La norme SQL ANSI distingue deux catégories d'attributs numériques : les *numériques exacts*, et les *numériques flottants*. Les types de la première catégorie (essentiellement `integer` et `decimal`) permettent de spécifier la précision souhaitée pour un attribut numérique, et donc de représenter une valeur exacte. Les numériques flottants correspondent aux types couramment utilisés en programmation (`float`, `double`) et ne représentent une valeur qu'avec une précision limitée.

Le type `integer` permet de stocker des entiers, sur 4 octets en général. Il existe deux variantes du type `integer` : `smallint` et `bigint`. Ces types diffèrent par la taille utilisée pour le stockage : voir le tableau des types SQL.

Le type `decimal(M,D)` correspond à un numérique de taille maximale `M`, avec un nombre de décimales fixé à `D`. `numeric` est un synonyme de `decimal`. Ces types sont surtout utiles pour manipuler des valeurs dont la précision est connue, comme les valeurs monétaires. Afin de préserver cette précision, les instances de ces types sont stockées comme des chaînes de caractères.

Types numériques flottants

Ces types s'appuient sur la représentation des numériques flottants propre à la machine, en simple ou double précision. Leur utilisation est donc analogue à celle que l'on peut en faire dans un langage de programmation comme le C.

— Le type `float` correspond aux flottants en simple précision.
— Le type `double precision` correspond aux flottants en double précision ; le raccourci `double` est accepté.

Caractères et chaînes de caractères

Les deux types principaux de la norme ANSI sont `char` et `varchar`. Ces deux types permettent de stocker des chaînes de caractères d'une taille maximale fixée par le paramètre `M`. Les syntaxes sont identiques. Pour le premier, `char(M)`, et `varchar(M)` pour le second. La différence essentielle est qu'une valeur `char` a une taille fixée, et se trouve donc complétée avec des blancs si sa taille est inférieure à `M`. En revanche une valeur `varchar` a une taille variable et est tronquée après le dernier caractère non blanc.

Quand on veut stocker des chaînes de caractères longues (des textes, voire des livres), dont la taille dépasse, typiquement, 255 caractères, le type `varchar` ne suffit plus. La norme SQL propose un type `bit varying` qui correspond à de très longues chaînes de caractères. Souvent les systèmes proposent des variantes de ce type sous le nom `text` ou `blob` (pour *Binary Long Object*).

Dates

Un attribut de type `date` stocke les informations jour, mois et année (sur 4 chiffres). La représentation interne n'est pas spécifiée par la norme. Tous les systèmes proposent de nombreuses opérations de conversion (non normalisées) qui permettent d'obtenir un format d'affichage quelconque.

Un attribut de type `time` représente un horaire avec une précision à la seconde. Le type `datetime` permet de combiner une date et un horaire.

3.3.2 Création des tables

D'une manière générale, les objects du schéma sont créés avec `create`, modifiés avec `alter` et détruits avec `drop`, alors que les données, instances du schéma sont créées, modifiées et détruites avec, respectivement, `insert`, `update` et `delete`.

Voici un premier exemple avec la commande de création de la table *Internaute*.

```
create table Internaute (email varchar (40) not null,
                         nom varchar (30) not null ,
                         prenom varchar (30) not null,
                         region varchar (30),
                         primary key (email));
```

La syntaxe se comprend aisément. La seule difficulté est de choisir correctement le type de chaque attribut.

Conventions : noms des tables, des attributs, mots-clé SQL

On dispose, comme dans un langage de programmation, d'une certaine liberté. La seule recommandation est d'être cohérent pour des raisons de lisibilité. D'une manière générale, SQL n'est pas sensible à la casse. Quelques propositions :

- Le nom des tables devrait commencer par une majuscule, le nom des attributs par une minuscule ;
- quand un nom d'attribut est constitué de plusieurs mots, on peut soit les séparer par des caractères '_', soit employer la convention *CamelCase* : minuscule au premier mot, majuscule aux suivants. Exemple : `mot_de_passe` ou `motDePasse`.
- Majuscule ou minuscule pour les mots-clé SQL ? Quand on inclut une commande SQL dans un langage de programmation, il est sans doute plus lisible d'utiliser des majuscules pour les mots-clé.

Les accents et caractères diacritiques sont-ils acceptés ? En principe oui, cela dépend des systèmes. Mais en pratique le codage au-delà de l'ASCII est toujours susceptible de soulever des problèmes (par exemple pour interroger une base dont le schéma contient des lettres accentuées, avec un clavier QWERTY). Il vaut sans doute mieux malheureusement l'éviter.

Le `not null` dans la création de table *Internaute* indique que l'attribut correspondant doit *toujours* avoir une valeur. Il s'agit d'une différence importante entre la pratique et la théorie : on admet que certains attributs peuvent ne pas avoir de valeur, ce qui est très différent d'une chaîne vide ou de 0. Quand on parle de valeur `null` en SQL, il s'agit en fait d'une *absence* de valeur. En conséquence :

- on ne peut pas faire d'opération incluant un `null` ;
- on ne peut pas faire de comparaison avec un `null`.

L'absence de valeur a des conséquences parfois déroutantes. Il est préférable d'ajouter la contrainte `not null` quand c'est pertinent : cela renforce la qualité de la base et facilite le travail des applications par la suite. L'option suivante permet ainsi de garantir que tout internaute a un mot de passe.

```
motDePasse  varchar(60) not null
```

Le SGBD rejettera alors toute tentative d'insérer une ligne dans *Internaute* sans donner de mot de passe.

Important : La clé primaire doit *toujours* être déclarée `not null`.

Une autre manière de forcer un attribut à toujours prendre une valeur est de spécifier une *valeur par défaut* avec l'option `default`.

```
create table Cinema (id integer not null,
                     nom varchar (30) not null ,
                     adresse varchar(255) default 'Inconnue'
                     primary key (id));
```

Quand on insérera une ligne dans la table *Cinéma* sans indiquer d'adresse, le système affectera automatiquement la valeur `'Inconnue'` à cet attribut. En général on utilise comme valeur par défaut une constante, sauf pour quelques variables fournies par le système (par exemple `sysdate` pour indiquer la date courante).

3.3.3 Contraintes

La création d'une table telle qu'on l'a vue précédemment est extrêmement sommaire car elle n'indique que le contenu de la table sans spécifier les contraintes que doit respecter ce contenu. Or il y a *toujours* des contraintes et il est indispensable de les inclure dans le schéma pour assurer (dans la mesure du possible) l'intégrité de la base.

Voici les règles (ou *contraintes d'intégrité*) que l'on peut demander au système de garantir :

- La valeur d'un attribut doit être unique au sein de la table.
- Un attribut doit toujours avoir une valeur. C'est la contrainte `not null` vue précédemment.
- Un attribut (ou un ensemble d'attributs) constitue(nt) la clé de la table.
- Un attribut dans une table est liée à la clé primaire d'une autre table (*intégrité référentielle*).
- Enfin toute règle s'appliquant à la valeur d'un attribut (min et max par exemple).

Les contraintes sur les clés doivent être systématiquement spécifiées. La dernière (clause `check`) s'appuie en grande partie sur la connaissance du langage d'interrogation de SQL et sera vue ultérieurement.

Clés d'une table

Une *clé* est un attribut (ou un ensemble d'attributs) qui identifie(nt) de manière unique un nuplet d'une table. Il peut y avoir plusieurs clés mais l'une d'entre elles doit être choisie comme *clé primaire*. Ce choix est important : la clé primaire est la clé utilisée pour référencer une ligne et une seule à partir d'autres tables. Il est donc très délicat de la remettre en cause après coup. En revanche les clés secondaires peuvent être créées ou supprimées beaucoup plus facilement.

La clé primaire est spécifiée avec l'option `primary key`.

```
create table Pays (code varchar(4) not null,
                   nom  varchar (30) default 'Inconnu' not null,
                   langue varchar (30) not null,
                   primary key (code));
```

Il doit *toujours* y avoir une clé primaire dans une table afin de pouvoir désigner une ligne de manière univoque. Une clé peut être constituée de plusieurs attributs :

```
create table Notation (idFilm integer not null,
                       email  varchar (40) not null,
                       note  integer not null,
                       primary key (idFilm, email));
```

Tous les attributs figurant dans une clé doivent être déclarés `not null`. Cela n'a pas vraiment de sens en effet d'identifier des lignes par des valeurs absentes.

Comme nous l'avons déjà expliqué à plusieurs reprises, la méthode recommandée pour gérer la clé primaire est d'utiliser un attribut `id`, sans aucune signification particulière autre que celle de contenir la valeur unique identifiant une ligne. Voici un exemple typique :

```
create table Artiste (id integer not null,
                      nom varchar (30) not null,
                      prenom varchar (30) not null,
                      anneeNaiss integer,
                      primary key (id))
```

La valeur de cet identifiant peut même est automatiquement engendrée à chaque insertion, ce qui soulage d'avoir à implanter un mécanisme de génération d'identifiant. La méthode varie d'un système à l'autre, et repose de manière générale sur la notion de *séquence*. Voici la syntaxe MySQL pour indiquer qu'une clé est auto-incrémentée.

```
create table Artiste (id integer not null auto increment,
                     ...,
                      primary key (id))
```

L'utilisation d'un identifiant artificiel n'apporte rien pour le contrôle des redondances. Il est possible d'insérer des centaines de lignes dans la table *Artiste* ci-dessus ayant toutes exactement les mêmes valeurs, et ne différant que par la clé.

Les contraintes empêchant la redondance (et plus généralement assurant la cohérence d'une base) sont spécifiées indépendamment de la clé par la clause `unique`. On peut par exemple indiquer que deux artistes distincts ne peuvent avoir les mêmes nom et prénom.

```
create table Artiste  (idArtiste integer not null auto increment,
                      nom varchar (30) not null,
                      prenom varchar (30) not null,
                      anneeNaiss integer,
                      primary key (idArtiste),
                      unique (nom, prenom))
```

Il est facile de supprimer cette contrainte (dite de "clé secondaire") par la suite. Ce serait beaucoup plus difficile si on avait utilisé la paire `(nom,prenom)` comme clé primaire puisqu'elle serait alors utilisée pour référencer un artiste dans d'autres tables.

La clause `unique` ne s'applique pas aux valeurs `null`.

3.3.4 Clés étrangères

SQL permet d'indiquer quelles sont les clés étrangères dans une table, autrement dit, quels sont les attributs qui font référence à une ligne dans une autre table. On peut spécifier les clés étrangères avec l'option foreign key.

```
create table Film  (idFilm integer not null,
                    titre    varchar (50) not null,
                    annee    integer not null,
                    idRealisateur    integer not null,
                    genre varchar (20) not null,
                    resume      varchar(255),
                    codePays    varchar (4),
                    primary key (idFilm),
                    foreign key (idRealisateur) references Artiste,
                    foreign key (codePays) references Pays);
```

La commande

```
foreign key (idRealisateur) references Artiste,
```

indique que `idRealisateur` référence la clé primaire de la table *Artiste*. Le SGBD vérifiera alors, pour toute modification pouvant affecter le lien entre les deux tables, que la valeur de `idRealisateur` correspond bien à une ligne de *Artiste*. Ces modifications sont :

— l'insertion dans *Film* avec une valeur inconnue pour `idRealisateur` ;
— la destruction d'un artiste ;
— la modification de `id` dans *Artiste* ou de `idRealisateur` dans *Film*.

En d'autres termes on a la garantie que le lien entre *Film* et *Artiste* est *toujours* valide. Cette contrainte est importante pour s'assurer qu'il n'y a pas de fausse référence dans la base, par exemple qu'un film ne fait pas référence à un artiste qui n'existe pas. Il est beaucoup plus confortable d'écrire une application par la suite quand on sait que les informations sont bien là où elles doivent être.

Il faut noter que l'attribut `codePays` *n'est pas déclaré ''not null'*, ce qui signifie que l'on s'autorise à ne pas connaître le pays de production d'un film. Quand un attribut est à `null`, la contrainte d'intégrité référentielle ne s'applique pas. En revanche, on impose de connaître le réalisateur d'un film. C'est une contrainte forte, qui d'un côté améliore la richesse et la cohérence de la base, mais de l'autre empêche toute insertion, même provisoire, d'un film dont le metteur en scène est inconnu. Ces deux situations correspondent respectivement aux associations 0..* et 1..* dans la modélisation entité/association.

Note : On peut facilement passer un attribut de `not null` à `null`. L'inverse n'est pas vrai s'il existe déjà des valeurs à `null` dans la base.

Que se passe-t-il quand la violation d'une contrainte d'intégrité est détectée par le système ? Par défaut, la mise à jour est rejetée, mais il est possible de demander la répercussion de cette mise à jour de manière à ce que la contrainte soit respectée. Les événements que l'on peut répercuter sont la modification ou la destruction de la ligne référencée, et on les désigne par `on update` et `on delete` respectivement. La répercussion elle-même consiste soit à mettre la clé étrangère à `null` (option `set null`), soit à appliquer la même opération aux lignes de l'entité composante (option `cascade`).

Voici comment on indique que la destruction d'un pays déclenche la mise à `null` de la clé étrangère `codePays` pour tous les films de ce pays.

```
create table Film  (idFilm integer not null,
                    titre    varchar (50) not null,
                    annee    integer not null,
                    idRealisateur    integer not null,
                    genre varchar (20) not null,
                    resume      varchar(255),
                    codePays    varchar (4),
                    primary key (idFilm),
                    foreign key (idRealisateur) references Artiste,
                    foreign key (codePays) references Pays
                       on delete set null)
```

Dans le cas d'une entité faible, on décide en général de détruire le *composant* quand on détruit le *composé*. Par exemple, quand on détruit un cinéma, on veut également détruire les salles ; quand on modifie la clé d'un cinéma, on veut répercuter la modification sur ses salles. Dans ce cas c'est l'option `cascade` qui s'impose.

```
create table Salle  (idCinema integer not null,
                     no    integer not null,
                     capacite    integer not null,
                     primary key (idCinema, noSalle),
                     foreign key (idCinema) references Cinema
                       on delete cascade,
                       on update cascade)
```

L'attribut `idCinema` fait partie de la clé et ne peut donc pas être `null`. On ne pourrait donc pas spécifier ici `on delete set null`.

La spécification des actions `on delete` et `on update` simplifie la gestion de la base par la suite : on n'a plus par exemple à se soucier de détruire les salles quand on détruit un cinéma.

3.3.5 La clause `check`

La clause `check` exprime des contraintes portant soit sur un attribut, soit sur une ligne. La condition elle-même peut être toute expression suivant la clause `where` dans une requête SQL. Les contraintes les plus courantes sont celles consistant à restreindre un attribut à un ensemble de valeurs, comme expliqué ci-dessous. On peut trouver des contraintes arbitrairement complexes, faisant référence à d'autres tables. Nous reviendrons sur cet aspect après avoir étudié le langage d'interrogation SQL.

Voici un exemple simple qui restreint les valeurs possibles des attributs `annee` et `genre` dans la table *Film*.

```
create table Film  (idFilm integer not null,
                    titre    varchar (50) not null,
                    annee    integer not null
                     check (annee between 1890 and 2063) not null,
                    idRealisateur    integer,
                    genre varchar (20) not null
                      check (genre in ('Histoire','Western','Drame')),
                    resume      varchar(255),
                    codePays    varchar (4),
                    primary key (idFilm),
                    foreign key (idRealisateur) references Artiste
                        on delete set null,
                    foreign key (codePays) references Pays)
```

Au moment d'une insertion dans la table *Film*, ou d'une modification de l'attribut `annee` ou `genre`, le SGBD vérifie que la valeur insérée dans `genre` appartient à l'ensemble énuméré défini par la clause `check`.

Une autre manière de définir, dans la base, l'ensemble des valeurs autorisées pour un attribut – en d'autres termes, une codification imposée – consiste à placer ces valeurs dans une table et la lier à l'attribut par une contrainte de clé étrangère. C'est ce que nous pouvons faire par exemple pour la table *Pays*.

```
create table Pays (code    varchar(4) not null,
                   nom  varchar (30) default 'Inconnu' not null,
                  langue varchar (30) not null,
                  primary key (code));
 insert into Pays (code, nom, langue) values ('FR', 'France', 'Français');
 insert into Pays (code, nom, langue) values ('USA', 'Etats Unis', 'Anglais');
 insert into Pays (code, nom, langue) values ('IT', 'Italie', 'Italien');
 insert into Pays (code, nom, langue) values ('GB', 'Royaume-Uni', 'Anglais');
 insert into Pays (code, nom, langue) values ('DE', 'Allemagne', 'Allemand');
 insert into Pays (code, nom, langue) values ('JP', 'Japon', 'Japonais');
```

Si on ne fait pas de vérification automatique, soit avec `check`, soit avec la commande `foreign key`, il faut faire cette vérification dans l'application, ce qui est plus lourd à gérer.

3.3.6 Modification du schéma

La création d'un schéma n'est qu'une première étape dans la vie d'une base de données. On est toujours amené par la suite à créer de nouvelles tables, à ajouter des attributs ou à en modifier la définition. La forme générale de la commande permettant de modifier une table est :

```
alter table <nomTable> <action> <description>
```

où `action` peut être principalement `add`, `modify`, `drop` ou `rename` et `description` est la commande de modification associée à `action`. La modification d'une table peut poser des problèmes si elle est incompatible avec le contenu existant. Par exemple passer un attribut à `not null` implique que cet attribut a déjà des valeurs pour toutes les lignes de la table.

Modification des attributs

Voici quelques exemples d'ajout et de modification d'attributs. On peut ajouter un attribut `region` à la table *Internaute* avec la commande :

```
alter table Internaute add region varchar(10)
```

S'il existe déjà des données dans la table, la valeur sera à `null` ou à la valeur par défaut. La taille de `region``étant certainement insuffisante,on peut l'agrandir avec ``modify`, et la déclarer `not null` par la même occasion :

```
alter table Internaute modify region varchar(30) not null
```

Il est également possible de diminuer la taille d'une colonne, avec le risque d'une perte d'information pour les données existantes. On peut même changer son type, pour passer par exemple de `varchar` à `integer`, avec un résultat imprévisible.

La commande `alter table` permet d'ajouter une valeur par défaut.

```
alter table Internaute add region set default 'PACA'
```

Enfin, on peut détruire un attribut avec `drop`.

```
alter table Internaute drop region
```

3.3.7 Création d'index

Pour compléter le schéma d'une table, on peut définir des *index*. Un index offre un chemin d'accès aux lignes d'une table qui est considérablement plus rapide que le balayage de cette table – du moins quand le nombre de lignes est très élevé. Les SGBD créent systématiquement un index sur la clé primaire de chaque table. Il y a deux raisons à cela ;

— l'index permet de vérifier rapidement, au moment d'une insertion, que la clé n'existe pas déjà ;
— beaucoup de requêtes SQL, notamment celles qui impliquent plusieurs tables (*jointure*), se basent sur les clés des tables pour reconstruire les liens. L'index peut alors être utilisé pour améliorer les temps de réponse.

Un index est également créé pour chaque clause `unique` utilisée dans la création de la table. On peut de plus créer d'autres index, sur un ou plusieurs attributs, si l'application utilise des critères de recherche autres que les clés primaire ou secondaires.

La commande pour créer un index est la suivante :

```
create [unique] index <nomIndex>  on <nomTable> (<attribut1> [, ...])
```

L'option `unique` indique qu'on ne peut pas trouver deux fois la même clé dans l'index. La commande ci-dessous crée un index de nom *idxNom* sur les attributs `nom` et `prenom``de la table *Artiste*. Cet index a donc une fonction équivalente à la clause ``unique` déjà utilisée dans la création de la table.

```
create unique index idxNom  on Artiste (nom, prenom)
```

On peut créer un index, cette fois non unique, sur l'attribut `genre` de la table *Film*.

```
create index idxGenre on Film (genre)
```

Cet index permettra d'exécuter très rapidement des requêtes SQL ayant comme critère de recherche le genre d'un film.

```
select * from Film where genre = 'Western'
```

Cela dit il ne faut pas créer des index à tort et à travers, car ils ont un impact négatif sur les commandes d'insertion et de destruction. À chaque fois, il faut en effet mettre à jour tous les index portant sur la table, ce qui représente un coût certain.

Pour en savoir plus sur les index, et en général sur la gestion de l'organisation des données, je vous renvoie à la seconde partie du cours disponible à http://sys.bdpedia.fr.

CHAPITRE 4

L'algèbre relationnelle

Le premier langage étudié dans ce cours est *l'algèbre relationnelle*. Elle consiste en un ensemble d'opérations qui permettent de manipuler des *relations*, considérées comme des ensembles de nuplets : on peut ainsi faire *l'union* ou la *différence* de deux relations, *sélectionner* une partie de la relation, effectuer des *produits cartésiens* ou des *projections*, etc.

Une propriété fondamentale de chaque opération est qu'elle prend une ou deux relations en entrée, et produit une relation en sortie. Cette propriété (dite de *clôture*) permet de *composer* des opérations : on peut appliquer une sélection au résultat d'un produit cartésien, puis une projection au résultat de la sélection et ainsi de suite. En fait on peut construire des *expressions algébriques* arbitrairement complexes qui permettent d'effectuer des manipulations sur un grand nombre de relations à l'aide d'un petit nombre d'opérations de base.

Une *requête* est une expression algébrique qui s'applique à un ensemble de relations (la base de données) et produit une relation finale (le résultat de la requête). On peut voir l'algèbre relationnelle comme un langage de programmation très simple qui permet d'exprimer des requêtes sur une base de données relationnelle. Les requêtes du langage SQL peuvent se transposer en expressions algébriques, ce qui nous donnera un moyen d'interpréter leur signification d'une part, et de comprendre comment les exécuter d'autre part.

Dans tout ce chapitre on va prendre l'exemple de la (petite) base de données d'un organisme de voyage. Cet organisme propose des séjours (sportifs, culturels, etc) se déroulant dans des stations de vacances. Chaque station propose un ensemble d'activités (ski, voile, tourisme). Enfin on maintient une liste des clients et des séjours auxquels ils ont participé avec leurs semaines de début et de fin.

Voici le schéma de la base. Les clés primaires sont en **gras**, les clés étrangères en *italiques*.

- Station (**id**, nom, capacité, lieu, région, tarif)
- Activité (**idStation, libellé**, prix)
- Client (**id**, nom, prénom, ville, région, solde)
- Séjour (**id**, *idClient*, *idStation*, début, fin, nbPlaces)

La table Station

Voici le contenu de la table *Station*. La clé est un code synthétisant le nom de la station.

id	nom	capacité	lieu	région	tarif
va	Venusa	350	Guadeloupe	Antilles	1200
fa	Farniente	200	Seychelles	Océan Indien	1500
sa	Santalba	150	Martinique	Antilles	2000
pa	Passac	400	Alpes	Europe	1000

La table Activité

Cette table contient les activités proposées par les stations. La clé est la paire consituée de `(idStation,libellé)`.

idStation	libellé	prix
va	Voile	150
va	Plongée	120
fa	Plongée	130
pa	Ski	200
pa	Piscine	20

La table des clients

Les clients sont identifiés par un numéro séquentiel incrémenté de 10 en 10.

id	nom	prénom	ville	région	solde
10	Fogg	Phileas	Londres	Europe	12465
20	Pascal	Blaise	Paris	Europe	6763
30	Kerouac	Jack	New York	Amérique	9812

La table des séjours

Les séjours sont identifiés par un numéro séquentiel incrémenté par unités. Le début et la fin sont des numéros de semaine dans l'année.

id	idClient	idStation	début	fin	nbPlaces
1	10	Passac	20	20	2
2	30	Santalba	21	21	5
3	20	Santalba	21	22	4
4	30	Passac	2	3	3
5	30	Venusa	19	23	3
6	20	Venusa	23	23	6
7	30	Farniente	22	24	5
8	10	Farniente	23	25	3

Nous allons découvrir comme appliquer des opérations de l'algèbre aux relations de cette base, afin de construire une nouvelle relation. Exprimer une requête, c'est simplement construire une expression combinant plusieurs opérations afin d'obtenir une relation représentant le résultat cherché.

4.1 Les opérateurs de l'algèbre

L'algèbre se compose d'un ensemble d'opérateurs, parmi lesquels 5 sont nécessaires et suffisants et permettent de définir les autres par composition. Ce sont :

- La sélection, dénotée σ ;
- La projection, dénotée π ;
- Le produit cartésien, dénoté $\times$;
- L'union, $\cup$;
- La différence, $-$.

Les deux premiers sont des opérateurs *unaires* (ils prennent en entrée une seule relation) et les autres sont des opérateurs *binaires*. À partir de ces opérateurs il est possible d'en définir d'autres, et notamment la *jointure*, $\bowtie$, qui est la composition d'un produit cartésien et d'une sélection.

Ces opérateurs sont maintenant présentés tour à tour.

4.1.1 La sélection, σ

La sélection $\sigma_F(R)$ s'applique à une relation, R, et extrait de cette relation les nuplets qui satisfont un critère de sélection, F. Ce critère peut être :

— La comparaison entre un attribut de la relation, A, et une constante a. Cette comparaison s'écrit $A\Theta a$, où Θ appartient à $\{=, <, >, \leq, \geq\}$.
— La comparaison entre deux attributs A_1 et A_2, qui s'écrit $A_1\Theta A_2$ avec les mêmes opérateurs de comparaison que précédemment.

Premier exemple : exprimer la requête qui donne toutes les stations aux Antilles.

$$\sigma_{region='Antilles'}(Station)$$

On obtient donc le résultat :

id	nom	capacité	lieu	région	tarif
va	Venusa	350	Guadeloupe	Antilles	1200
sa	Santalba	150	Martinique	Antilles	2000

La sélection a pour effet de supprimer des lignes, mais chaque ligne garde l'ensemble de ses attributs.

4.1.2 La projection, π

La projection $\pi_{A_1,A_2,\ldots,A_k}(R)$ s'applique à une relation R, et construit une relation contenant toutes les lignes de R, dans lesquelles seuls les attributs $A_1, A_2, \ldots A_k$ sont conservés. Donc, contrairement à la sélection, on ne supprime pas des lignes mais des colonnes. Par exemple : on veut le nom des stations, et leur région.

$$\pi_{nom,region}(Station)$$

On obtient le résultat suivant, après suppression des colonnes `capacité`, `lieu` et tarif} :

nom	région
Venusa	Antilles
Farniente	Océan Indien
Santalba	Antilles
Passac	Europe

On pourrait s'attendre à ce que le nombre de lignes dans le résultat soit le même que dans la relation initiale. C'est presque le cas, avec cependant une petite subtilité. Comme le résultat est une relation, il ne peut pas y avoir deux lignes identiques (il n'y a pas deux fois le même élément dans un ensemble). Il peut arriver qu'après une projection, deux lignes qui étaient distinctes initialement se retrouvent identiques, justement parce ce que l'attribut qui permettait de les distinguer a été supprimé. Dans ce cas on ne conserve qu'une seule des deux (ou plus) lignes identiques.

Exemple : on souhaite connaître toutes les régions où il y a des stations. On exprime cette requête par :

$$\pi_{region}(Station)$$

et on obtient :

région
Antilles
Océan Indien
Europe

La ligne 'Antilles' était présente deux fois dans la relation *Station*, et n'apparaît plus qu'en un seul exemplaire dans le résultat.

4.1.3 Le produit cartésien, $\times$

Le premier opérateur binaire, et le plus utilisé, est le produit cartésien, $\times$. Le produit cartésien entre deux relations R et S se note $R \times S$, et permet de créer une nouvelle relation où chaque nuplet de R est associé à chaque nuplet de S.

Voici deux relations, la première, R, contient

A	B
a	b
x	y

et la seconde, S, contient :

C	D
c	d
u	v
x	y

Et voici le résultat de $R \times S$:

A	B	C	D
a	b	c	d
a	b	u	v
a	b	x	y
x	y	c	d
x	y	u	v
x	y	x	y

Le nombre de lignes dans le résultat est exactement $|R| \times |S|$ ($|R|$ dénote le nombre de lignes dans la relation R).

En lui-même, le produit cartésien ne présente pas un grand intérêt puisqu'il associe aveuglément chaque ligne de R à chaque ligne de S. Il ne prend vraiment son sens qu'associé à l'opération de sélection, ce qui permet d'exprimer des *jointures*, opération fondamentale qui sera détaillée plus loin.

4.1.4 Renommage

Quand les schémas des relations R et S sont complètement distincts, il n'y a pas d'ambiguité sur la provenance des colonnes dans le résultat. Par exemple on sait que les valeurs de la colonne A dans $R \times S$ viennent de la relation R. Il peut arriver (il arrive de fait très souvent) que les deux relations aient des attributs qui ont le même nom. On doit alors se donner les moyens de distinguer l'origine des colonnes dans la table résultat en donnant un nom distinct à chaque attribut.

Voici par exemple une table T qui a les mêmes noms d'attributs que R.

A	B
m	n
o	p

Le schéma du résultat du produit cartésien $R \times T$ a pour schéma (A, B, A, B) et présente donc des ambiguités, avec les colonnes A et ***B*** en double.

La première solution pour lever l'ambiguité est d'adopter une convention par laquelle chaque attribut est préfixé par le nom de la table d'où il provient. Le résultat de $R \times T$ devient alors :

R.A	R.B	T.A	T.B
a	b	m	n
a	b	n	p
x	y	m	n
x	y	n	p

Cette convention pose quelques problèmes quand on crée des expressions complexes. Il existe une seconde possibilité, plus rigoureuse, pour résoudre les conflits de noms : le *renommage*. Il s'agit d'un opérateur particulier, dénoté ρ, qui permet de renommer un ou plusieurs attributs d'une relation. L'expression $\rho_{A \to C, B \to D}(T)$ permet ainsi de renommer A en C et B en D dans la relation ***T***. Le produit cartésien

$$R \times \rho_{A \to C, B \to D}(T)$$

ne présente alors plus d'ambiguités. Le renommage est une solution très générale, mais asez lourde à utiliser

Il est tout à fait possible de faire le produit cartésien d'une relation avec elle-même. Dans ce cas le renommage où l'utilisation d'un préfixe distinctif est impératif. Voici par exemple le résultat de $R \times R$, dans lequel on préfixe par $R1$ et $R2$ respectivement les attributs venant de chacune des opérandes.

R1.A	R1.B	R1.A	R2.B
a	b	a	b
a	b	x	y
x	y	a	b
x	y	x	y

4.1.5 L'union, $\cup$

Il existe deux autres opérateurs binaires, qui sont à la fois plus simples et moins fréquemment utilisés.

Le premier est l'union. L'expression $R \cup S$ crée une relation comprenant tous les nuplets existant dans l'une ou l'autre des relations R et S. Il existe une condition impérative : *les deux relations doivent avoir le même schéma*, c'est-à-dire même nombre d'attributs, mêmes noms et mêmes types.

L'union des relations $R(A, B)$ et $S(C, D)$ données en exemple ci-dessus est donc interdite (on ne saurait pas comment nommer les attributs dans le résultat). En revanche, en posant $S' = \rho_{C \to A, D \to B}(S)$, il devient possible de calculer $R \cup S'$, avec le résultat suivant :

A	B
a	b
x	y
c	d
u	v

Comme pour la projection, il faut penser à éviter les doublons. Donc le nuplet `(x,y)` qui existe à la fois dans R et dans S' ne figure qu'une seule fois dans le résultat.

4.1.6 La différence, $-$

Comme l'union, la différence s'applique à deux relations qui ont le même schéma. L'expression $R - S$ a alors pour résultat tous les nuplets de R qui ne sont pas dans S.

Voici la différence de R et S', les deux relations étant définies comme précédemment.

A	B
a	b

La différence est le seul opérateur qui permet d'exprimer des requêtes comportant une négation (on veut "*rejeter*" quelque chose, on "*ne veut pas*" des lignes ayant telle propriété). Il s'agit d'une fonctionnalité importante et difficile à manier : elle sera détaillée plus loin.

4.1.7 Jointure, $\bowtie$

Toutes les requêtes exprimables avec l'algèbre relationnelle peuvent se construire avec les 5 opérateurs présentés ci-dessus. En principe, on pourrait donc s'en contenter. En pratique, il existe d'autres opérations, très couramment utilisées, qui peuvent se contruire par composition des opérations de base. La plus importante est la jointure.

Afin de comprendre l'intérêt de cet opérateur, regardons le produit cartésien $\mathrm{Station} \times \mathrm{Activit}$.

id	nom	capacité	lieu	région	tarif	idStation	libellé	prix
va	Venusa	350	Guadeloupe	Antilles	1200	va	Voile	150
va	Venusa	350	Guadeloupe	Antilles	1200	va	Plongée	120
va	Venusa	350	Guadeloupe	Antilles	1200	fa	Plongée	130
va	Venusa	350	Guadeloupe	Antilles	1200	pa	Ski	200
va	Venusa	350	Guadeloupe	Antilles	1200	pa	Piscine	20
fa	Farniente	200	Seychelles	Océan Indien	1500	va	Voile	150
fa	Farniente	200	Seychelles	Océan Indien	1500	va	Plongée	120
fa	Farniente	200	Seychelles	Océan Indien	1500	fa	Plongée	130
fa	Farniente	200	Seychelles	Océan Indien	1500	pa	Ski	200
fa	Farniente	200	Seychelles	Océan Indien	1500	pa	Piscine	20
sa	Santalba	150	Martinique	Antilles	2000	va	Voile	150
sa	Santalba	150	Martinique	Antilles	2000	va	Plongée	120
sa	Santalba	150	Martinique	Antilles	2000	fa	Plongée	130
sa	Santalba	150	Martinique	Antilles	2000	pa	Ski	200
sa	Santalba	150	Martinique	Antilles	2000	pa	Piscine	20
pa	Passac	400	Alpes	Europe	1000	va	Voile	150
pa	Passac	400	Alpes	Europe	1000	va	Plongée	120
pa	Passac	400	Alpes	Europe	1000	fa	Plongée	130
pa	Passac	400	Alpes	Europe	1000	pa	Ski	200
pa	Passac	400	Alpes	Europe	1000	pa	Piscine	20

Le résultat comprend manifestement un grand nombre de lignes qui ne nous intéressent pas. Cela ne présente pas beaucoup de sens de rapprocher des informations sur Santalba, aux Antilles et sur l'activité de ski à Passac.

Si, en revanche, on considère le produit cartésien comme un *résultat intermédiaire*, on voit qu'il permet de se ramener au cas on on effectue, par une simple sélection, des rapprochements de lignes distinctes. Sur notre exemple, en considérant la table ci-dessus comme une table de la base, on rapproche les informations générales sur une station et la liste des activités de cette station.

La sélection qui effectue une rapprochement pertinent est la suivante :

$$\sigma_{id=idStation}(\text{Station} \times \text{Activit})$$

Prenez bien le temps de méditer cette opération de sélection : nous ne voulons conserver que les lignes de $\text{Station} \times \text{Activit}$ pour lesquelles l'identifiant de la station (provenant de *Station*) est identique à celui provenant de *Activité*. En regardant le produit cartésien ci-dessous, vous devriez pouvoir vous convaincre que cela revient à conserver les lignes qui ont un sens : chacune contient des informations sur une station et sur une activité dans cette *même* station. Si vous saisissez cette logique, vous avez fait un grand pas dans la connaissance des bases relationelles : consacrez-y le temps de réflexion nécessaire.

On obtient le résultat ci-dessous.

id	nom	capacité	lieu	région	tarif	idStation	libellé	prix
va	Venusa	350	Guadeloupe	Antilles	1200	va	Voile	150
va	Venusa	350	Guadeloupe	Antilles	1200	va	Plongée	120
fa	Farniente	200	Seychelles	Océan Indien	1500	fa	Plongée	130
pa	Passac	400	Alpes	Europe	1000	pa	Ski	200
pa	Passac	400	Alpes	Europe	1000	pa	Piscine	20

On a donc effectué une *composition* de deux opérations (un produit cartésien, une sélection) afin de rapprocher des informations réparties dans plusieurs tables, mais ayant des liens entre elles (toutes les informations dans un nuplet du résultat sont relatives à une seule station). Cette opération est une *jointure*, que l'on peut directement, et simplement, noter :

$$\text{Station} \bowtie_{\text{id=idStation}} \text{Activit}$$

La jointure consiste donc à rapprocher les lignes de deux relations pour lesquelles les valeurs d'un (ou plusieurs) attributs sont identiques. De fait, dans la plupart des cas, ces attributs communs sont (1) la clé primaire de l'une des relations et (2) la clé étrangère dans l'autre relation. Dans l'exemple ci-dessus, c'est le cas pour `id` (clé primaire de *Station*) et `idStation` (clé étrangère dans *Activité*). La jointure permet alors de *reconstruire* l'association conceptuelle entre *Station* et *Activité*.

Note : La station Santalba, qui ne propose pas d'activité, n'apparaît pas dans le résultat de la jointure. C'est normal et conforme à la définition que nous avons donnée, mais peut parfois apparaître comme une contrainte. Nous verrons que SQL propose une variante, la *jointure externe*, qui permet de la contourner.

La notation de la jointure, $R \bowtie_F S$, est un racourci pour $\sigma_F(R \times S)$. Le critère de rapprochement, F, peut être n'importe quelle opération de comparaison liant un attribut de R à un attribut de S. En pratique, on emploie peu les $\neq$ ou '<' qui sont difficiles à interpréter, et on effectue des égalités.

Note : Si on n'exprime pas de critère de rapprochement, la jointure est équivalente à un produit cartésien.

Il faut être attentif aux ambiguités dans le nommage des attributs qui peut survenir dans la jointure au même titre que dans le produit cartésien. Les solutions à employer sont les mêmes : on préfixe par le nom de la relation ou par un synonyme clair, ou bien on renomme des attributs avant d'effectuer la jointure.

4.2 Expression de requêtes avec l'algèbre

Cette section est consacrée à l'expression de requêtes algébriques complexes impliquant plusieurs opérateurs. On utilise la *composition* des opérations, rendue possible par le fait que tout opérateur produit en sortie une relation sur laquelle on peut appliquer à nouveau des opérateurs.

4.2.1 Sélection généralisée

Regardons d'abord comment on peut généraliser les critères de sélection de l'opérateur σ. Jusqu'à présent on a vu comment sélectionner des lignes satisfaisant *un* critère de sélection, par exemple : "les stations aux Antilles". Maintenant supposons que l'on veuille retrouver les stations qui sont aux Antilles *et* dont la capacité est supérieure à 200. On peut exprimer cette requête par une composition :

$$\sigma_{capacite>200}(\sigma_{region='Antilles'}(Station))$$

Ce qui revient à pouvoir exprimer une sélection avec une *conjonction* de critères. La requête précédente est donc équivalente à celle ci-dessous, où le $\wedge$ dénote le 'et'.

$$\sigma_{capacite>200 \wedge region='Antilles'}(Station)$$

La composition de plusieurs sélections revient à exprimer une conjonction de critères de recherche. De même la composition de la sélection et de l'union permet d'exprimer la *disjonction*. Voici la requête qui recherche les stations qui sont aux Antilles, *ou* dont la capacité est supérieure à 200.

$$\sigma_{capacite>200}(Station) \cup \sigma_{region='Antilles'}(Station)$$

Ce qui permet de s'autoriser la syntaxe suivante, où le '$\vee$' dénote le 'ou'.

$$\sigma_{capacite>200 \ \vee \ region='Antilles'}(Station)$$

Enfin la *différence* permet d'exprimer la *négation* et "d'éliminer" des lignes. Par exemple, voici la requête qui sélectionne les stations dont la capacité est supérieure à 200 mais qui ne sont *pas* aux Antilles.

$$\sigma_{capacite>200}(Station) - \sigma_{region='Antilles'}(Station)$$

Cette requête est équivalente à une sélection où on s'autorise l'opérateur '$\neq$' :

$$\sigma_{capacite>200 \wedge region\neq'Antilles'}(Station)$$

Important : Attention avec les requêtes comprenant une négation, dont l'interprétation est parfois subtile. D'une manière générale, l'utilisation du '$\neq$' *n'est pas* équivalente à l'utilisation de la différence. Voir la prochaine section.

En résumé, les opérateurs d'union et de différence permettent de définir une sélection σ_F où le critère F est une expression booléenne quelconque. Attention cependant : si toute sélection avec un 'ou' peut s'exprimer par une union, l'inverse n'est pas vrai (exercice).

4.2.2 Requêtes conjonctives

Les requêtes dites *conjonctives* constituent l'essentiel des requêtes courantes. Intuitivement, il s'agit de toutes les recherches qui s'expriment avec des 'et', par opposition à celles qui impliquent des 'ou' ou des 'not'. Dans l'algèbre, ces requêtes sont toutes celles qui peuvent s'écrire avec seulement trois opérateurs : π, σ, $\times$ (et donc, indirectement, $\bowtie$).

Les plus simples sont celles où on n'utilise que π et σ. En voici quelques exemples.

— Nom des stations aux Antilles :
$\pi_{nom}(\sigma_{region='Antilles'}(Station))$
— Id des stations où l'on pratique la voile.
$\pi_{idStation}(\sigma_{libelle='Voile'}(Activite))$
— Nom et prénom des clients européens
$\pi_{nom,prenom}(\sigma_{region='Europe'}(Client))$

Des requêtes légèrement plus complexes - et extrêmement utiles - sont celles qui impliquent la jointure. On doit utiliser la jointure dès que les attributs nécessaires pour évaluer une requête sont réparties dans au moins deux tables. Ces "attributs nécessaires" peuvent être :

— Soit des attributs qui figurent dans le résultat ;
— Soit des attributs sur lesquels on exprime un critère de sélection.

Considérons par exemple la requête suivante : "Donner le nom et la région des stations où l'on pratique la voile". Une analyse très simple suffit pour constater que l'on a besoin des attributs `région` et `nom` qui apparaîssent dans la relation *Station*, et de `libellé` qui apparaît dans *Activité*.

Donc il faut faire une jointure, de manière à rapprocher les lignes de *Station* et de *Activité*. Il reste donc à déterminer le (ou les) attribut(s) sur lesquels se fait ce rapprochement. Ici, comme dans la plupart des cas, la jointure permet de "recalculer" l'association entre les relations *Station* et *Activité*. Elle s'effectue donc par appariement de la clé primaire d'une part (dans *Station*), de la clé étrangère d'autre part.

$$\pi_{nom,region}(Station \bowtie_{id=idStation} \sigma_{libelle='Voile'}(\text{Activité}))$$

En pratique, la grande majorité des opérations de jointure s'effectue sur des attributs qui sont clé primaire dans une relation, et clé secondaire dans l'autre. Il ne s'agit pas d'une règle absolue, mais elle résulte du fait que la jointure permet le plus souvent de reconstituer le lien entre des informations qui sont naturellement associées (comme une station et ses activités, ou une station et ses clients), mais qui ont été réparties dans plusieurs relations au moment de la modélisation logique de la base. Cette reconstitution s'appuie sur le mécanisme de clé étrangère qui a été étudié dans le chapitre consacré à la conception.

Voici quelques autres exemples qui illustrent cet état de fait :

— Nom des clients qui sont allés à Passac :
$\pi_{nom}(Client \bowtie_{id=idClient} \sigma_{nom='Passac'}(\text{Séjour}))$
— Quelles régions a visité le client 30 :
$\pi_{region}(\sigma_{idClient=30}(Sejour) \bowtie_{idStation=id} (Station))$
— Nom des clients qui ont eu l'occasion de faire de la voile :
$\pi_{nom}(Client \bowtie_{id=idClient} (Sejour \bowtie_{idStation=idStation} \sigma_{libelle='Voile'}(Activite)))$

La dernière requête comprend deux jointures, portant à chaque fois sur des clés primaires et/ou étrangères. Encore une fois ce sont les clés qui définissent les liens entre les relations, et elle servent donc naturellement de support à l'expression des requêtes.

Voici maintenant un exemple qui montre que cette règle n'est pas systématique. On veut exprimer la requête qui recherche les noms des clients qui sont partis en vacances dans leur région, ainsi que le nom de cette région.

Ici on a besoin des informations réparties dans les relations *Station*, *Séjour* et *Client*. Voici l'expression algébrique :

$$\pi_{nom,client.region}(Client \bowtie_{id=idClient \wedge region=region} (Sejour \bowtie_{idStation=id} Station))$$

Les jointures avec la table *Séjour* se font sur les couples (clé primaire, clé étrangère), mais on a en plus un critère de rapprochement relatif à l'attribut `région` de *Client* et de *Station*.

Note : Dans la projection finale, on utilise la notation client.region pour éviter toute ambiguité.

4.2.3 Requêtes avec $\cup$ et $-$

Pour finir, voici quelques exemples de requêtes impliquant les deux opérateurs $\cup$ et $-$. Leur utilisation est moins fréquente, mais elle peut s'avérer absolument nécessaire puisque ni l'un ni l'autre ne peuvent s'exprimer à l'aide des trois opérateurs "conjonctifs" étudiés précédemment. En particulier, la différence permet d'exprimer toutes les requêtes où figure une négation : on veut sélectionner des données qui *ne* satisfont *pas* telle propriété, ou tous les "untels" *sauf* les 'x' et les 'y', etc.

Illlustration concrète sur la base de données avec la requête suivante : quelles sont les identifiants des stations qui *ne* proposent *pas* de voile ?

$$\pi_{id}(Station) - \pi_{idStation}(\sigma_{libelle='Voile'}(Activite))$$

Comme le suggère cet exemple, la démarche générale pour construire une requête du type "Tous les O qui ne satisfont pas la propriété p" est la suivante :
— Construire une première requête A qui sélectionne tous les O.
— Construire une deuxième requête B'$quislectionnetousles : math :$ '*O qui satisfont :math :'p'*.
— Finalement, faire $A - B$.

Les requêtes A et B peuvent bien entendu être arbitrairement complexes et mettre en oeuvre des jointures, des sélections, etc. La seule contrainte est que le résultat de A et de B comprenne le même nombre d'attributs.

Important : Attention à ne pas considérer que l'utilisation du comparateur $\neq$ est équivalent à la différence. La requête suivante par exemple *ne donne pas* les stations qui ne proposent pas de voile

$$\pi_{idStation}(\sigma_{libelle \neq 'Voile'}(Activite))$$

Pas convaincu(e) ? Réfléchissez un peu plus, faites le calcul concret. C'est l'un de pièges à éviter.

Voici quelques exemples complémentaires qui illustrent ce principe.
— Régions où il y a des clients, mais pas de station.

$$\pi_{region}(Client) - \pi_{region}(Station)$$

— Identifiant des stations qui n'ont pas reçu de client américain.

$$\pi_{id}(Station) - \pi_{idStation}(Sejour \bowtie_{idClient=id} \sigma_{region='Amerique'}(Client))$$

— Id des clients qui ne sont pas allés aux Antilles.

$$\pi_{idClient}(Client) - \pi_{idClient}(\sigma_{region='Antilles'}(Station) \bowtie_{id=idStation} Sejour)$$

La dernière requête construit l'ensemble des `idClient` pour les clients qui ne sont pas allés aux Antilles. Pour obtenir le nom de ces clients, il suffit d'ajouter une jointure (exercice).

4.2.4 Complément d'un ensemble

La différence peut être employée pour calculer le *complément* d'un ensemble. Prenons l'exemple suivant : on veut les ids des clients *et* les stations où ils ne sont pas allés. En d'autres termes, parmi toutes les associations Client/Station possibles, on veut justement celles qui *ne sont pas* représentées dans la base !

C'est un des rares cas où le produit cartésien seul est utile : il permet justement de constituer "toutes les associations possibles". Il reste ensuite à en soustraire celles qui sont dans la base avec l'opérateur $-$.

$$(\pi_{id}(Client) \times \pi_{id}(Station)) - \pi_{idClient,idStation}(Sejour)$$

4.2.5 Quantification universelle

Enfin la différence est nécessaire pour les requêtes qui font appel à la quantification universelle : celles où l'on demande par exemple qu'une propriété soit *toujours* vraie. A priori, on ne voit pas pourquoi la différence peut être utile dans de tels cas. Cela résulte simplement de l'équivalence suivante : une propriété est vraie pour *tous* les éléments d'un ensemble si et seulement si *il n'existe pas* un élément de cet ensemble pour lequel la propriété est *fausse*. La quantification universelle s'exprime par une double négation.

En pratique, on se ramène toujours à la seconde forme pour exprimer des requêtes. Prenons un exemple : quelles sont les stations dont *toutes* les activités ont un prix supérieur à 100 ? On l'exprime également par 'quelles sont stations pour lesquelles *il n'existe pas* d'activité avec un prix *inférieur* à 100'. Ce qui donne l'expression suivante :

$$\pi_{id}(Station) - \pi_{idStation}(\sigma_{prix<100}(Activite))$$

Pour finir, voici une des requêtes les plus complexes, la *division*. L'énoncé (en français) est simple, mais l'expression algébrique ne l'est pas du tout. L'exemple est le suivant : on veut les ids des clients qui sont allés dans *toutes* les stations.

Traduit avec (double) négation, cela donne : les ids des clients tels *qu'il n'existe pas* de station où ils *ne soient pas* allés. Ce qui donne l'expression algébrique suivante :

$$\pi_{id}(Client) - \pi_{id}((\pi_{id}(Client) \times \pi_{id}(Station)) - \pi_{idClient,idStation}(Sejour))$$

On réutilise l'expression donnant les clients et les stations où ils ne sont pas allés (voir plus haut) :

$$\pi_{id}(Client) \times \pi_{id}(Station)) - \pi_{idClient,idStation}(Sejour)$$

On obtient un ensemble B. Il reste à prendre tous les clients, sauf ceux qui sont dans B.

$$\pi_{id}(Client) - B$$

Ce type de requête est rare (heureusement) mais illustre la capacité de l'algèbre à exprimer par de simples manipulations ensemblistes des opérations complexes.

CHAPITRE 5

SQL

Le langage SQL est l'outil standard pour effectuer des recherches ou mises à jour dans une base de données relationnelle. Ce chapitre est consacré à la partie *Langage de Manipulation de Données* (LMD) complémentaire de la partie *Langage de Définition de Données* (LDD) présentée précédemment.

SQL est un langage relativement facile dans la mesure où il ne permet que des opérations assez limitées : manipuler plusieurs tables en entrée (celles de la base) pour obtenir une table en sortie (le résultat). La seule véritable difficulté réside dans l'interprétation des requêtes complexes qui font parfois appel à des logiques sophistiquées. Par ailleurs la multiplicité des variantes syntaxiques offre de nombreuses manières d'exprimer la même interrogation ce qui peut parfois troubler. Nous avons donc choisi d'insister sur la signification des opérations SQL et sur la démarche de conception d'une requête.

Nous utilisons le schéma de la base *Immeuble* du chapitre précédent et l'instance de base donnée ci-dessous. Ce schéma et cette base sont fournis respectivement dans les scripts SchemaImmeuble.sql et BaseImmeuble.sql sur notre site si vous souhaitez effectuer réellement les requêtes proposées parallèlement à votre lecture.

La table Immeuble

Voici le contenu de la table *Immeuble*.

id	nom	adresse
1	Koudalou	3 Rue Blanche
2	Barabas	2 Allée Nikos

La table Appart

Voici le contenu de la table *Appart*.

id	no	surface	étage	id_immeuble
100	1	150	14	1
101	34	50	15	1
102	51	200	2	1
103	52	50	5	1
201	1	250	1	2
202	2	250	2	2

La table Personne

Voici le contenu de la table *Personne*.

id	prénom	nom	profession	id_appart
1		Ross	Informaticien	202
2	Alice	Black	Cadre	103
3	Rachel	Verte	Stagiaire	100
4	William	Dupont	Acteur	102
5	Doug	Ramut	Rentier	201

La table Possede

Voici le contenu de la table *Possede*.

id_personne	id_appart	quote_part
1	100	33
5	100	67
1	101	100
5	102	100
1	202	100
5	201	100
2	103	100

5.1 Principes de SQL

Nous commençons par un abrégé d'interprétation des requêtes SQL qui va nous guider pendant tout ce chapitre. Il existe quelques principes généraux qui s'appliquent quelle que soit la complexité de la requête et auxquels il est très utile de pouvoir se ramener en présence d'un cas complexe.

5.1.1 Forme de base et interprétation

La recherche s'effectue avec la commande `select` dont la forme de base est :

```
select liste_expressions
from   source
[where liste_conditions]
```

L'ordre des trois clauses `select from` et `where` est trompeur pour la signification d'une requête. En fait l'intepré-tation s'effectue *toujours* de la manière suivante :

- la clause `from` définit l'espace de recherche en fonction d'un ensemble de sources de données : cet espace a *toujours* conceptuellement la forme d'une table relationnelle que nous appellerons *Tfrom* ;
- la clause `where` exprime un ensemble de conditions sur les lignes de la table *Tfrom* : seules les lignes pour lesquelles ces conditions sont satisfaites sont conservées ;
- enfin la clause `select` consiste en une liste d'expressions appliquées à chaque ligne de *Tfrom* ayant passé le filtre du `where`.

Nous vous conseillons de revenir toujours à la définition ci-dessus quand vous avez un problème avec vos requêtes SQL. Elle permet de rendre compte uniformément de leur signification de la requête la plus simple à la plus complexe. Le sens de lecture `from where select` facilite la conception et l'interprétation. Voyons tout de suite concrètement comme elle s'applique à une requête très simple.

```
select nom adresse from Immeuble where id=1
```

Cette requête renvoie le résultat suivant.

nom	adresse
Koudalou	3 Rue Blanche

L'espace de recherche est ici constitué d'une seule table de la base Immeuble. Il s'agit du cas le plus simple et le plus courant. On évalue ensuite pour toutes les lignes de cette table les conditions exprimées dans le `where` : seules les lignes (une seule en l'occurrence) pour lesquelles l'attribut `id` vaut 1 satisfont cette condition. Finalement on extrait de cette ligne les valeurs des attributs `nom` et `adresse`.

5.1.2 L'espace de recherche : clause `from`

L'espace de recherche est défini dans la clause `from` par une table ou une combinaison de plusieurs tables. Par "table" il ne faut pas ici comprendre forcément "une des tables de la base" courante même si c'est le cas le plus souvent rencontré. SQL est beaucoup général que cela : une table dans un `from` peut également être *résultat* d'une autre requête. On parlera de table *basée* et de table *calculée* pour distinguer ces deux cas. Ce peut également être une table stockée dans une autre base ou une table calculée à partir de tables basées dans plusieurs bases ou une combinaison de tout cela.

La première requête effectuée ci-dessus ramène les immeubles dont l'id vaut 1. Il n'aura pas échappé au lecteur attentif que le résultat est lui-même une table calculée. Pourquoi ne pourrait-on pas interroger cette table calculée comme une autre ? C'est possible en SQL comme le montre l'exemple suivant :

```
select *
from (select nom adresse from Immeuble where id=1) as Koudalou
```

On a donc placé une requête SQL dans le `from` où elle définit un espace de recherche constitué de son propre résultat. Le mot-clé `as` permet de donner un nom temporaire au résultat. En d'autres termes `Koudalou` est le nom de la table calculée sur laquelle s'effectue la requête. Cette table temporaire n'existe que pendant l'exécution.

On peut aller un peu plus loin et donner définivement un nom à cette requête qui sélectionne l'immeuble. En SQL cela s'appelle une *vue*. On crée une vue dans un schéma avec la commande `create view` (les vues sont traitées en détail plus loin). Par exemple :

```
create view Koudalou as
       select nom adresse from Immeuble where id=1
```

Une fois créée une vue peut être utilisée comme espace de recherche exactement comme une table basée. Le fait que son contenu soit calculé reste transparent pour l'utilisateur.

```
select nom adresse from Koudalou
```

nom	adresse
Koudalou	3 Rue Blanche

Encore une fois, l'interprétation du `from` est indépendante de l'origine des tables : tables basées, tables calculées, et vues.

Note : On peut également interroger les tables ou vues d'une *autre* base pourvu que l'utilisateur connecté ait les droits suffisants. Dans ce cas on préfixe le nom de la table par le nom de la base.

Venons-en maintenant au cas où le `from` est défini par plusieurs tables. Dans un tel cas les contenus des tables sont *toujours* combinées de manière à définir une table virtuelle (le *Tfrom* évoqué précédemment) qui tient lieu d'espace de recherche par la suite. La combinaison la plus simple de deux tables *A* et *B*, obtenue en séparant les noms des tables par une virgule, consiste à effectuer toutes les associations possibles d'une ligne de *A* et d'une ligne de *B*. Voici par exemple ce que cela donne pour les tables *Immeuble* et *Appart*.

```
select * from Immeuble, Appart
```

L'affichage ci-dessus nous montre quel est l'espace de recherche *Tfrom* considéré quand on place ces deux tables dans un `from`.

id	nom	adresse	id	surface	etage	id_immeuble	no
1	Koudalou	3 Rue Blanche	100	150	14	1	1
2	Barabas	2 Allée Nikos	100	150	14	1	1
1	Koudalou	3 Rue Blanche	101	50	15	1	34
2	Barabas	2 Allée Nikos	101	50	15	1	34
1	Koudalou	3 Rue Blanche	102	200	2	1	51
2	Barabas	2 Allée Nikos	102	200	2	1	51
1	Koudalou	3 Rue Blanche	103	50	5	1	52
2	Barabas	2 Allée Nikos	103	50	5	1	52
1	Koudalou	3 Rue Blanche	201	250	1	2	1
2	Barabas	2 Allée Nikos	201	250	1	2	1
1	Koudalou	3 Rue Blanche	202	250	2	2	2
2	Barabas	2 Allée Nikos	202	250	2	2	2

La virgule est en fait un synonyme de `cross join`, terme qui indique que l'on effectue un produit cartésien (*cross product*) des deux tables vues comme des ensembles. Reportez-vous au chapitre *L'algèbre relationnelle* (page 41) pour des explications détaillées. La requête donnant le même résultat est donc :

```
select * from Immeuble cross join Appart
```

D'autre types de combinaisons plus restrictives et souvent plus utiles sont les *jointures*. Regardons à nouveau le résultat précédent. Le fait d'associer chaque ligne de *Immeuble* à chaque ligne de *Appart* nous donne des résultats peu utiles à priori. La seconde ligne associe par exemple le Barabas à un appartement du Koudalou. On ne voit pas quelle application peut trouver intérêt à un tel regroupement. En revanche, associer un appartement à l'immeuble où il se situe est très utile (ne serait-ce que pour trouver son adresse).

Un peu de réflexion suffit pour se convaincre que les lignes intéressantes dans le résultat du produit cartésien sont celles pour lesquelles l'attribut `id` provenant de la table *Immeuble* est égal à l'attribut `id_immeuble` provenant de *Appart*. Un tel produit cartésien "restreint" est une *jointure* dans la terminologie du modèle relationnel. Une combinaison de deux tables par jointure doit indiquer quelles sont les conditions d'association de deux lignes. En SQL on l'exprime avec la syntaxe suivante :

```
table1 join table2 on (conditions)
```

Donc on peut effectuer la jointure dans la clause `from`. On obtient un espace de recherche constitué des paires de lignes pour lesquelles la condition de jointure est vraie.

```
select *
from Immeuble join Appart on (Immeuble.id=Appart.id_immeuble)
```

On obtient le résultat suivant.

id	nom	adresse	id	surface	etage	id_immeuble	no
1	Koudalou	3 Rue Blanche	100	150	14	1	1
1	Koudalou	3 Rue Blanche	101	50	15	1	34
1	Koudalou	3 Rue Blanche	102	200	2	1	51
1	Koudalou	3 Rue Blanche	103	50	5	1	52
2	Barabas	2 Allée Nikos	201	250	1	2	1
2	Barabas	2 Allée Nikos	202	250	2	2	2

Les jointures seront revues plus loin. Comment souvent en SQL il existe plusieurs moyens de les exprimer, la plus utilisée consistant à effectuer un produit cartésien (virgules dans le `from`) suivi de conditions de jointures exprimées dans le `where`.

Ce qu'il faut retenir : le `from` consiste à définir un espace de recherche qui est constitué par combinaison d'une ou plusieurs tables basées ou calculées provenant du même schéma ou d'un autre schéma. Quel que soit le cas cet espace

est lui-même une table (calculée) dont on va chercher à extraire une ou plusieurs lignes (par le `where`) puis une ou plusieurs colonnes (par le `select`).

Beaucoup de difficultés d'interprétation proviennent d'une mauvaise compréhension de l'espace dans lequel on cherche les données. La première étape pour les résoudre consiste à se poser correctement la question : que signifie le `from` de ma requête.

5.1.3 Les conditions : la clause `where`

La clause `where` permet d'exprimer des conditions portant sur les lignes de la table définie par la clause `from`. Ces conditions prennent la forme classique de tests combinés par les connecteurs booléens classiques `and`, `or` et `not`. Les règles de priorité sont exprimées par des parenthèses. Voici quelques exemples simples de requêtes avec leur résultat.

```
select * from Appart where surface > 50
```

id	surface	etage	id_immeuble	no
100	150	14	1	1
102	200	2	1	51
201	250	1	2	1
202	250	2	2	2

```
select * from Appart
where surface > 50
and etage > 3
```

id	surface	etage	id_immeuble	no
100	150	14	1	1

```
select * from Appart
where (surface > 50 and etage > 3)
or id_immeuble=2
```

id	surface	etage	id_immeuble	no
100	150	14	1	1
201	250	1	2	1
202	250	2	2	2

Il est également possible d'exprimer des conditions sur des tables calculées par d'autre requêtes SQL incluses dans la clause `where` et habituellement désignées par le terme de "requêtes imbriquées". On pourra par exemple demander la liste des personnes dont l'appartement fait partie de la table calculée des appartements situés au-dessus du troisième étage.

```
select * from Personne
where id_appart in (select id from Appart where etage < 3)
```

id	prenom	nom	profession	id_appart
1	null	Ross	Informaticien	202
4	William	Dupont	Acteur	102
5	Doug	Ramut	Rentier	201

Avec les requêtes imbriquées on entre dans le monde incertain des requêtes qui semblent claires mais finissent par ne plus l'être du tout. La difficulté vient souvent du fait qu'il faut raisonner simultanément sur plusieurs requêtes qui, de plus, sont souvent interdépendantes (les données sélectionnées dans l'une servent de paramètre à l'autre). Il est très souvent possible d'éviter les requêtes imbriquées comme nous l'expliquons dans ce chapitre.

5.1.4 Les expressions : la clause `select`

Finalement, une fois obtenues les lignes du `from` qui satisfont le `where` on crée à partir de ces lignes le résultat final avec les expressions du `select`. Le terme "expression" désigne ici, comme dans tout langage, une construction syntaxique qui prend une ou plusieurs valeurs en entrée et produit une valeur en sortie. Dans sa forme la plus simple, une expression est simplement un nom d'attribut ou une constante comme dans l'exemple suivant.

```
select surface etage 18 as 'Euros/m2'
from Appart
```

surface	etage	Euros/m2
150	14	18
50	15	18
200	2	18
50	5	18
250	1	18
250	2	18

Les attributs `surface` et `etage` proviennent de *Appart* alors que 18 est une constante qui sera répétée autant de fois qu'il y a de lignes dans le résultat. De plus, on peut donner un nom à cette colonne avec la commande `as`. Voici un second exemple qui montre une expression plus complexe. L'utilisateur (certainement un agent immobilier avisé et connaissant bien SQL) calcule le loyer d'un appartement en fonction d'une savante formule qui fait intervenir la surface et l'étage.

```
select no, surface, etage,
        (surface * 18) * (1 (0.03 * etage)) as loyer
from Appart
```

no	surface	etage	loyer
1	150	14	3834.00
34	50	15	1305.00
51	200	2	3816.00
52	50	5	1035.00
1	250	1	4635.00
2	250	2	4770.00

SQL fournit de très nombreux opérateurs et fonctions de toute sorte qui sont clairement énumérées dans la documentation de chaque système. Elles sont particulièrement utiles pour des types de données un peu délicat à manipuler comme les dates.

Une extension rarement utilisée consiste à effectuer des tests sur la valeur des attributs à l'intérieur de la clause `select` avec l'expression `case` dont la syntaxe est :

```
case
  when test then expression
  [when ...]
  else  expression
end
```

Ces tests peuvent être utilisés par exemple pour effectuer un *décodage* des valeurs quand celles-ci sont difficiles à interpréter ou quand on souhaite leur donner une signification dérivée. La requête ci-dessous classe les appartements en trois catégories selon la surface.

```
 select no etage surface
       case when surface <= 50 then 'Petit'
            when surface > 50 and surface <= 100 then 'Moyen'
            else 'Grand'
```

```
      end as categorie
from Appart
```

no	etage	surface	catégorie
1	14	150	Grand
34	15	50	Petit
51	2	200	Grand
52	5	50	Petit
65	12	70	Moyen
65	12	70	Moyen
1	1	250	Grand
2	2	250	Grand

Voici donc tout ce qu'il faut savoir sur l'interprétation des requêtes SQL exprimées "à plat", sans requête imbriquée. Nous reprenons maintenant la présentation du langage en apportant des précisions sur chaque partie.

5.2 Recherche avec SQL

La recherche la plus simple consiste à récupérer le contenu complet d'une table. On n'utilise pas la clause `where` et le * désigne tous les attributs.

```
select * from Immeuble
```

id	nom	adresse
1	Koudalou	3 Rue Blanche
2	Barabas	2 Allée Nikos

5.2.1 Construction d'expressions

Si on indique explicitement les attributs au lieu d'utiliser *, leur nombre détermine le nombre de colonnes de la table calculée. Le nom de chaque attribut dans cette table est par défaut l'expression du `select` mais on peut indiquer explicitement ce nom avec `as`. Voici un exemple qui illustre également une fonction assez utile, la concaténation de chaînes.

```
select prenom || nom as 'Prenom et nom'
from Personne
```

Prenom et nom
null
Alice Black
Rachel Verte
William Dupont
Doug Ramut

Le résultat montre que l'une des valeurs est à `null`. Le `null` en SQL correspond à *l'absence* de valeur. Logiquement toute opération appliquée à un `null` renvoie un `null` en sortie puisqu'on ne peut calculer aucun résultat à partir d'une valeur inconnue. Ici c'est le prénom de l'une des personnes qui manque. La concaténation du prénom avec le nom est une opération qui "propage" cette valeur à `null`.

5.2.2 La clause `where`

Les conditions de la clause `where` suivent en général la syntaxe `expr1 [not]` Θ `expr2`, où `expr1` et `expr2` sont deux expressions construites à partir de noms d'attributs de constantes et des fonctions et Θ est l'un des opérateurs de comparaison classique < > <= >= != =.

Les conditions se combinent avec les connecteurs booléens `and` `or` et `not`. SQL propose également un prédicat `in` qui teste l'appartenance d'une valeur à un ensemble. Il s'agit (du moins tant qu'on n'utilise pas les requêtes imbriquées) d'une facilité d'écriture pour remplacer le `or`. La requête

```
select *
from Personne
where profession='Acteur'
or profession='Rentier'
```

s'écrit de manière équivalente avec un `in` comme suit :

```
select *
from Personne
where profession in ('Acteur', 'Rentier')
```

id	prenom	nom	profession	id_appart
4	William	Dupont	Acteur	102
5	Doug	Ramut	Rentier	201

Pour les chaînes de caractères, SQL propose l'opérateur de comparaison `like`, avec deux caractères de substitution :

— le "%" remplace n'importe quelle sous-chaîne ;
— le "_" remplace n'importe quel caractère.

L'expression `_ou%ou` est donc interprétée par le `like` comme toute chaîne commençant par un caractère suivi de "ou" suivi de n'importe quelle chaîne suivie une nouvelle fois de "ou".

```
select *
from Immeuble
where nom like '_ou%ou'
```

id	nom	adresse
1	Koudalou	3 Rue Blanche

5.2.3 Valeurs nulles

Il est impossible de déterminer quoi que ce soit à partir d'une valeur à `null`. Nous avons vu que toute opération appliquée à un `null` renvoie `null`. Dans le cas des comparaisons, la présence d'un `null` renvoie un résultat qui n'est ni `true` ni `false` mais `unknown`, une valeur booléenne intermédiaire. Reprenons à nouveau la table *Personne* avec un des prénoms à `null`. La requête suivante devrait ramener toutes les lignes.

```
select *
from Personne
where prenom like '%'
```

Mais la présence d'un `null` empêche l'inclusion de la ligne correspondante dans le résultat.

id	prenom	nom	profession	id_appart
2	Alice	Black	Cadre	103
3	Rachel	Verte	Stagiaire	100
4	William	Dupont	Acteur	102
5	Doug	Ramut	Rentier	201

Cependant la condition `like` n'a pas été évaluée à `false` comme le montre la requête suivante.

```
select  *
from Personne
where prenom not like  '%'
```

On obtient un résultat vide, ce qui montre bien que le `like` appliqué à un `null` ne renvoie pas `false` (car sinon on aurait `not false = true`). C'est d'ailleurs tout à fait normal puisqu'il n'y a aucune raison de dire qu'une absence de valeur ressemble à n'importe quelle chaîne.

Les tables de vérité de la logique trivaluée de SQL sont définies de la manière suivante. Tout d'abord on affecte une valeur aux trois constantes logiques :

- `true` vaut 1
- `false` vaut 0
- `unknown` vaut 0.5

Les connecteurs booléens s'interprètent alors ainsi :

- `val1 and val2` = max(val1 val2)
- `val1 or val2` = min(val1 val2)
- `not val1` = 1 - val1.

On peut vérifier notamment que `not unknown` vaut toujours `unknown`. Ces définitions sont claires et cohérentes. Cela étant il faut mieux prévenir de mauvaises surprises avec les valeurs à `null`, soit en les interdisant à la création de la table avec les options `not null` ou `default`, soit en utilisant le test `is null` (ou son complément `is not null`). La requête ci-dessous ramène toutes les lignes de la table, même en présence de `null`.

```
select *
from Personne
where prenom like '%'
or prenom is null
```

id	prenom	nom	profession	id_appart
1	null	Ross	Informaticien	202
2	Alice	Black	Cadre	103
3	Rachel	Verte	Stagiaire	100
4	William	Dupont	Acteur	102
5	Doug	Ramut	Rentier	201

Attention le test `valeur = null` n'a pas de sens. On ne peut pas être égal à une absence de valeur.

5.2.4 Tri et élimination de doublons

SQL renvoie les lignes du résultat sans se soucier de la présence de doublons. Si on cherche par exemple les surfaces des appartements avec

```
select surface
from Appart
```

on obtient le résultat suivant.

surface
150
50
200
50
250
250

On a autant de fois une valeur qu'il y a de lignes dans le résultat intermédiaire après exécution des clauses `from` et `where`. En général, on ne souhaite pas conserver ces lignes identiques dont la répétition n'apporte aucune information. Le mot-clé `distinct` placé juste après le `select` permet d'éliminer ces doublons.

```
select distinct surface
from Appart
```

surface
150
50
200
250

Le `distinct` est à éviter quand c'est possible car l'élimination des doublons peut entraîner des calculs coûteux. Il faut commencer par calculer entièrement le résultat, puis le trier ou construire une table de hachage, et enfin utiliser la structure temporaire obtenue pour trouver les doublons et les éliminer. Si le résultat est de petite taille cela ne pose pas de problème. Sinon, on risque de constater une grande différence de temps de réponse entre une requête sans `distinct` et la même avec `distinct`.

On peut demander explicitement le tri du résultat sur une ou plusieurs expressions avec la clause `order by` qui vient toujours à la fin d'une requête `select`. La requête suivante trie les appartements par surface puis, pour ceux de surface identique, par l'étage.

```
select *
from Appart
order by surface etage
```

id	surface	etage	id_immeuble	no
103	50	5	1	52
101	50	15	1	34
100	150	14	1	1
102	200	2	1	51
201	250	1	2	1
202	250	2	2	2

Par défaut, le tri est en ordre ascendant. On peut inverser l'ordre de tri d'un attribut avec le mot-clé `desc` .

```
select *
from Appart
order by surface desc,  etage desc
```

id	surface	etage	id_immeuble	no
202	250	2	2	2
201	250	1	2	1
102	200	2	1	51
100	150	14	1	1
101	50	15	1	34
103	50	5	1	52

Bien entendu, on peut trier sur des expressions au lieu de trier sur de simples noms d'attribut.

5.3 Jointures

La jointure est une opération indisoensable dès que l'on souhaite combiner des données réparties dans plusieurs tables. À la base de la jointure, on trouve le produit cartésien qui consiste à trouver toutes les associations possibles d'une

ligne de la première table avec une ligne de la seconde. La jointure consiste à restreindre le résultat du produit cartésien en ne conservant que les associations qui sont intéressantes pour la requête.

Il existe beaucoup de manières différentes d'exprimer les jointures en SQL. Pour les débutants, il est recommandé de se limiter à la forme de base donnée ci-dessous qui est plus facile à interpréter et se généralise à un nombre de tables quelconques.

5.3.1 Syntaxe classique

La méthode la plus courante consiste à effectuer un produit cartésien dans le `from` puis à éliminer les lignes inutiles avec le `where`

Important : Attention, nous parlons ici de la manière dont on conçoit la requête pas de l'évaluation par le système qui essaiera en général d'éviter le calcul complet du produit cartésien.

Prenons l'exemple d'une requête cherchant la surface et l'étage de l'appartement de M. William Dupont. On doit associer deux tables. La première est une table calculée contenant les informations sur William Dupont. On l'obtient par la requête suivante.

```
select *
from Personne
where prenom='William'
and nom='Dupont'
```

id	prenom	nom	profession	id_appart
4	William	Dupont	Acteur	102

La seconde table est l'ensemble des appartements. L'association des deux tables par un produit cartésien consiste simplement à les placer toutes les deux dans le `from`.

```
select  p.id, nom, prenom, id_appart, a.id, surface, etage
from Personne as p, Appart as a
where prenom='William' and nom='Dupont'
```

Ce qui donne le résultat suivant :

id	nom	prenom	id_appart	id	surface	etage
4	Dupont	William	102	100	150	14
4	Dupont	William	102	101	50	15
4	Dupont	William	102	102	200	2
4	Dupont	William	102	103	50	5
4	Dupont	William	102	201	250	1
4	Dupont	William	102	202	250	2

Une première difficulté à résoudre quand on utilise plusieurs tables est la possibilité d'avoir des attributs de même nom dans l'union des schémas, ce qui soulève des ambiguités dans les clauses `where` et `select`. On résout cette ambiguité en préfixant les attributs par le nom des tables dont ils proviennent. Ici, on a même simplifié l'écriture en donnant des *alias* aux nom des tables avec le mot-clé `as`. La table *Personne* et la table *Appart* sont donc respectivement référencées par `p` et `a`.

Notez que la levée de l'ambiguité en préfixant par le nom ou l'alias de la table n'est nécessaire que pour les attributs qui apparaissent en double soit ici `id` qui peut désigner l'identifiant de la personne ou celui de l'appartement.

On n'a placé aucun critère de filtrage pour l'association des lignes provenant des deux tables. William Dupont est donc associé à toutes les lignes de la table *Appart*. La jointure consiste à ajouter une condition complémentaire dans le `where` exprimant le fait que l'appartement auquel on s'intéresse (désigné par sa clé `a.id`) est celui où habite William

Dupont (désigné par la clé étrangère `id_appart`). Le résultat de la requête précédente montre bien que la seule ligne qui nous intéresse est celle où ces deux attributs ont la même valeur. D'où la requête donnant le bon résultat :

```
select  p.id, p.nom, p.prenom, p.id_appart, a.id, a.surface, a.etage
from Personne as p Appart as a
where prenom='William'
and nom='Dupont'
and   a.id = p.id_appart
```

id	nom	prenom	id_appart	id	surface	etage
4	Dupont	William	102	102	200	2

Comme dans la très grande majorité des cas la jointure consiste à exprimer une égalité entre la clé primaire de l'une des tables et la clé étrangère correspondante de l'autre. Un peu de réflexion permet de réaliser que l'on recrée ainsi le lien défini au niveau du modèle entitéassociation. Mais SQL est plus puissant que cette reconstitution puisqu'on peut exprimer des conditions de jointure sur n'importe quel attribut et pas seulement sur ceux qui sont des clés.

Imaginons que l'on veuille trouver les appartements d'un même immeuble qui ont la même surface. On veut associer une ligne de *Appart* à une autre ligne de *Appart* avec les conditions suivantes :

— elles sont dans le même immeuble (attribut `id_immeuble`);
— elles ont la même valeur pour l'attribut `surface`;
— elles correspondent à des appartements distincts (attributs `id`).

La requête exprimant ces conditions est donc :

```
select a1.id, a1.surface, a1.etage, a2.id, a2.surface, a2.etage
from Appart a1 Appart a2
where a1.id != a2.id
and a1.surface = a2.surface
and a1.id_immeuble = a2.id_immeuble
```

Ce qui donne le résultat suivant :

id	surface	etage	id	surface	etage
101	50	15	103	50	5
103	50	5	101	50	15
201	250	1	202	250	2
202	250	2	201	250	1

On peut noter que dans le résultat la même paire apparaît deux fois avec des ordres inversés. On peut éliminer cette redondance en remplaçant `a1.id != a2.id` par `a1.id < a2.id`.

Les alias `a1` et `a2` jouent le rôle de "curseurs" qui référencent, à un moment donné, deux lignes de la table *Appart*. Ces deux lignes sont associées et incluses dans le résultat si la condition du `where` est satisfaite.

Voici quelques exemples complémentaires de jointures impliquant parfois plus de deux tables exprimées par produit cartésien, puis restriction avec `where`.

— Qui habite un appartement de plus de 200 m2 ?

```
select prenom, nom, profession
from Personne, Appart
where id_appart = Appart.id
and  surface >= 200
```

Attention à lever l'ambiguité sur les noms d'attributs quand ils peuvent provenir de deux tables (c'est le cas ici pour `id`).

— Qui habite le Barabas ?

```
select prenom, p.nom, no, surface, etage
from   Personne as p, Appart as a, Immeuble as i
where  p.id_appart=a.id
and    a.id_immeuble=i.id
and    i.nom='Barabas'
```

— Qui habite un appartement qu'il possède et avec quelle quote-part ?

```
select prenom, nom, quote_part
from   Personne as p, Possede as p2, Appart as a
where  p.id=p2.id_personne /* p est propriétaire */
and    p2.id_appart=a.id   /* de l'appartement a */
and    p.id_appart=a.id   /* et il y habite     */
```

— De quel(s) appartement(s) Alice Black est-elle propriétaire et dans quel immeuble ?
Voici la requête sur les quatre tables avec des commentaires inclus montrant les jointures.

```
 select i.nom, no, etage, surface
from  Personne as p, Appart as a, Immeuble as i, Possede as p2
where  p.id=p2.id_personne /* Jointure PersonnePossede */
and    p2.id_appart = a.id /* Jointure PossedeAppart */
and    a.id_immeuble= i.id /* Jointure AppartImmeuble */
and    p.nom='Black' and p.prenom='Alice'
```

Attention à lever l'ambiguité sur les noms d'attributs quand ils peuvent provenir de deux tables (c'est le cas ici pour `id`).

La jointure classique "à plat" où toutes les tables sont placées dans le `from` et associées par produit cartésien est une manière tout à fait recommandable de procéder surtout pour les débutants SQL. Elle permet de se ramener toujours à la même méthode d'interprétation et consolide la compréhension des principes d'interrogation d'une base relationnelle.

Toutes ces jointures peuvent s'exprimer avec d'autres syntaxes : tables calculées dans le `from` opérateur de jointure dans le `from` ou (pas toujours) requêtes imbriquées. À l'exception notable des jointures externes et des requêtes imbriquées avec négation, elles n'apportent aucune expressivité supplémentaire. Toutes ces variantes constituent des moyens plus ou moins commodes d'exprimer différemment la jointure.

5.3.2 Tables calculées dans le `from`

La possibilité de placer des requêtes dans le `from` est prise en compte depuis peu de temps par les SGBD relationnels. Cette option n'apporte rien en terme d'expressivité mais elle peut (question de goût) parfois être considérée comme plus claire ou plus proche du raisonnement intuitif. Elle ne sert que dans les cas où il existe un critère de sélection sur une table. En revanche les conditions de jointure restent dans le `where`.

Nous reprenons quelques exemples de requêtes que nous avons déjà résolues avec des jointures "à plat".

— Quel est l'appartement où habite M. William Dupont ?

```
select no, surface, etage
from  Appart, (select id_appart from Personne
               where prenom='William'
               and nom='Dupont') as Dupont
where id=id_appart
```

no	surface	etage
51	200	2

Un (léger) avantage est de supprimer dans la table calculée les attributs qui soulèvent des ambiguités (par exemple l'id de la personne qui ne sert à rien par la suite). Il faut *toujours* donner un alias avec le mot-clé `as` à la table calculée.

— Qui habite un appartement de plus de 200 m2 ?

```
select prenom, nom, profession
from Personne as p,  (select id from Appart where surface>=200) as a
where p.id_appart=a.id
```

— Qui habite le Barabas ?
Dernier exemple montrant la possibilité d'effectuer une jointure pour obtenir la table calculée.

```
select prenom, nom, no, surface, etage
from    Personne p, (select Appart.id, no, surface, etage
                     from Appart, Immeuble
                     where id_immeuble=Immeuble.id
                     and nom='Barabas') as a
where  p.id_appart=a.id
```

5.3.3 Opérateurs de jointure

La virgule séparant deux tables dans un `from` exprime un produit cartésien : toutes les combinaisons de lignes sont considérées puis soumises aux conditions du `where`. Comme nous l'avons vu il s'agit d'un moyen d'effectuer la jointure. Mais SQL propose également des opérateurs de jointure dans le `from` avec la syntaxe suivante : `table1 opérateur table2 [condition]`

Les opérateurs disponibles sont :
— `cross join` le produit cartésien synonyme de la virgule ;
— `join` la jointure accompagnée de conditions ;
— `straight join` jointure forçant l'ordre d'accès aux tables (déconseillé : mieux vaut laisser faire le système) ;
— `left [outer] join` et `right [outer] join` : jointures externes ;
— `natural [left,right [outer]] join` : jointure "naturelle".

Le seul opérateur qui n'est pas redondant avec des syntaxes présentées précédemment est la jointure externe (`outer`). On peut à bon droit considérer que cette multiplication de variantes syntaxiques ne fait que compliquer le langage. Nous présentons dans ce qui suit les opérateurs `join` et `outer join`. La jointure naturelle est une variante possible quand la clé primaire et la clé étrangère ont le même nom.

L'opérateur `join` permet d'exprimer directement dans le `from` les conditions de jointure. Il est très proche dans sa conception des jointures à plat étudiées précédemment. Quelques exemples reprenant des requêtes déjà exprimées suffiront pour comprendre le mécanisme.

— Quel est l'appartement où habite M. William Dupont ?

```
select no, surface, etage
from Personne join Appart on (Personne.id_appart=Appart.id)
where prenom='William' and nom='Dupont'
```

Il suffit de comparer avec la version combinant produit cartésien et `where` pour se rendre compte que la différence est minime. On peut tout à fait considérer que la version avec `join` est plus claire.

— Qui habite un appartement de plus de 200 m2 ?

```
select prenom, nom, profession
from Personne as p join Appart as a
        on (id_appart=a.id and surface>=200)
```

Cet exemple illustre un "détournement" de la condition de jointure. On y a inclus une condition de sélection (la surface supérieure à 200). La requête est correcte mais il est possible que la méthode d'évaluation ne soit pas optimale.
Il faut souligner (nous y reviendrons) que ces variantes alternatives représentent un défi pour le module chargé d'optimiser l'évaluation des requêtes. Un avantage potentiel des jointures à plat où tout est exprimé dans le `where` est qu'elles offrent une forme canonique à partir de laquelle un optimiseur peut travailler avec un maximum de liberté.

— Qui habite le Barabas ?
On peut composer deux opérateurs de jointure comme le montre ce dernier exemple.

```
select prenom, p.nom, no, surface, etage
from  Personne p join (Appart a join Immeuble i
                       on id_immeuble=i.id)
         on (id_appart=a.id)
where i.nom='Barabas'
```

Contrairement aux jointures exprimées avec `join`, les jointures externes ne peuvent pas s'exprimer avec un `where`. Qu'est-ce qu'une jointure externe ? Effectuons la requête qui affiche tous les appartements avec leur occupant.

```
select id_immeuble, no, etage, surface, nom, prenom
from  Appart as a join Personne as p on (p.id_appart=a.id)
```

Voici ce que l'on obtient :

id_immeuble	no	etage	surface	nom	prenom
2	2	2	250	Ross	null
1	52	5	50	Black	Alice
1	1	14	150	Verte	Rachel
1	51	2	200	Dupont	William
2	1	1	250	Ramut	Doug

Il manque un appartement le 34 du Koudalou. En effet cet appartement n'a pas d'occupant. Il n'y a donc aucune possibilité que la condition de jointure soit satisfaite.

La jointure externe permet d'éviter cette élimination parfois indésirable. On considère alors une hiérarchie entre les deux tables. La première table (en général celle de gauche) est dite "directrice" et toutes ses lignes, même celle qui ne trouvent pas de correspondant dans la table de droite, seront prises en compte. Les lignes de la table de droite sont en revanche optionnelle.

Si pour une ligne de la table de gauche on trouve une ligne satisfaisant le critère de jointure dans la table de droite, alors la jointure s'effectue normalement. Sinon, les attributs provenant de la table de droite sont affichés à `null`. Voici la jointure externe entre *Appart* et *Personne*. Le mot-clé `outer` est optionnel.

```
select id_immeuble, no etage, surface, nom, prenom
from  Appart as a left outer join Personne as p on (p.id_appart=a.id)
```

id_immeuble	no	etage	surface	nom	prenom
1	1	14	150	Verte	Rachel
1	34	15	50	null	null
1	51	2	200	Dupont	William
1	52	5	50	Black	Alice
2	1	1	250	Ramut	Doug
2	2	2	250	Ross	null

Notez les deux attributs `prenom` et `nom` à `null` pour l'appartement 34.

Il existe un `right outer join` qui prend la table de droite comme table directrice. On peut combiner la jointure externe avec des jointures normales des sélections des tris etc. Voici la requête qui affiche le nom de l'immeuble en plus des informations précédentes et trie par numéro d'immeuble et numéro d'appartement.

```
select i.nom, no, etage, surface, p.nom, prenom
from  Immeuble  as i
       join
           (Appart as a left outer join Personne as p
                     on (p.id_appart=a.id))
```

```
        on (i.id=a.id_immeuble)
order by i.id, a.no
```

5.3.4 Opérations ensemblistes

La norme SQL ANSI comprend des opérations qui considèrent les tables comme des ensembles et effectuent des intersections des unions ou des différences avec les mots-clé `union`, `intersect` ou `except`. Chaque opérateur s'applique à deux tables de schéma identique (même nombre d'attributs mêmes noms mêmes types).

L'union est un opérateur peu utilisé car il est difficile de trouver des cas où son application est justifiée. Voici un exemple montrant le calcul de l'union des noms d'immeuble et des noms de personne.

```
select nom from Immeuble
 union
select nom from Personne
```

Le `except` exprime la différence entre deux ensembles. Il est avantageusement remplacé par l'utilisation des requêtes imbriquées et des `not in` et `not exists` présentés ci-dessous.

5.3.5 Requêtes imbriquées

Les requêtes peuvent être imbriquées les unes dans les autres de deux manières :

- dans la clause `from` : la requête est placée entre parenthèses et son résultat est vu comme une table ;
- dans la clause `where` avec l'opérateur `in` qui permet de tester l'appartenance de la valeur d'un attribut à un ensemble calculé par une requête.

Reprenons l'exemple de la requête trouvant la surface et l'étage de l'appartement de M. Dupont. On peut l'exprimer avec une requête imbriquée de deux manières :

```
select surface, etage
from Appart (select id_appart
             from Personne
             where prenom='William' and nom='Dupont') as ri
where Appart.id = ri.id_appart

select surface, etage from Appart
where  id  in (select id_appart
               from Personne
               where prenom='William' and nom='Dupont')
```

Le mot-clé `in` exprime la condition d'appartenance de l'identifiant de l'appartement à l'ensemble d'identifiants constitué avec la requête imbriquée. Il doit y avoir correspondance entre le nombre et le type des attributs auxquels s'applique la comparaison par `in`. L'exemple suivant montre une comparaison entre des paires d'attributs (ici on cherche des informations sur les propriétaires).

```
select prenom, nom, surface, etage
from Appart as a join Personne as p on (a.id=p.id_appart)
where  (p.id p.id_appart)
          in (select id_personne id_appart from Possede)
```

prenom	nom	surface	etage
null	Ross	250	2
Alice	Black	50	5
Doug	Ramut	250	1

Il est bien entendu assez direct de réécrire la requête ci-dessus comme une jointure classique. Parfois l'expression avec requête imbriquée peut s'avérer plus naturelle. Supposons que l'on cherche les immeubles dans lesquels on trouve un appartement de 50 m2. Voici l'expression avec requête imbriquée.

```
select *
from Immeuble
where id in (select id_immeuble from Appart where surface=50)
```

id	nom	adresse
1	Koudalou	3 Rue Blanche

La requête directement réécrite en jointure donne le résultat suivant :

```
select i.* from Immeuble as i join Appart as a
                 on (i.id=a.id_immeuble)
where surface=50
```

id	nom	adresse
1	Koudalou	3 Rue Blanche
1	Koudalou	3 Rue Blanche

On obtient deux fois le même immeuble puisqu'il peut être associé à deux appartements différents de 50 m2. Il suffit d'ajouter un `distinct` après le `select` pour régler le problème, mais on peut considérer que dans ce cas la requête imbriquée est plus appropriée. Attention cependant : il n'est pas possible d'obtenir dans le résultat des attributs appartenant aux tables des requêtes imbriquées.

Le principe général des requêtes imbriquées est d'exprimer des conditions sur des tables calculées par des requêtes. Ces conditions sont les suivantes :

- `exists R` : renvoie `true` si *R* n'est pas vide `false` sinon.
- *t* `in R` où est une ligne dont le type (le nombre et le type des attributs) est celui de *R* : renvoie `true` si *t* appartient à *R* `false` sinon.
- *v cmp* `any R` où *cmp* est un comparateur SQL (< > = etc.) : renvoie `true` si la comparaison avec *au moins une* des lignes de la table *R* renvoie `true`.
- *v cmp* `all R` où *cmp* est un comparateur SQL (< > = etc.) : renvoie `true` si la comparaison avec *toutes* les lignes de la table *R* renvoie `true`.

De plus toutes ces expressions peuvent être préfixées par `not` pour obtenir la négation. La richesse des expressions possibles permet d'effectuer une même interrogation en choisissant parmi plusieurs syntaxes possibles. En général, tout ce qui n'est pas basé sur une négation `not in` ou `not exists` peut s'exprimer *sans* requête imbriquée.

Le cas du `all` correspond à une négation puisque si une propriété est *toujours* vraie il n'existe pas de cas où elle est fausse. La requête ci-dessous applique le `all` pour chercher l'étage le plus élevé de l'immeuble 1.

```
select * from Appart
    where id_immeuble=1
    and    etage >= all (select etage from Appart where id_immeuble=1)
```

Le `all` exprime une comparaison qui vaut pour *toutes* les lignes ramenées par la requête imbriquée. Attention aux valeurs à `null` dans ce genre de situation : toute comparaison avec une de ces valeurs renvoie `UNKNOWN` et cela peut entraîner l'échec du `all`. Il n'existe pas d'expression avec jointure qui puisse exprimer ce genre de condition.

5.3.6 Requêtes correlées

Les exemples de requêtes imbriquées donnés précédemment pouvaient être évalués indépendamment de la requête principale, ce qui permet au système (s'il le juge nécessaire) d'exécuter la requête en deux phases. La clause `exists` fournit encore un nouveau moyen d'exprimer les requêtes vues précédemment en basant la sous-requête sur une ou plusieurs valeurs issues de la requête principale. On parle alors de requêtes *correlées*.

Voici encore une fois la recherche de l'appartement de M. William Dupont exprimée avec `exists` :

```
select * from Appart
where  exists  (select * from Personne
                where  prenom='William' and nom='Dupont'
                and    Personne.id_appart=Appart.id)
```

On obtient donc une nouvelle technique d'expression qui permet d'aborder le critère de recherche sous une troisième perspective : on conserve un appartement si, *pour cet appartement*, l'occupant s'appelle William Dupont. Il s'agit assez visiblement d'une jointure mais entre deux tables situées dans des requêtes (ou plutôt des "blocs") distinctes. La condition de jointure est appelée corrélation d'où le nom de ce type de technique.

Toutes les jointures peuvent d'exprimer avec `exists` ou `in`. Voici quelques exemples reprenant des requêtes déjà vues précédemment.

— Qui habite un appartement de plus de 200 m2 ?
Avec `in` :

```
select prenom nom profession
from Personne
where id_appart in (select id from Appart where surface >= 200)
```

Avec `exists` :

```
select prenom nom profession
from Personne  p
where exists (select * from Appart a
              where a.id=p.id_appart
              and surface >= 200)
```

— Qui habite le Barabas ?
Avec `in` :

```
select prenom p.nom no surface etage
from   Personne as p Appart as a
where  p.id_appart=a.id
and a.id_immeuble in
         (select id from Immeuble
          where  nom='Barabas')
```

Avec `exists` :

```
select prenom p.nom no surface etage
from Personne as p Appart as a
where  p.id_appart=a.id
and exists (select * from Immeuble i
            where  i.id=a.id_immeuble
            and i.nom='Barabas')
```

Important : dans une sous-requête associée à la clause `exists` peu importe les attributs du `select` puisque la condition se résume à : cette requête ramène-t-elle au moins une ligne ou non ? On peut donc systématiquement utiliser `select *`.

Enfin rien n'empêche d'utiliser plusieurs niveaux d'imbrication au prix d'une forte dégradation de la lisibilité. Voici la requête "De quel(s) appartement(s) Alice Black est-elle propriétaire et dans quel immeuble ?" écrite avec plusieurs niveaux.

```
select i.nom no etage surface
from  Immeuble as i Appart as a
```

```
where  a.id_immeuble= i.id
and    a.id in
             (select id_appart from Possede
              where id_personne in
                       (select id from Personne
                        where  nom='Black'
                        and prenom='Alice'))
```

En résumé une jointure entre les tables *R* et *S* de la forme :

```
select R.*
from R S
where R.a = S.b
```

peut s'écrire de manière équivalente avec une requête imbriquée :

```
select [distinct] *
from R
where R.a in (select S.b from S)
```

ou bien encore sous forme de requête corrélée :

```
select [distinct] *
from R
where exists (select S.b from S where S.b = R.a)
```

Le choix de la forme est matière de goût ou de lisibilité, ces deux critères relevant de considérations essentiellement subjectives.

5.3.7 Requêtes avec négation

Les requêtes imbriquées sont en revanche irremplaçables pour exprimer des négations. On utilise alors `not in` ou (de manière équivalente) `not exists`. Voici un premier exemple avec la requête : *donner les appartements sans occupant*.

```
select * from Appart
where  id not in (select id_appart from Personne)
```

id	surface	etage	id_immeuble	no
101	50	15	1	34

La négation est aussi un moyen d'exprimer des requêtes courantes comme celle recherchant l'appartement le plus élevé de son immeuble. En SQL, on utilisera typiquement une sous-requête pour prendre l'étage maximal d'un immeuble, et on utilisera cet étage pour sélectionner un ou plusieurs appartements, le tout avec une requête correlée pour ne comparer que des appartements situés dans le même immeuble.

```
select *
from Appart as a1
where etage =  (select max(etage) from Appart as a2
                where a1.id_immeuble=a2.id_immeuble)
```

id	surface	etage	id_immeuble	no
101	50	15	1	34
202	250	2	2	2

Il existe en fait beaucoup de manières d'exprimer la même chose. Tout d'abord cette requête peut en fait s'exprimer sans la fonction *max()* avec la négation : si *a* est l'appartement le plus élevé, c'est *qu'il n'existe pas* d'étage plus elevé

que *a*. On utilise alors habituellement une requête dite "corrélée" dans laquelle la sous-requête est basée sur une ou plusieurs valeurs issues des tables de la requête principale.

```
select *
from Appart as a1
where not exists  (select * from Appart as a2
                   where a2.etage > a1.etage
                   and a1.id_immeuble = a2.id_immeuble)
```

Autre manière d'exprimer la même chose : si l'étage est le plus élevé, tous les autres sont situés à un étage inférieur. On peut utiliser le mot-clé `all` qui indique que la comparaison est vraie avec *tous* les éléments de l'ensemble constitué par la sous-requête.

```
select *
from Appart as a1
where etage >= all (select etage from Appart as a2
                    where a1.id_immeuble=a2.id_immeuble)
```

Dernier exemple de négation : quels sont les personnes qui ne possèdent aucun appartement même partiellement ? Les deux formulations ci-dessous sont équivalentes, l'une s'appuyant sur `not in`, et l'autre sur `not exists`.

```
select *
from Personne
where id not in (select id_personne from Possede)

select *
from Personne as p1
where not exists (select * from Possede as p2
                  where p1.id=p2.id_personne)
```

5.4 Agrégats

Les requêtes agrégat en SQL consistent à effectuer des regroupements de lignes en fonction des valeurs d'une ou plusieurs expressions. Ce regroupement est spécifié par la clause `group by`. On obtient une structure qui n'est pas une table relationnelle puisqu'il s'agit d'un ensemble de groupes de lignes. On doit ensuite ramener cette structure à une table en appliquant des *fonctions de groupes* qui déterminent des valeurs agrégées calculées pour chaque groupe.

Enfin il est possible d'exprimer des conditions sur les valeurs agrégées pour ne conserver qu'un ou plusieurs des groupes constitués. Ces conditions portent sur des *groupes* de lignes et ne peuvent donc être obtenues avec `where`. On utilise alors la clause `having`.

Les agrégats s'effectuent *toujours* sur le résultat d'une requête classique `select -from`. On peut donc les voir comme une extension de SQL consistant à partitionner un résultat en groupes selon certains critères, puis à exprimer des conditions sur ces groupes, et enfin à appliquer des fonctions d'agrégation.

Il existe un groupe par défaut : c'est la table toute entière. Sans même utiliser `group by`, on peut appliquer les fonctions d'agrégation au contenu entier de la table comme le montre l'exemple suivant.

```
select count(*), count(prenom), count(nom)
from Personne
```

Ce qui donne :

count(*)	count(prenom)	count(nom)
5	4	5

On obtient 5 pour le nombre de lignes, 4 pour le nombre de prénoms, et 5 pour le nombre de noms. En effet, l'attribut `prenom` est à `null` pour la première personne et n'est en conséquence pas pris en compte par la fonction d'agrégation. Pour compter toutes les lignes, on doit utiliser `count(*)` ou un attribut déclaré comme `not null`. On peut aussi compter le nombre de valeurs distinctes dans un groupe avec `count(distinct <expression>)`.

5.4.1 La clause `group by`

Le rôle du `group by` est de partitionner une table (calculée ou basée) en fonction d'un critère (un attribut ou plus généralement une expression sur des attributs). Pour bien analyser ce qui se passe pendant une requête avec `group by` on peut décomposer l'exécution d'une requête en deux étapes. Prenons l'exemple de celle permettant de vérifier que la somme des quote-part des propriétaires est bien égale à 100 pour tous les appartements.

```
select  id_appart sum(quote_part)
from    Possede
group by id_appart
```

id_appart	sum(quote_part)
100	100
101	100
102	100
103	100
201	100
202	100

Dans une première étape le système va constituer les groupes. On peut les représenter avec un tableau comprenant, pour chaque ligne, d'une part la (ou les) valeur(s) du (ou des) attribut(s) de partitionnement (ici `id_appart`), d'autre part l'ensemble de lignes dans lesquelles on trouve cette valeur. Ces lignes "imbriquées" sont séparées par des points-virgule dans la représentation ci-dessous.

id_appart	Groupe	count
100	(id_personne=1 quote_part=33 ; id_personne=5 quote_part=67)	2
101	(id_personne=1 quote_part=100)	1
102	(id_personne=5 quote_part=100)	1
202	(id_personne=1 quote_part=100)	1
201	(id_personne=5 quote_part=100)	1
103	(id_personne=2 quote_part=100)	1

Le groupe associé à l'appartement 100 est constitué de deux copropriétaires. Le tableau ci-dessus n'est donc pas une table relationnelle dans laquelle chaque cellule ne peut contenir qu'une seule valeur.

Pour se ramener à une table relationnelle, on transforme durant la deuxième étape chaque groupe de lignes en une valeur par application d'une fonction d'agrégation. La fonction `count()` compte le nombre de lignes dans chaque groupe, `max()` donne la valeur maximale d'un attribut parmi l'ensemble des lignes du groupe, etc. La liste des fonctions d'agrégation est donnée ci-dessous :

- `count(expression)`, Compte le nombre de lignes pour lesquelles `expression` est `not null`.
- `avg(expression)`, Calcule la moyenne de `expression`.
- `min(expression)`, Calcule la valeur minimale de `expression`.
- `max(expression)`, Calcule la valeur maximale de `expression`.
- `sum(expression)`, Calcule la somme de `expression`.
- `std(expression)`, Calcule l'écart-type de `expression`.

Dans la norme SQL l'utilisation de fonctions d'agrégation pour les attributs qui n'apparaissent pas dans le `group by` est *obligatoire*. Une requête comme :

```
select  id, surface, max(etage)
from    Appart
group by surface
```

srea rejetée parce que le groupe associé à une même surface contient deux appartements différents (et donc deux valeurs différentes pour `id`), et qu'il n'y a pas de raison d'afficher l'un plutôt que l'autre.

5.4.2 La clause having

Finalement, on peut faire porter des conditions sur les groupes, ou plus précisément sur le résultat de fonctions d'agrégation appliquées à des groupes avec la clause `having`. La clause `where` ne peut exprimer des conditions que sur les lignes prises une à une. Par exemple, on peut sélectionner les appartements pour lesquels on connaît au moins deux copropriétaires.

```
select  id_appart, count(*)
from    Possede
group by id_appart
having count(*) >= 2
```

On voit que la condition porte ici sur une propriété de l'*ensemble* des lignes du groupe et pas de chaque ligne prise individuellement. La clause `having` est donc toujours exprimée sur le résultat de fonctions d'agrégation.

Pour conclure, voici une requête sélectionnant la surface possédée par chaque copropriétaire pour l'immeuble 1. La surface possédée est la somme des surfaces d'appartements possédés par un propriétaire, pondérées par leur quote-part. On regroupe par propriétaire et on trie sur la surface possédée.

```
select prenom nom,
       sum(quote_part * surface / 100) as 'Surface possédée'
from (Personne as p1 join Possede as p2 on (p1.id=p2.id_personne) )
          join  Appart as a on (a.id=p2.id_appart)
where id_immeuble = 1
group by p1.id
order by sum(quote_part * surface / 100)
```

On obtient le résultat suivant.

prénom	nom	Surface possédée
Alice	Black	50.00000
null	Ross	125.00000
Doug	Ramut	275.00000

5.5 Mises à jour

Les commandes de mise à jour (insertion, destruction, modification) sont considérablement plus simples que les interrogations.

5.5.1 Insertion

L'insertion s'effectue avec la commande `insert`, avec trois variantes. Dans la première on indique la liste des valeurs à insérer sans donner explicitement le nom des attributs. Le système suppose alors qu'il y a autant de valeurs que d'attributs, et que l'ordre des valeurs correspond à celui des attributs dans la table. On peut indiquer `null` pour les valeurs inconnues.

```
insert into Immeuble
values (1 'Koudalou' '3 Rue Blanche')
```

Si on veut insérer dans une partie seulement des attributs, il faut donner la liste explicitement.

```
insert into Immeuble (id nom adresse)
values (1 'Koudalou' '3 Rue Blanche')
```

Il est d'ailleurs préférable de toujours donner la liste des attributs. La description d'une table peut changer par ajout d'attribut, et l'ordre `insert` qui marchait un jour ne marchera plus le lendemain.

Enfin avec la troisième forme de `insert` il est possible d'insérer dans une table le résultat d'une requête. Dans ce cas la partie `values` est remplacée par la requête elle-même. Voici un exemple avec une nouvelle table *Barabas* dans laquelle on insère uniquement les informations sur l'immeuble "Barabas".

```
create table Barabas (id int not null,
                      nom varchar(100) not null,
                      adresse varchar(200),
                      primary key (id)
)

insert into Barabas
select * from Immeuble where nom='Barabas'
```

5.5.2 Destruction

La destruction s'effectue avec la clause `delete` dont la syntaxe est :

```
delete from table
where condition
```

`table` étant bien entendu le nom de la table, et `condition` toute condition ou liste de conditions valide pour une clause `where`. En d'autres termes, si on effectue avant la destruction la requête

```
select * from table
where condition
```

on obtient l'ensemble des lignes qui seront détruites par `delete`. Procéder de cette manière est un des moyens de s'assurer que l'on va bien détruire ce que l'on souhaite.

5.5.3 Modification

La modification s'effectue avec la clause `update`. La syntaxe est proche de celle du `delete` :

```
update table set A1=v1, A2=v2, ... An=vn
where condition
```

Comme précédemment `table` est la table, les `Ai` sont les attributs les `vi` les nouvelles valeurs, et `condition` est toute condition valide pour la clause `where`.

5.6 Les vues

Une requête SQL produit toujours une table. Cela suggère la possibilité d'ajouter au schéma des tables *calculées*, qui ne sont rien d'autre que le résultat de requêtes stockées. De telles tables sont nommées des *vues* dans la terminologie relationnelle. On peut interroger des vues comme des tables stockées et, dans certaines limites, faire des mises à jour des tables stockées au travers de vues.

Une vue n'induit aucun stockage puisqu'elle n'existe pas physiquement. Elle permet d'obtenir une représentation différente des tables sur lesquelles elle est basée avec deux grands avantages :

— on peut faciliter l'interrogation de la base en fournissant sous forme de vues des requêtes prédéfinies ;
— on peut masquer certaines informations en créant des vues et en forçant par des droits d'accès l'utilisateur à passer par ces vues pour accéder à la base.

Les *vues* constituent donc un moyen complémentaire de contribuer à la sécurité (par restriction d'accès) et à la facilité d'utilisation (en offrant une "schéma virtuel" simplifié).

5.6.1 Création et interrogation d'une vue

Une vue est en tout point comparable à une table : en particulier on peut l'interroger par SQL. La grande différence est qu'une vue est le résultat d'une requête avec la caractéristique essentielle que ce résultat est réévalué à chaque fois que l'on accède à la vue. En d'autres termes une vue est *dynamique* : elle donne une représentation fidèle de la base au moment de l'évaluation de la requête.

Une vue est essentiellement une requête à laquelle on a donné un nom. La syntaxe de création d'une vue est très simple :

```
create view nomvue ([listeattributs])
as          requete
[with check option]
```

Nous avons déjà vu au début de ce chapitre vl'exemple d'une vue sur la table *Immeuble* montrant uniquement le Koudalou. En voici une nouvelle version plus complète avec le nombre d'appartements.

```
create Koudalou as
     select nom, adresse, count(*) as nb_apparts
     from Immeuble as i join Appart as a on (i.id=a.id_immeuble)
     where i.id=1
     group by i.id, nom, adresse
```

La destruction d'une vue a évidemment beaucoup moins de conséquences que pour une table puisqu'on supprime uniquement la *définition* de la vue pas son *contenu*.

On interroge la vue comme n'importe quelle table.

```
select * from Koudalou
```

nom	adresse	nb_apparts
Koudalou	3 Rue Blanche	4

La vue fait maintenant partie du schéma. On ne peut d'ailleurs évidemment pas créer une vue avec le même nom qu'une table (ou vue) existante. La définition d'une vue peut consister en un requête SQL aussi complexe que nécessaire, avec jointures, regroupements, tris.

Allons un peu plus loin en définissant sous forme de vues un accès aux informations de notre base *Immeuble*, mais restreint uniquement à tout ce qui concerne l'immeuble Koudalou. On va en profiter pour offrir dans ces vues un accès plus facile à l'information. La vue sur les appartements, par exemple, va contenir contrairement à la table *Appart* le nom et l'adresse de l'immeuble et le nom de son occupant.

```
create or replace view AppartKoudalou as
   select no, surface, etage, i.nom as immeuble, adresse,
          p.prenom ||  p.nom as occupant
   from (Immeuble as i join Appart as a on (i.id=a.id_immeuble))
             join Personne as p on (a.id=p.id_appart)
   where i.id=1
```

On voit bien sur cet exemple que l'un des intérêts des vues est de donner une représentation "dénormalisée" de la base en regroupant des informations par des jointures. Le contenu étant virtuel, il n'y a ici aucun inconvénient à "voir" la redondance du nom de l'immeuble et de son adresse. Le bénéfice, en revanche, est la possibilité d'obtenir très simplement toutes les informations utiles.

```
select * from AppartKoudalou
```

no	surface	etage	immeuble	adresse	occupant
52	50	5	Koudalou	3 Rue Blanche	Alice Black
1	150	14	Koudalou	3 Rue Blanche	Rachel Verte
51	200	2	Koudalou	3 Rue Blanche	William Dupont

Le nom des attributs de la vue est celui des expressions de la requête associée. On peut également donner ces noms après le `create view` à condition qu'il y ait correspondance univoque entre un nom et une expression du `select`. On peut ensuite donner des droits en lecture sur cette vue pour que cette information limitée soit disponible à tous.

```
grant select on Immeuble.Koudalou Immeuble.AppartKoudalou to adminKoudalou
```

Pour peu que cet utilisateur n'ait aucun droit de lecture sur les tables de la base *Immeuble*, on obtient un moyen simple de masquer et restructurer l'information.

5.6.2 Mise à jour d'une vue

L'idée de modifier une vue peut sembler étrange puisqu'une vue n'a pas de contenu. En fait il s'agit bien entendu de modifier la table qui sert de support à la vue. Il existe de sévères restrictions sur les droits d'insérer ou de mettre à jour des tables au travers des vues. Un exemple suffit pour comprendre le problème. Imaginons que l'on souhaite insérer une ligne dans la vue *AppartKoudalou*.

```
insert into AppartKoudalou (no, surface, etage, immeuble, adresse, occupant)
values (1, 12, 4, 'Globe', '2 Avenue Leclerc', 'Palamède')
```

Le système rejettera cette requête (par exemple, pour MySQL, avec le message `Can not modify more than one base table through a join view 'Immeuble.AppartKoudalou'`). Cet ordre s'adresse à une vue issue de trois tables. Il n'y a clairement pas assez d'information pour alimenter ces tables de manière cohérente et l'insertion n'est pas possible (de même que toute mise à jour). De telles vues sont dites *non modifiables*. Les règles définissant les vues modifiables sont assez strictes et difficiles à résumer simplement d'autant qu'elles varient selon l'opération (`update`, `delete`, ou `insert`). En première approximation on peut retenir les points suivants qui donnent lieu à quelques exceptions sur lesquelles nous reviendrons ensuite.

- la vue doit être basée sur une seule table ; toute colonne non référencée dans la vue doit pouvoir être mise à `null` ou disposer d'une valeur par défaut ;
- on ne peut pas mettre à jour un attribut qui résulte d'un calcul ou d'une opération.

On ne peut donc pas insérer ou modifier la vue *Koudalou* à cause de la jointure et de l'attribut calculé. La requête suivante serait rejetée.

```
insert into Koudalou (nom, adresse)
values ('Globe', '2 Avenue Leclerc')
```

En revanche une vue portant sur une seule table avec un `select *` est modifiable.

```
create view PossedeAlice
   as select * from Possede
  where id_personne=2

insert into PossedeAlice values (2 100 20)
insert into PossedeAlice values (3 100 20)
```

Maintenant, si on fait :

```
select * from PossedeAlice
```

On obtient :

id_personne	id_appart	quote_part
2	100	20
2	103	100

L'insertion précédente illustre une petite subtilité : on peut insérer dans une vue sans être en mesure de voir la ligne insérée au travers de la vue par la suite ! On a en effet inséré dans la vue le propriétaire 3 qui est ensuite filtré quand on interroge la vue.

SQL propose l'option `with check option` qui permet de garantir que toute ligne insérée dans la vue satisfait les critères de sélection de la vue.

```
create view PossedeAlice
as select * from Possede
where id_personne=2
with check option
```

SQL permet également la modification de vues définies par des jointures. Les restrictions sont essentielement les même que pour les vues mono-tabulaires : on ne peut insérer que dans une des tables (il faut donc préciser la liste des attributs) et tous les attributs `not null` doivent avoir une valeur. Voici un exemple de vue modifiable basée sur une jointure.

```
create or replace view ToutKoudalou
as select i.id as id_imm, nom, adresse, a.*
   from Immeuble as i join Appart as a on (i.id=a.id_immeuble)
   where i.id=1
   with check option
```

Il est alors possible d'insérer à condition d'indiquer des attributs d'une seule des deux tables. La commande ci-dessous ajoute un nouvel appartement au *Koudalou*.

```
insert into ToutKoudalou (id, surface, etage, id_immeuble, no)
values (104, 70, 12, 1, 65)
```

En conclusion, l'intérêt principal des vues est de permettre une restructuration du schéma en vue d'interroger et/ou de protéger des données. L'utilisation de vues pour des mises à jour devrait rester marginale.

CHAPITRE 6

Le langage PL/SQL

Le langage SQL n'est pas un langage de programmation au sens courant du terme. Il ne permet pas, par exemple, de définir des fonctions ou des variables, d'effectuer des itérations ou des instructions conditionnelles. Il ne s'agit pas d'un défaut dans la conception du langage, mais d'une orientation délibérée de SQL vers les opérations de recherche de données dans une base volumineuse, la priorité étant donnée à la *simplicité* et à l'efficacité*. Ces deux termes ont une connotation forte dans le contexte d'un langage d'interrogation, et correspondent à des critères (et à des contraintes) précisément définis. La simplicité d'un langage est essentiellement relative à son caractère *déclaratif*, autrement dit à la capacité d'exprimer des recherches en laissant au système le soin de déterminer le meilleur moyen de les exécuter. L'efficacité est, elle, définie par des caractéristiques liées à la complexité d'évaluation sur lesquelles nous ne nous étendrons pas ici. Signalons cependant que la terminaison d'une requête SQL est *toujours* garantie, ce qui n'est pas le cas d'un programme écrit dans un langage plus puissant, .

Il est donc clair que SQL ne suffit pas pour le développement d'applications, et tous les SGBD relationnels ont, dès l'origine, proposé des interfaces permettant de l'associer à des langages plus classiques comme le C ou Java. Ces interfaces de programmation permettent d'utiliser SQL comme outil pour récupérer des données dans des programmes réalisant des tâches très diverses : interfaces graphiques, traitements "batch", production de rapports ou de sites web, etc. D'une certaine manière, on peut alors considérer SQL comme une interface d'accès à la base de données, intégrée dans un langage de programmation généraliste. Il s'agit d'ailleurs certainement de son utilisation la plus courante.

Pour certaines fonctionnalités, le recours à un langage de programmation "externe" s'avère cependant inadapté ou insatisfaisant. Une évolution des SGBD consiste donc à proposer, au sein même du système, des primitives de programmation qui viennent pallier le manque relatif d'expressivité des langages relationnnels. Le présent chapitre décrit ces évolutions et leur application à la création de *procédures stockées* et de *triggers*. Les premières permettent d'enrichir un schéma de base de données par des calculs ou des fonctions qui ne peuvent pas - parfois même dans des cas très simples - être obtenus avec SQL ; les seconds étendent la possibilité de définir des contraintes.

Parallèlement à ces applications pratiques, les procédures stockées illustrent simplement les techniques d'intégration de SQL à un langage de programmation classique, et soulignent les limites d'utilisation d'un langage d'interrogation, et plus particulièrement du modèle relationnel.

6.1 S1. Procédures stockées

Supports complémentaires :
— Diapositives: PL/SQL et triggers

Comme mentionné ci-dessus, les procédures stockées constituent une alternative à l'écriture de programmes avec une langage de programmation généraliste. Commençons par étudier plus en détail les avantages et inconvénients respectifs des deux solutions avant d'entrer dans les détails techniques.

6.1.1 Rôle et fonctionnement des procédures stockées

Une procédure stockée s'exécute au sein du SGBD, ce qui évite les échanges réseaux qui sont nécessaires quand les mêmes fonctionnalités sont implantées dans un programme externe communiquant en mode client/serveur avec la base de données. La figure *Comparaison programmes externes/procédures stockées* (page 78) illustre la différence entre les deux mécanismes. À gauche un programme externe, écrit par exemple en C, doit tout d'abord se connecter au serveur du SGBD. Le programme s'exécute alors en communiquant avec le serveur pour exécuter les requêtes et récupérer les résultats. Dans cette architecture, chaque demande d'exécution d'un ordre SQL implique une transmission sur le réseau, du programme vers le client, suivie d'une analyse de la requête par le serveur, de sa compilation et de son exécution (Dans certains cas les requêtes du programme client peuvent être précompilées, ou "préparées". Ensuite, chaque fois que le programme client souhaite récupérer un n-uplet du résultat, il doit effectuer un appel externe, via le réseau. Tous ces échanges interviennent de manière non négligeable dans la performance de l'ensemble, et cet impact est d'autant plus élevé que les communications réseaux sont lentes et/ou que le nombre d'appels nécessaires à l'exécution du programme est important.

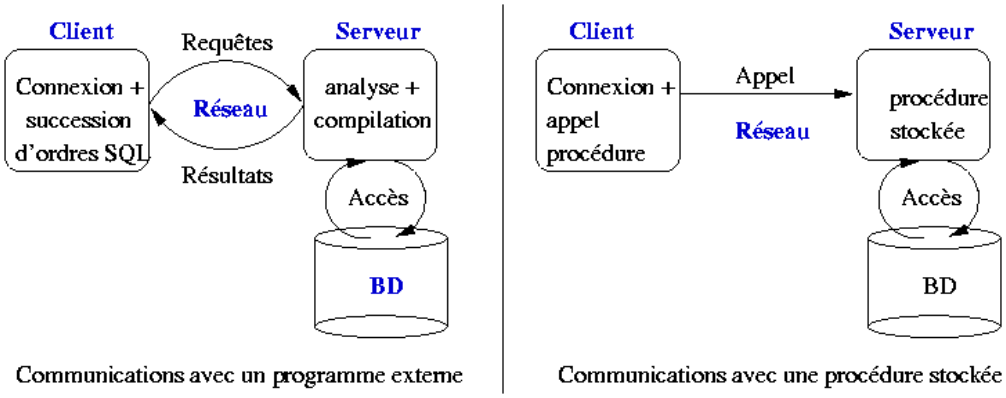

Fig. 6.1 – Comparaison programmes externes/procédures stockées

Le recours à une procédure stockée permet de regrouper du côté serveur l'ensemble des requêtes SQL et le traitement des données récupérées. La procédure est compilée une fois par le SGBD, au moment de sa création, ce qui permet de l'exécuter rapidement au moment de l'appel. De plus les échanges réseaux ne sont plus nécessaires puisque la logique de l'application est étroitement intégrée aux requêtes SQL. Le rôle du programme externe se limite alors à se connecter au serveur et à demander l'exécution de la procédure, en lui passant au besoin les paramètres nécessaires.

Bien entendu, en pratique, les situations ne sont pas aussi tranchées et le programme externe est en général amené à appeler plusieurs procédures, jouant en quelque sorte le rôle de coordinateur. Si les performances du système sont en cause, un recours judicieux aux procédures stockées reste cependant un bon moyen de réduire le trafic client-serveur.

L'utilisation de procédures stockées est par ailleurs justifiée, même en l'absence de problèmes de performance, pour des fonctions très "sensibles", terme qui recouvre (non exclusivement) les cas suivants :

1. la fonction est basée sur des règles complexes qui doivent être implantées très soigneusement ;
2. la fonction met à jour des données dont la correction et la cohérence sont indispensable au bon fonctionnement de l'application ;
3. la fonction évolue souvent.

On est souvent amené, quand on développe une application, à utiliser plusieurs langages en fonction du contexte : le langage C ou Java pour les traitements *batch*, PHP ou Python pour l'interface web, un générateur d'application propriétaire pour la saisie et la consultation à l'écran, un langage de script pour la production de rapports, etc. Il est important alors de pouvoir factoriser les opérations de base de données partagées par ces différents contextes, de manière à les rendre disponibles pour les différents langages utilisés. Par exemple la réservation d'un billet d'avion, ou l'exécution d'un virement bancaire, sont des opérations dont le fonctionnement correct (pas deux billets pour le même

siège ; pas de débit sans faire le crédit correspondant) et cohérent (les mêmes règles doivent être appliquées, quel que soit le contexte d'utilisation) doit toujours être assuré.

C'est facile avec une procédure stockée, et cela permet d'une part d'implanter une seule fois des fonctions "sensibles", d'autre part de garantir la correction, la cohérence et l'évolutivité en imposant l'utilisation systématique de ces fonctions au lieu d'un accès direct aux données.

Enfin, le dernier avantage des procédures stockées est la relative facilité de programmation des opérations de bases de données, en grande partie à cause de la très bonne intégration avec SQL. Cet aspect est favorable à la qualité et à la rapidité du développement, et aide également à la diffusion et à l'installation du logiciel puisque les procédures sont compilées par les SGBD et fonctionnent donc de manière identique sur toute les plateformes.

Il existe malheureusement une contrepartie à tous ces avantages : chaque éditeur de SGBD propose sa propre extension procédurale pour créer des procédures stockées, ce qui rend ces procédures incompatibles d'un système à un autre. Cela peut être dissuasif si on souhaite produire un logiciel qui fonctionne avec tous les SGBD relationnels.

La description qui suit se base sur le langage PL/SQL d'Oracle ("PL" signifie *Procedural Language*) qui est sans doute le plus riche du genre. Le même langage, simplifié, avec quelques variantes syntaxiques mineures, est proposé par PostgreSQL, et les exemples que nous donnons peuvent donc y être transposés sans trop de problème. Les syntaxes des langages utilisés par d'autres systèmes sont un peu différentes, mais tous partagent cependant un ensemble de concepts et une proximité avec SQL qui font de PL/SQL un exemple tout à fait représentatif de l'intérêt et de l'utilisation des procédures stockées.

6.1.2 Introduction à PL/SQL

Nous allons commencer par quelques exemples très simples, appliqués à la base *Films*, afin d'obtenir un premier aperçu du langage. Le premier exemple consiste en quelques lignes permettant d'afficher des statistiques sur la base de données (nombre de films et nombre d'artistes). Il ne s'agit pas pour l'instant d'une procédure stockée, mais d'un code qui est compilé et exécuté en direct.

```
-- Exemple de bloc PL/SQL donnant des informations sur la base

DECLARE
  -- Quelques variables
  v_nbFilms   INTEGER;
  v_nbArtistes INTEGER;

BEGIN
  -- Compte le nombre de films
  SELECT COUNT(*) INTO v_nbFilms FROM Film;
  -- Compte le nombre d'artistes
  SELECT COUNT(*) INTO v_nbArtistes FROM Artiste;

  -- Affichage des résultats
  DBMS_OUTPUT.PUT_LINE ('Nombre de films: ' || v_nbFilms);
  DBMS_OUTPUT.PUT_LINE ('Nombre d''artistes: ' || v_nbArtistes);

  EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE ('Problème rencontré dans StatsFilms');

END;
/
```

Le code est structuré en trois parties qui forment un "bloc" : déclarations des variables, instructions (entre `BEGIN` et `END`) et gestion des exceptions. La première remarque importante est que les variables sont typées, et que les types sont exactement ceux de SQL (ou plus largement les types supportés par le SGBD, qui peuvent différer légèrement

de la norme). Un autre aspect de l'intégration forte avec SQL est la possibilité d'effectuer des requêtes, d'utiliser dans cette requête des critères basés sur la valeur des variables de la procédure, et de placer le résultat dans une (ou plusieurs) variables grâce à la clause `INTO`. En d'autres termes on transfère directement des données représentées selon le modèle relationnel et accessibles avec SQL, dans des unités d'information manipulables avec les structures classiques (test ou boucles) d'un langage impératif.

Les fonctions SQL fournies par le SGBD sont également utilisables, ainsi que des librairies spécifiques à la programmation procédurale (des *packages* chez ORACLE). Dans l'exemple ci-dessus on utilise le *package* `DBMS_OUTPUT` qui permet de produire des messages sur la sortie standard (l'écran en général).

La dernière section du programme est celle qui gère les "exceptions". Ce terme désigne une erreur qui est soit définie par l'utilisateur en fonction de l'application (par exemple l'absence d'un n-uplet, ou une valeur incorrecte dans un attribut), soit engendrée par le système à l'exécution (par exemple une division par zéro, ou l'absence d'une table). Au moment où une erreur est rencontrée, PL/SQL redirige le flux d'exécution vers la section `EXCEPTION` où le programmeur doit définir les actions à entreprendre au cas par cas. Dans l'exemple précédent, on prend toutes les exceptions indifféremment (mot-clé `OTHERS` qui désigne le choix par défaut) et on affiche un message.

Note : Un programme Pl/SQL peut être placé dans un fichier et executé avec la commande `start` sous l'utilitaire de commandes. par exemple :

```
SQL>start StatsFilms
```

En cas d'erreur de compilation, la commande `SHOW ERRORS` donne la liste des problèmes rencontrés. Sinon le code est exécuté. Voici par exemple ce que l'on obtient avec le code donné précédemment en exemple (La commande `set serveroutput on` assure que les messages sont bien affichés à l'écran..

```
SQL> set serveroutput on
SQL> start StatsFilms
Nombre de films: 48
Nombre d'artistes: 126
```

Voici maintenant un exemple de procédure stockée. On retrouve la même structuration que précédemment (déclarations, instructions, exception), mais cette fois ce "bloc" est nommé, stocké dans la base au moment de la compilation, et peut ensuite être appelé par son nom. La procédure implante la règle suivante : l'insertion d'un texte dans la table des genres s'effectue toujours en majuscules, et on vérifie au préalable que ce code n'existe pas déjà.

```
-- Insère un nouveau genre, en majuscules, et en vérifiant
-- qu'il n'existe pas déjà

CREATE OR REPLACE PROCEDURE InsereGenre (p_genre VARCHAR) AS

  -- Déclaration des variables
  v_genre_majuscules VARCHAR(20);
  v_count INTEGER;
  genre_existe EXCEPTION;
BEGIN
  -- On met le paramètre en majuscules
  v_genre_majuscules := UPPER(p_genre);

  -- On vérifie que le genre n'existe pas déjà
  SELECT COUNT(*) INTO v_count
  FROM Genre WHERE code = v_genre_majuscules;

  -- Si on n'a rien trouvé: on insère
  IF (v_count = 0) THEN
   INSERT INTO Genre (code) VALUES (v_genre_majuscules);
```

```
  ELSE
   RAISE genre_existe;
 END IF;

  EXCEPTION
    WHEN genre_existe THEN
     DBMS_OUTPUT.PUT_LINE('Le genre existe déjà en ' ||
                          v_count || ' exemplaire(s).');

END;
/
```

La procédure accepte des paramètres qui, comme les variables, sont typées. Le corps de la procédure montre un exemple d'utilisation d'une fonction SQL fournie par le système, ici la fonction `UPPER` qui prend une chaîne de caractères en entrée et la renvoie mise en majuscules.

La requête SQL garantit que l'on obtient un et un seul n-uplet. Nous verrons plus loin comment traiter le cas où le résultat de la requête est une table contenant un nombre quelconque de n-uplets. Dans l'exemple ci-dessus on obtient toujours un attribut donnant le nombre de n-uplets existants dans la table pour le code que l'on veut insérer. Si ce nombre n'est pas nul, c'est que le genre existe déjà dans la table, et on produit une "exception" avec la clause `RAISE EXCEPTION`, sinon c'est que le genre n'existe pas et on peut effectuer la clause d'insertion, en indiquant comme valeurs à insérer celles contenues dans les variables appropriées.

On peut appeler cette procédure à partir de n'importe quelle application connectée au SGBD. Sous SQL*Plus on utilise l'instruction `execute`. Voici par exemple ce que l'on obtient avec deux appels successifs.

```
SQL>  execute InsereGenre('Policier');

SQL>  execute InsereGenre('Policier');
Le genre existe déjà en 1 exemplaire(s).
```

Le premier appel s'est correctement déroulé puisque le genre "Policier" n'existait pas encore dans la table. Le second en revanche a échoué, ce qui a déclenché l'exception et l'affichage du message d'erreur.

On peut appeler `InsereGenre()` depuis un programme C, Java, ou tout autre outil. Si on se fixe comme règle de toujours passer par cette procédure pour insérer dans la table `Genre`, on est donc sûr que les contraintes implantées dans la procédure seront toujours vérifiées.

Note : Pour forcer les développeurs à toujours passer par la procédure, on peut fixer les droits d'accès de telle sorte que les utilisateurs ORACLE aient le droit d'exécuter `InsereGenre()`, mais pas de droit de mise à jour sur la table `Genre` elle-même.

Voici un troisième exemple qui complète ce premier tour d'horizon rapide du langage. Il s'agit cette fois d'une *fonction*, la différence avec une procédure étant qu'elle renvoie une valeur, instance de l'un des types SQL. Dans l'exemple qui suit, la fonction prend en entrée l'identifiant d'un film et renvoie une chaîne de caractères contenant la liste des prénom et nom des acteurs du film, séparés par des virgules.

```
-- Fonction retournant la liste des acteurs pour un film donné

CREATE OR REPLACE FUNCTION MesActeurs(v_idFilm INTEGER) RETURN VARCHAR IS
  -- Le résultat
  resultat VARCHAR(255);

BEGIN
  -- Boucle prenant tous les acteurs du films
```

```
  FOR art IN
    (SELECT Artiste.* FROM Role, Artiste
    WHERE idFilm = v_idFilm  AND idActeur=idArtiste)
  LOOP

    IF (resultat IS NOT NULL) THEN
      resultat := resultat || ', ' || art.prenom || ' ' || art.nom;
    ELSE
      resultat := art.prenom || ' ' || art.nom;
   END IF;

  END LOOP;

 return resultat;
END;
/
```

La fonction effectue une requête SQL pour rechercher tous les acteurs du film dont l'identifiant est passé en paramètre. Contrairement à l'exemple précédent, cette requête renvoie en général plusieurs n-uplets. Une des caractéristiques principales des techniques d'accès à une base de données avec un langage procédural est que l'on ne récupère pas d'un seul coup le résultat d'un ordre SQL. Il existe au moins deux raisons à cela :

1. le résultat de la requête peut être extrêmement volumineux, ce qui poserait des problèmes d'occupation mémoire si on devait tout charger dans l'espace du programme client ;
2. les langages de programmation ne sont en général pas équipés nativement des types nécessaires à la représentation d'un ensemble de n-uplets.

Le concept utilisé, plus ou moins implicitement, dans toutes les interfaces permettant aux langages procéduraux d'accéder aux bases de données est celui de *curseur*. Un curseur permet de parcourir, à l'aide d'une boucle, l'ensemble des n-uplets du résultat d'une requête, en traitant le n-uplet courant à chaque passage dans la boucle. Ici nous avons affaire à la version la plus simple qui soit d'un curseur, mais nous reviendrons plus loin sur ce mécanisme.

Une fonction renvoie une valeur, ce qui permet de l'utiliser dans une requête SQL comme n'importe quelle fonction native du système. Voici par exemple une requête qui sélectionne le titre et la liste des acteurs du film dont l'identifiant est 5.

```
SQL> SELECT titre, MesActeurs(idFilm)  FROM Film WHERE idFilm=5;

TITRE        MESACTEURS(IDFILM)
------------ ----------------------------
Volte/Face   John Travolta, Nicolas Cage
```

On peut noter que le résultat de la fonction `MesActeurs()` ne peut pas être obtenu avec une requête SQL. Il est d'ailleurs intéressant de se demander pourquoi, et d'en tirer quelques conclusions sur certaines limites de SQL. Il est important de mentionner également qu'ORACLE ne permet pas l'appel, dans un ordre `SELECT` de fonctions effectuant des mises à jour dans la base : une requête n'est pas censée entraîner des modifications, surtout si elles s'effectuent de manière transparente pour l'utilisateur.

6.1.3 Syntaxe de PL/SQL

Voici maintenant une présentation plus systématique du langage PL/SQL. Elle vise à expliquer et illustrer ses principes les plus intéressants et à donner les éléments nécessaires à une expérimentation sur machine mais ne couvre cependant pas toutes ses possibilités, très étendues.

Types et variables

PL/SQL reconnaît tous les types standard de SQL, plus quelques autres dont le type `Boolean` qui peut prendre les valeurs `TRUE` ou `FALSE`. Il propose également deux constructeurs permettant de créer des types composés :

1. le constructeur `RECORD` est comparable au schéma d'une table ; il décrit un ensemble d'attributs typés et nommés ;
2. le constructeur `TABLE` correspond aux classiques tableaux unidimensionnels.

Le constructeur `RECORD` est particulièrement intéressant pour représenter un n-uplet d'une table, et donc pour définir des variables servant à stocker le résultat d'une requête SQL. On peut définir soit-même un type avec `RECORD`, avec une syntaxe très similaire à celle du `CREATE TABLE`.

```
DECLARE
  -- Déclaration d'un nouveau type
  TYPE adresse IS RECORD
   (no          INTEGER,
    rue         VARCHAR(40),
    ville       VARCHAR(40),
    codePostal VARCHAR(10)
   );
```

Mais PL/SQL offre également un mécanisme extrêmement utile consistant à dériver automatiquement un type `RECORD` en fonction d'une table ou d'un attribut d'une table. On utilise alors le nom de la table ou de l'attribut, associées respectivement au qualificateur `%ROWTYPE` ou à `%TYPE` pour désigner le type dérivé. Voici quelques exemples :

1. `Film.titre%TYPE` est le titre de l'attribut `titre` de la table `Film`;
2. `Artiste%ROWTYPE` est un type `RECORD` correspondant aux attributs de la table `Artiste`.

Le même principe de dérivation automatique d'un type s'applique également aux requêtes SQL définies dans le cadre des curseurs. Nous y reviendrons au moment de la présentation de ces derniers.

La déclaration d'une variable consiste à donner son nom, son type, à indiquer si elle peut être `NULL` et a donner éventuellement une valeur initiale. Elle est de la forme :

```
<nomVariable> <typeVariable> [NOT NULL] [:= <valeurDéfaut>]
```

Il est possible de définir également des constantes, avec la syntaxe :

```
<nomConstante> CONSTANT <typeConstante> := <valeur>
```

Toutes les déclarations de variables ou de constantes doivent être comprises dans la section `DECLARE`. Toute variable non initialisée est à `NULL`. Voici quelques exemples de déclarations. Tous les noms de variables sont systématiquement préfixés par `v_`. Ce n'est pas une obligation mais ce type de convention permet de distinguer plus facilement les variables de PL/SQL des attributs des tables dans les ordres SQL.

```
DECLARE
  -- Constantes
  v_aujourdhui  CONSTANT DATE  := SYSDATE;
  v_pi          CONSTANT NUMBER(7,5) := 3.14116;

  -- Variables scalaires
  v_compteur    INTEGER NOT NULL := 1;
  v_nom         VARCHAR(30);

  -- Variables pour un n-uplet de la table Film et pour le résumé
  v_film Film%ROWTYPE;
  v_resume Film.resume%TYPE;
```

Structures de contrôle

L'affectation d'une variable est effectuée par l'opérateur := avec la syntaxe :

```
<nomVariable> := <expression>;
```

où expression est toute expression valide retournant une valeur de même type que celle de la variable. Rappelons que tous les opérateurs SQL (arithmétiques, concaténation de chaînes, manipulation de dates) et toutes les fonctions du SGBD sont utilisables en PL/SQL. Un autre manière d'affecter une variable est d'y transférer tout ou partie d'un n-uplet provenant d'une requête SQL avec la syntaxe :

```
SELECT <nomAttribut1>, [<nomAttribut2>, ...]
INTO   <nomVariable1>, [<nomVariable2>, ... ]
FROM   [...]
```

La variable doit être du même type que l'attribut correspondant de la clause SELECT, ce qui incite fortement à utiliser le type dérivé avec %TYPE. Dès que l'on veut transférer plusieurs valeurs d'attributs dans des variables, on a sans doute intérêt à utiliser un type dérivé %ROWTYPE qui limite le nombre de déclarations à effectuer. L'exemple suivant illustre l'utilisation de la clause SELECT ... INTO associée à des types dérivés. La fonction renvoie le titre du film concaténé avec le nom du réalisateur.

```
-- Retourne une chaîne avec le titre du film et sont réalisateur

CREATE OR REPLACE FUNCTION TitreEtMES(v_idFilm INTEGER) RETURN VARCHAR IS

  -- Déclaration des variables
  v_titre Film.titre%TYPE;
  v_idMES Film.idMES%TYPE;
  v_mes  Artiste%ROWTYPE;

BEGIN
  -- Recherche du film
  SELECT  titre, idMES
  INTO v_titre, v_idMES
  FROM Film
  WHERE idFilm=v_idFilm;

  -- Recherche du metteur en scène
  SELECT * INTO v_mes FROM Artiste WHERE idArtiste = v_idMES;

  return v_titre  || ', réalisé par ' || v_mes.prenom
                  || ' ' || v_mes.nom;
END;
/
```

L'association dans la requête SQL de noms d'attributs et de noms de variables peut parfois s'avérer ambiguë d'où l'utilité d'une convention permettant de distinguer clairement ces derniers.

Les structures de test et de boucles sont tout à fait standard. La structure conditionnelle est le IF dont la syntaxe est la suivante :

```
IF <condition> THEN
  <instructions1>;
ELSE
 <instruction2>;
END IF;
```

Les conditions sont exprimées comme dans une clause WHERE de SQL, avec notamment la possibilité de tester si une

valeur est à NULL, des opérateurs comme LIKE et les connecteurs usuels AND, OR et NOT. Le ELSE est optionnel, et peut éventuellement être associé à un autre IF, selon la syntaxe généralisée suivante :

```
IF <condition 1> THEN
  <instructions 1>;
ELSIF <condition 2> THEN
 <instruction 2>;
  [...]
ELSIF <condition n> THEN
 <instruction n>;
ELSE
 <instruction n+1>;
END IF;
```

Il existe trois formes de boucles : LOOP, FOR et WHILE. Seules les deux dernières sont présentées ici car elles suffisent à tous les besoins et sont semblables aux structures habituelles.

La boucle WHILE répète un ensemble d'instructions tant qu'une condition est vérifiée. La condition est testée à chaque entrée dans la boucle. Voici la syntaxe :

```
WHILE <condition> LOOP
  <instructions>;
END LOOP;
```

Rappelons que les expressions booléennes en SQL peuvent prendre trois valeurs : TRUE, FALSE et UNKNOWN quand l'évaluation de l'expression rencontre une valeur à NULL. Une condition est donc vérifiée quand elle prend la valeur TRUE, et une boucle WHILE s'arrête en cas de FALSE ou UNKNOWN.

La boucle FOR permet de répéter un ensemble d'instructions pour chaque valeur d'un intervalle de nombres entiers. La syntaxe est donnée ci-dessous. Notez les deux points entre les deux bornes de l'intervalle, et la possibilité de parcourir cet intervalle de haut en bas avec l'option REVERSE.

```
FOR <variableCompteur> IN [REVERSE] <min>..<max> LOOP
  <instructions>;
END LOOP;
```

Des itérations couramment utilisées en PL/SQL consistent à parcourir le résultat d'une requête SQL avec un curseur.

Structure du langage

Le code PL/SQL est structuré en *blocs*. Un bloc comprend trois sections explicitement délimitées : les déclarations, les instructions (encadrée par begin et end) et les exceptions, placées en général à la fin de la section d'instruction. On peut partout mettre des commentaires, avec deux formes possibles : soit une ligne commençant par deux tirets --, soit, comme en C/C++, un texte de longueur quelconque compris entre /* et */. La structure générale d'un bloc est donc la suivante :

```
[DECLARE]
  -- Déclaration des variables, constantes, curseurs et exceptions

BEGIN
  -- Instructions, requêtes SQL, structures de contrôle

EXCEPTION
  -- Traitement des erreurs
END;
```

Le bloc est l'unité de traitement de PL/SQL. Un bloc peut être anonyme. Il commence alors par l'instruction DECLARE, et ORACLE le compile et l'exécute dans la foulée au moment où il est rencontré. Le premier exemple que nous avons donné est un bloc anonyme.

Un bloc peut également être nommé (cas des procédures et fonctions) et stocké. Dans ce cas le DECLARE est remplacé par l'instruction CREATE. Le SGBD stocke la procédure ou la fonction et l'exécute quand on l'appelle dans le cadre d'un langage de programmation. La syntaxe de création d'une procédure stockée est donnée ci-dessous.

```
CREATE [OR REPLACE] PROCEDURE <nomProcédure>
  [(<paramètre 1>, ... <paramètre n>)] AS
 [<déclarations<]
BEGIN
 <instructions>;

 [EXCEPTION
    <gestionExceptions>;
 ]
END;
```

La syntaxe des fonctions est identique, à l'exception d'un RETURN <type> précédant le AS et indiquant le type de la valeur renvoyée. Procédures et fonctions prennent en entrée des *paramètres* selon la syntaxe suivante :

```
<nomParamètre> [IN | OUT | IN OUT] <type> [:= <valeurDéfaut>]
```

La déclaration des paramètres ressemble à celle des variables. Tous les types PL/SQL sont acceptés pour les paramètres, notamment les types dérivés avec %TYPE et %ROWTYPE, et on peut définir des valeurs par défaut. Cependant la longueur des chaînes de caractères (CHAR ou VARCHAR) ne doit pas être précisée pour les paramètres. La principale différence avec la déclaration des variables est le mode d'utilisation des paramètres qui peut être IN, OUT ou IN OUT. Le mode détermine la manière dont les paramètres servent à communiquer avec le programme appelant :

1. IN indique que la valeur du paramètre peut être lue mais pas être modifiée ; c'est le mode par défaut ;
2. OUT indique que la valeur du paramètre peut être modifée mais ne peut pas être lue ;
3. IN OUT indique que la valeur du paramètre peut être lue et modifiée.

En d'autres termes les paramètres IN permettent au programme appelant de passer des valeurs à la procédure, les paramètres OUT permettent à la procédure de renvoyer des valeurs au programme appelant, et les paramètres IN OUT peuvent jouer les deux rôles. L'utilisation des paramètres OUT permet à une fonction ou une procédure de renvoyer plusieurs valeurs.

Gestion des erreurs

Les *exceptions* en PL/SQL peuvent être soit des erreurs renvoyées par le SGBD lui-même en cas de manipulation incorrecte des données, soit des erreurs définies par le programmeur lui-même. Le principe est que toute erreur rencontrée à l'exécution entraîne la levée} (RAISE) d'une exception, ce qui amène le flux de l'exécution à se dérouter vers la section EXCEPTION du bloc courant. Cette section rassemble les actions (*exception handlers*) à effectuer pour chaque type d'exception rencontrée. Voici quelques-unes des exceptions les plus communes levées par le SGBD.

1. INVALID_NUMBER, indique une conversion impossible d'une chaîne de caractères vers un numérique ;
2. INVALID_CURSOR, indique une tentative d'utiliser un nom de curseur inconnu ;
3. NO_DATA_FOUND, indique une requête SQL qui ne ramène aucun n-uplet ;
4. TOO_MANY_ROWS, indique une requête SELECT ... INTO qui n'est pas traitée par un curseur alors qu'elle ramène plusieurs n-uplets.

Les exceptions utilisateurs doivent être définies dans la section de déclaration avec la syntaxe suivante.

```
<nomException> EXCEPTION;
```

On peut ensuite lever, au cours de l'exécution d'un bloc PL/SQL, les exceptions, systèmes ou utilisateurs, avec l'instruction `RAISE`.

```
RAISE <nomException>;
```

Quand une instruction `RAISE` est rencontrée, l'exécution PL/SQL est dirigée vers la section des exceptions, et recherche si l'exception levée fait l'objet d'un traitement particulier. Cette section elle-même consiste en une liste de conditions de le forme :

```
WHEN <nomException> THEN
   <traitementException>;
```

Si le nom de l'exception levée correspond à l'une des conditions de la liste, alors le traitement correspondant est exécuté. Sinon c'est la section `OTHERS` qui est utilisée. S'il n'y a pas de section gérant les exceptions, l'exception est passée au programme appelant. La procédure suivante montre quelques exemples d'exceptions.

```
-- Illustration des exceptions. La procédure prend un
-- identifiant de film, et met le titre en majuscules.
-- Les exceptions suivantes sont levées:
--   Exception système: NO_DATA_FOUND si le film n'existe pas
--   Exception utilisateur: DEJA_FAIT si le titre
--            est déjà en majuscule

CREATE OR REPLACE PROCEDURE TitreEnMajuscules (p_idFilm INT) AS

  -- Déclaration des variables
  v_titre Film.titre%TYPE;
  deja_fait EXCEPTION;
BEGIN
  -- Recherche du film. Une exception est levée si on ne trouve rien
  SELECT titre INTO v_titre
  FROM Film WHERE idFilm = p_idFilm;

  -- Si le titre est déjà en majuscule, on lève une autre
  -- exception
  IF (v_titre = UPPER(v_titre)) THEN
      RAISE deja_fait;
  END IF;

  -- Mise à jour du titre
   UPDATE Film SET titre=UPPER(v_titre) WHERE idFilm=p_idFilm;

  EXCEPTION
    WHEN NO_DATA_FOUND THEN
     DBMS_OUTPUT.PUT_LINE('Ce film n''existe pas');

    WHEN deja_fait THEN
     DBMS_OUTPUT.PUT_LINE('Le titre est déjà en majuscules');

    WHEN OTHERS THEN
     DBMS_OUTPUT.PUT_LINE('Autre erreur...');
END;
/
```

Voici quelques exécutions de cette procédure qui montrent comment les exceptions sont levées selon le cas. On peut

noter qu'ORACLE considère comme une erreur le fait un ordre SELECT ne ramène aucun n-uplet, et lève alors l'exception NO_DATA_FOUND.

```
SQL> execute TitreEnMajuscules(900);
Le film n'existe pas

SQL> execute TitreEnMajuscules(5);

SQL> execute TitreEnMajuscules(5);
Le titre est déjà en majuscules
```

6.2 Les curseurs

Comme nous l'avons indiqué précédemment, les *curseurs* constituent un mécanisme de base dans les programmes accèdant aux bases de données. Ce mécanisme repose sur l'idée de traiter *un n-uplet à la fois* dans le résultat d'une requête, ce qui permet notamment d'éviter le chargement, dans l'espace mémoire du client, d'un ensemble qui peut être très volumineux.

Le traitement d'une requête par un curseur a un impact sur le style de programmation et l'intégration avec un langage procédural, sur les techniques d'évaluation de requêtes, et sur la gestion de la concurrence d'accès. Ces derniers aspects sont traités dans d'autres chapitres. La présentation qui suit est générale pour ce qui concerne les concepts, et s'appuie sur PL/SQL pour les exemples concrets. L'avantage de PL/SQL est de proposer une syntaxe très claire et un ensemble d'options qui mettent bien en valeur les points importants.

6.2.1 Déclaration d'un curseur

Un curseur doit être déclaré dans la section DECLARE du bloc PL/SQL. En général on déclare également une variable dont le type est dérivé de la définition du curseur. Voici un exemple de ces deux déclarations associées :

```
-- Déclaration d'un curseur
CURSOR MonCurseur IS
SELECT * FROM Film, Artiste
WHERE idMES = idArtiste;

-- Déclaration de la variable
v_monCurseur MonCurseur%ROWTYPE;
```

Le type de la variable, MonCurseur%ROWTYPE, est le type automatiquement calculé par PL/SQL pour représenter un n-uplet du résultat de la requête définie par le curseur MonCurseur. Ce typage permet, sans avoir besoin d'effectuer des déclarations et d'énumérer de longues listes de variables réceptrices au moment de l'exécution de la requête, de transférer très simplement chaque n-uplet du résultat dans une structure du langage procédural. Nous verrons que les choses sont beaucoup plus laborieuses avec un langage, comme le C, dont l'intégration avec SQL n'est pas du tout naturelle.

En général on utilise des curseurs *paramétrés* qui, comme leur nom l'indique, intègrent dans la requête SQL une ou plusieurs variables dont les valeurs, au moment de l'exécution, déterminent le résultat et donc l'ensemble de n-uplets à parcourir. Enfin on peut, optionnellement, déclarer l'intention de *modifier* les n-uplets traités par le curseur avec un UPDATE, ce qui entraîne au moment de l'exécution quelques conséquences importantes sur lesquelles nous allons revenir. La syntaxe générale d'un curseur est donc la suivante :

```
CURSOR <nomCurseur> [(<listeParamètres>)]
IS <requête>
[FOR UPDATE]
```

Les paramètres sont indiqués comme pour une procédure, mais le mode doit toujours être `IN` (cela n'a pas de sens de modifier le paramètre d'un curseur). Le curseur suivant effectue la même jointure que précédemment, mais les films sont sélectionnés sur l'année de parution grâce à un paramètre.

```
-- Déclaration d'un curseur paramétré
CURSOR MonCurseur (p_annee INTEGER) IS
 SELECT * FROM Film, Artiste
 WHERE idMES = idArtiste
 AND   annee = p_annee;
```

Une déclaration complémentaire est celle des variables qui vont permettre de recevoir les n-uplets au fur et à mesure de leur parcours. Le *type dérivé* d'un curseur est obtenu avec la syntaxe `<nomCurseur>\%ROWTYPE`. Il s'agit d'un type `RECORD` avec un champ par correspondant à expression de la clause `SELECT`. Le type de chaque champ est aisément déterminé par le système. Déterminer le nom du champ est un peu plus délicat car la clause `SELECT` peut contenir des attributs (c'est le plus courant) mais aussi des expressions construites sur ces attributs comme, par exemple, `AVG(annee)`. Il est indispensable dans ce dernier cas de donner un alias à l'expression, qui deviendra le nom du champ dans le type dérivé. Voici un exemple de cette situation :

```
-- Déclaration du curseur
CURSOR MonCurseur  IS
 SELECT prenom || nom AS nomRéalisateur, anneeNaiss, COUNT(*) AS nbFilms
 FROM Film, Artiste
 WHERE idMES = idArtiste
-- Déclaration d'une variable associée
v_realisateur MonCurseur%ROWTYPE;
```

Le type dérivé a trois champs, nommés respectivement `nomRealisateur`, `anneeNaiss` et `nbFilms`.

6.2.2 Exécution d'un curseur

Un curseur est toujours exécuté en trois phases :

1. ouverture du curseur (ordre `OPEN`) ;
2. parcours du résultat en itérant des ordres `FETCH` autant de fois que nécessaire ;
3. fermeture du curseur (`CLOSE`).

Il faut bien être conscient de la signification de ces trois phases. Au moment du `OPEN`, le SGBD va analyser la requête, construire un plan d'exécution (un programme d'accès aux fichiers) pour calculer le résultat, et initialiser ce programme de manière à être en mesure de produire un n-uplet dès qu'un `FETCH` est reçu. Ensuite, à chaque `FETCH`, le n-uplet courant est envoyé par le SGBD au curseur, et le plan d'exécution se prépare à produire le n-uplet suivant.

En d'autres termes le résultat est déterminé au moment du `OPEN`, puis exploité au fur est à mesure de l'appel des `FETCH`. Quelle que soit la période sur laquelle se déroule cette exploitation (10 secondes, 1 heure ou une journée entière), le SGBD doit assurer que les données lues par le curseur refléteront l'état de la base au moment de l'ouverture du curseur. Cela signifie notamment que les modifications effectuées par d'autres utilisateurs, ou par le programme client (c'est-à-dire celui qui exécute le curseur) lui-même, ne doivent pas être visibles au moment du parcours du résultat.

Les systèmes relationnels proposent différents niveaux d'isolation pour assurer ce type de comportement (pour en savoir plus, voir le chapitre sur la concurrence d'accès dans http://sys.bdpedia.fr). Il suffit d'imaginer ce qui se passerait si le curseur était sensible à des insertions, mises à jour ou suppressions effectuées pendant le parcours du résultat. Voici par exemple un pseudo-code montrant une situation où le parcours du curseur ne finirait jamais !

```
-- Un curseur qui s'exécute indéfiniment
OPEN du curseur sur la table T;
WHILE (FETCH du curseur ramène un n-upletdans T) LOOP
  Insérer un nouveau n-uplet dans T;
```

```
END LOOP;
CLOSE du curseur;
```

Chaque passage dans le WHERE entraîne l'insertion d'un nouveau n-uplet, et on se sortirait donc jamais de la boucle si le curseur prenait en compte ce dernier.

D'autres situations, moins caricaturales, et résultant d'actions effectuées par d'autres utilisateurs, poseraient des problèmes également. Le SGBD assure que le résultat est figé au moment du OPEN puisque c'est à ce moment-là que la requête est constituée et – au moins conceptuellement – exécutée.

Une solution triviale pour satisfaire cette contrainte est le calcul complet du résultat au moment du OPEN, et son stockage dans une table temporaire. Cette technique présente cependant de nombreux inconvénients :

1. il faut stocker le résultat quelque part, ce qui est pénalisant s'il est volumineux ;
2. le programme client doit attendre que l'intégralité du résultat soit calculé avant d'obtenir le premièr n-uplet ;
3. si le programme client souhaite effectuer des mises à jour, il faut réserver des n-uplets qui ne seront peut-être traités que dans plusieurs minutes voire plusieurs heures.

Dire que le résultat est *figé* ou *déterminé* à l'avance ne signifie par forcément qu'il est calculé et matérialisé quelque part. Les chapitres consacrés à l'évaluation de requêtes et à la concurrence d'accès dans http://sys.bdpedia.fr décrivent en détail les techniques plus sophistiquées pour gérer les curseurs. Ce qu'il faut retenir ici (et partout où nous parlerons de curseur), c'est que le résultat d'une requête n'est pas forcément pré-calculé dans son intégralité mais peut être construit, utilisé puis détruit au fur et à mesure de l'itération sur les ordres FETCH.

Ce mode d'exécution explique certaines restrictions qui semblent étranges si on n'en est pas averti. Par exemple un curseur ne fournit pas d'information sur le nombre de n-uplets du résultat, puisque ces n-uplets, parcourus un à un, ne permettent pas de savoir à l'avance combien on va en rencontrer. De même, on ne sait pas revenir en arrière dans le parcours d'un résultat puisque les n-uplets produits ne sont parfois pas conservés.

Il existe dans la norme une option SCROLL indiquant que l'on peut choisir d'aller en avançant ou en reculant sur l'ensemble des n-uplets. Cette option n'est disponible dans aucun système, du moins à l'heure où ces lignes sont écrites. Le SCROLL est problématique puisqu'il impose de conserver au SGBD le résultat complet pendant toute la durée de vie du curseur, l'utilisateur pouvant choisir de se déplacer d'avant en arrière sur l'ensemble des n-uplets. Le SCROLL est difficilement compatible avec la technique d'exécution employés dans tous les SGBD, et qui ne permet qu'un seul parcours séquentiel sur l'ensemble du résultat.

6.2.3 Les curseurs PL/SQL

La gestion des curseurs dans PL/SQL s'appuie sur une syntaxe très simple et permet, dans un grand nombre de cas, de limiter au maximum les déclarations et instructions nécessaires. La manière la plus générale de traiter un curseur, une fois sa déclaration effectuée, et de s'appuyer sur les trois instructions OPEN, FETCH et CLOSE dont la syntaxe est donnée ci-dessous.

```
OPEN <nomCurseur>[(<valeursParamètres>)];
FETCH <nomCurseur> INTO <variableRéceptrice>;
CLOSE <nomCurseur>;
```

La (ou les) variable(s) qui suivent le INTO doivent correspondre au type d'un n-uplet du résultat. En général on utilise une variable déclarée avec le type dérivé du curseur, <nomCurseur>%ROWTYPE.

Une remarque importante est que les curseurs ont l'inconvénient d'une part de consommer de la mémoire du côté serveur, et d'autre part de bloquer d'autres utilisateurs si des n-uplets doivent être réservés en vue d'une mise à jour (option FOR UPDATE). Une bonne habitude consiste à effectuer le OPEN le plus tard possible, et le CLOSE le plus tôt possible après le dernier FETCH.

Au cours de l'accès au résultat (c'est-à-dire après le premier FETCH et avant le CLOSE), on peut obtenir les informations suivantes sur le statut du curseur.

1. <nomCurseur>%FOUND est un booléen qui vaut TRUE si le dernier FETCH a ramené un n-uplet ;
2. <nomCurseur>%NOTFOUND est un booléen qui vaut TRUE si le dernier FETCH n'a pas ramené de n-uplet ;
3. <nomCurseur>%ROWCOUNT est le nombre de n-uplets parcourus jusqu'à l'état courant (en d'autres termes c'est le nombre d'appels FETCH) ;
4. <nomCurseur>%ISOPEN est un boolén qui indique si un curseur a été ouvert.

Cela étant dit, le parcours d'un curseur consiste à l'ouvrir, à effectuer une boucle en effectuant des FETCH tant que l'on trouve des n-uplets (et qu'on souhaite continuer le traitement), enfin à fermer le curseur. Voici un exemple assez complet qui utilise un curseur paramétré pour parcourir un ensemble de films et leurs metteur en scène pour une année donnée, en afichant à chaque FETCH le titre, le nom du metteur en scène et la liste des acteurs. Remarquez que cette liste est elle-même obtenu par un appel à la fonction PL/SQL MesActeurs.

```
-- Exemple d'un curseur pour rechercher les films
-- et leur metteur en scène pour une année donnée

CREATE OR REPLACE PROCEDURE CurseurFilms (p_annee INT) AS

  -- Déclaration d'un curseur paramétré
  CURSOR MonCurseur (v_annee INTEGER) IS
   SELECT idFilm, titre, prenom, nom
   FROM Film, Artiste
   WHERE idMES = idArtiste
   AND   annee = v_annee;

  -- Déclaration de la variable associée au curseur
  v_monCurseur MonCurseur%ROWTYPE;
  -- Déclaration de la variable pour la liste des acteurs
  v_mesActeurs VARCHAR(255);
BEGIN
  -- Ouverture du curseur
  OPEN MonCurseur(p_annee);

  -- On prend le premier n-uplet
  FETCH MonCurseur INTO v_monCurseur;
  -- Boucle sur les n-uplets
  WHILE (MonCurseur%FOUND) LOOP
    -- Recherche des acteurs avec la fonction MesActeurs
    v_mesActeurs := MesActeurs (v_monCurseur.idFilm);

    DBMS_OUTPUT.PUT_LINE('Ligne ' || MonCurseur%ROWCOUNT ||
           ' Film:  ' || v_monCurseur.titre ||
           ', de ' || v_monCurseur.prenom || ' ' ||
           v_monCurseur.nom || ', avec ' || v_mesActeurs);
    -- Passage au n-uplet suivant
    FETCH MonCurseur INTO v_monCurseur;
  END LOOP;
  -- Fermeture du curseur
  CLOSE MonCurseur;

  EXCEPTION
    WHEN OTHERS THEN
     DBMS_OUTPUT.PUT_LINE('Problème dans CurseurFilms : ' ||
                          sqlerrm);

END;
/
```

Le petit extrait d'une session sous SQL*Plus donné ci-dessous montre le résultat d'un appel à cette procédure pour

l'année 1992.

```
SQL> set serveroutput on
SQL> execute CurseurFilms(1992);
Ligne 1 Film:  Impitoyable, de Clint Eastwood, avec
         Clint Eastwood, Gene Hackman, Morgan Freeman
Ligne 2 Film:  Reservoir dogs, de Quentin Tarantino, avec
          Quentin Tarantino, Harvey Keitel, Tim Roth, Chris Penn
```

La séquence des instructions `OPEN`, `FETCH` et `CLOSE` et la plus générale, notamment parce qu'elle permet de s'arrêter à tout moment en interrompant la boucle. On retrouve cette structure dans les langages de programmations comme C, Java et PHP. Elle a cependant l'inconvénient d'obliger à écrire deux instructions `FETCH`, l'une avant l'entrée dans la boucle, l'autre à l'intérieur. PL/SQL propose une syntaxe plus concise, basée sur la boucle `FOR`, en tirant partie de sa forte intégration avec SQL qui permet d'inférer le type manipulé en fonction de la définition d'un curseur. Cette variante de `FOR` se base sur la syntaxe suivante :

```
FOR <variableCurseur> IN <nomCurseur> LOOP
  <instructions;
END LOOP;
```

L'économie de cette construction vient du fait qu'il n'est nécessaire ni de déclarer la variable `variableCurseur`, ni d'effectuer un `OPEN`, un `CLOSE` ou des `FETCH`. Tout est fait automatiquement par PL/SQL, la variable étant définie uniquement dans le contexte de la boucle. Voici un exemple qui montre également comment traiter des mises sur les n-uplets sélectionnés.

```
-- Exemple d'un curseur effectuant des mises à jour
-- On parcourt la liste des genres, et on les met en majuscules,
-- on détruit ceux qui sont à NULL

CREATE OR REPLACE PROCEDURE CurseurMAJ  AS

  -- Déclaration du curseur
  CURSOR CurseurGenre IS
   SELECT * FROM Genre  FOR UPDATE;
BEGIN
  -- Boucle FOR directe: pas de OPEN, pas de CLOSE

  FOR v_genre IN CurseurGenre LOOP
   IF (v_genre.code IS NULL) THEN
     DELETE FROM Genre WHERE CURRENT OF CurseurGenre;
   ELSE
     UPDATE Genre SET code=UPPER(code)
     WHERE CURRENT OF CurseurGenre;
   END IF;
  END LOOP;
END;
/
```

Notez que la variable `v_genre` n'est pas déclarée explicitement. Le curseur est défini avec une clause `FOR UPDATE` qui indique au SGBD qu'une mise à jour peut être effectuée sur chaque n-uplet. Dans ce cas – et dans ce cas seulement – il est possible de faire référence au n-uplet courant, au sein de la boucle `FOR`, avec la syntaxe `WHERE CURRENT OF <nomCurseur>`.

Si on n'a pas utilisé la clause `FOR UPDATE`, il est possible de modifier (ou détruire) le n-uplet courant, mais en indiquant dans le `WHERE` de la clause `UPDATE` la valeur de la clé. Outre la syntaxe légèrement moins concise, cette désynchronisation entre la lecture par le curseur, et la modification par SQL, entraîne des risques d'incohérence (mise à jour par un autre utilisateur entre le `OPEN` et le `FETCH`) qui sont développés dans le chapitre consacré à la concurrence d'accès (http://sys.bdpedia.fr).

Il existe une syntaxe encore plus simple pour parcourir un curseur en PL/SQL. Elle consiste à ne pas déclarer explicitement de curseur, mais à placer la requête SQL directement dans la boucle `FOR`, comme par exemple :

```
FOR v_genre IN (SELECT * FROM Genre) LOOP
  <instructions;
END LOOP;
```

Signalons pour conclure que PL/SQL traite *toutes* les requêtes SQL par des curseurs, que ce soit des ordres `UPDATE`, `INSERT`, `DELETE` ou des requêtes `SELECT` ne ramenant qu'une seule ligne. Ces curseurs sont "implicites" car non déclarés par le programmeur, et tous portent le même nom conventionnel, `SQL`. Concrètement, cela signifie que les valeurs suivantes sont définies après une mise à jour par `UPDATE`, `INSERT`, `DELETE` :

1. `SQL%FOUND` vaut `TRUE` si la mise à jour a affecté au moins un n-uplet ;
2. `SQL%NOTFOUND` vaut `TRUE` si la mise à jour n'a affecté aucun n-uplet ;
3. `SQL%ROWCOUNT` est le nombre de n-uplets affecté par la mise à jour ;
4. `SQL%ISOPEN` renvoie systématiquement `FALSE` puisque les trois phases (ouverture, parcours et fermeture) sont effectuées solidairement.

Le cas du `SELECT` est un peu différent : une exception est toujours levée quand une recherche sans curseur ne ramène pas de n-uplet (exception `NO_DATA_FOUND`) ou en ramène plusieurs (exception `TOO_MANY_ROWS`). Il faut donc être prêt à traiter ces exceptions pour ce type de requête. Par exemple, la recherche :

```
SELECT * INTO v_film
FROM Film
WHERE titre LIKE 'V%';
```

devrait être traitée par un curseur car il y n'y a pas de raison qu'elle ramène un seul n-uplet.

6.3 Triggers

Le mécanisme de *triggers* (que l'on peut traduire par "déclencheur" ou "réflexe") est implanté dans les SGBD depuis de nombreuses années, et a été normalisé par SQL99. Un *trigger* est simplement une procédure stockée dont la particularité principale est de se déclencher automatiquement sur certains événements mise à jour *spécifiés par le créateur du trigger*.

On peut considérer les *triggers* comme une extension du système de contraintes proposé par la clause `CHECK` : à la différence de cette dernière, l'événement déclencheur est explicitement indiqué, et l'action n'est pas limitée à la simple alternative acceptation/rejet. Les possibilités offertes par les *triggers* sont très intéressantes. Citons :

— la gestion des redondances ; l'enregistrement automatique de certains évènenements (*auditing*) ;
— la spécification de contraintes complexes liées à l'évolution des données (exemple : le prix d'une séance ne peut qu'augmenter) ;
— toute règle liée à l'environnement d'exécution (restrictions sur les horaires, les utilisateurs, etc.).

Les *triggers* sont discutés dans ce qui suit de manière générale, et illustrés par des exemples ORACLE. Il faut mentionner que la syntaxe de déclaration des *triggers* est suivie par la plupart des SGBD, les principales variantes se situant au niveau du langage permettant d'implanter la partie procédurale. Dans ce qui suit, ce langage sera bien entendu PL/SQL.

6.3.1 Principes des *triggers*

Le modèle d'exécution des *triggers* est basé sur la séquence *événement-Condition-Action* (ECA) que l'on peut décrire ainsi :

— un *trigger* est déclenché par un *évènement*, spécifié par le programmeur, qui est en général une insertion, destruction ou modification sur une table ;

— la première action d'un *trigger* est de tester une *condition* : si cette condition ne s'évalue pas à `TRUE`, l'exécution s'arrête ;
— enfin *l'action* proprement dite peut consister en toute ensemble d'opérations sur la base de données, effectuée si nécessaire à l'aide du langage procédural supporté par le SGBD.

Une caractéristique importante de cette procédure (action) est de pouvoir manipuler simultanément les valeurs ancienne et nouvelle de la donnée modifiée, ce qui permet de faire des tests sur l'évolution de la base.

Parmi les autres caractéristiques importantes, citons les deux suivantes. Tout d'abord un *trigger* peut être exécuté au choix une fois pour un seul ordre SQL, ou à chaque n-uplet concerné par cet ordre. Ensuite l'action déclenchée peut intervenir *avant* l'événement, ou *après*.

L'utilisation des *triggers* permet de rendre une base de données *dynamique* : une opération sur la base peut en déclencher d'autres, qui elles-mêmes peuvent entraîner en cascade d'autres réflexes. Ce mécanisme n'est pas sans danger à cause des risques de boucle infinie.

Prenons l'exemple suivant : on souhaite conserver au niveau de la table `Cinéma` le nombre total de places (soit la somme des capacités des salles). Il s'agit en principe d'une redondance à éviter en principe, mais que l'on peut gérer avec les *triggers*. On peut en effet implanter un *trigger* au niveau `Salle` qui, pour toute mise à jour, va aller modifier la donnée au niveau `Cinéma`.

Maintenant il est facile d'imaginer une situation où on se retrouve avec des *triggers* en cascade. Prenons le cas d'une table `Ville (nom,capacité)` donnant le nombre de places de cinéma dans la ville.

Maintenant, supposons que la ville gère l'heure de la première séance d'une salle : on aboutit à un cycle infini !

6.3.2 Syntaxe

La syntaxe générale de création d'un *trigger* est donnée ci-dessous.

```
 CREATE [OR REPLACE] TRIGGER <nomTrigger>
  {BEFORE | AFTER}
  {DELETE | INSERT | UPDATE [of column, [, column] ...]}
     [ OR {DELETE | INSERT | UPDATE [of column, [, column] ...]}] ...
  ON <nomTable> [FOR EACH ROW]
[WHEN <condition]
   <blocPLSQL>
```

On peut distinguer trois parties dans cette construction syntaxique. La partie *événement* est spécifiée après `BEFORE` ou `AFTER`, la partie *condition* après `WHEN` et la partie *action* correspond au bloc PL/SQL. Voici quelques explications complémentaires sur ces trois parties.

— "Evénement", peut être ' BEFORE'' ou `AFTER`, suivi de `DELETE`, `UPDATE` ou `INSERT` séparés par des `OR`.
— "Condition", `FOR EACH ROW` est optionnel. En son absence le *trigger* est déclenché une fois pour toute requête modifiant la table, *et ce sans condition*.
  Sinon `<condition>` est toute condition booléenne SQL. De plus on peut rérférencer les anciennes et nouvelles valeurs du tuple courant avec la syntaxe `new.attribut` et '' old.attribut'' respectivement.
— "Action" est une procédure qui peut être implantée, sous Oracle, avec le langage PL/SQL. Elle peut contenir des ordres SQL *mais pas de mise à jour de la table courante*.
  Les anciennes et nouvelles valeurs du tuple courant sont référencées par `:new.attr` et `:old.attr`.
  Il est possible de modifier `new` et `old`. Par exemple `:new.prix=500;` forcera l'attribut `prix` à 500 dans un `BEFORE` *trigger*.

La disponibilité de `new` et `old` dépend du contexte. Par exemple `new` est à `NULL` dans un *trigger* déclenché par `DELETE`.

6.3.3 Quelques exemples

Voici tout d'abord un exemple de *trigger* qui maintient la capacité d'un cinéma à chaque mise à jour sur la table `Salle`.

```
CREATE TRIGGER CumulCapacite
AFTER UPDATE ON Salle
FOR EACH ROW
WHEN (new.capacite != old.capacite)
BEGIN
  UPDATE Cinema
  SET capacite = capacite - :old.capacite  + :new.capacite
  WHERE  nom = :new.nomCinema;
END;
```

Pour garantir la validité du cumul, il faudrait créer des *triggers* sur les événements `UPDATE` et `INSERT`. Une solution plus concise (mais plus coûteuse) est de recalculer systématiquement le cumul : dans ce cas on peut utiliser un *trigger* qui se déclenche globalement pour la requête :

```
CREATE TRIGGER CumulCapaciteGlobal
AFTER UPDATE OR INSERT OR DELETE ON Salle
BEGIN
  UPDATE Cinema C
  SET capacite = (SELECT SUM (capacite)
                  FROM   Salle S
                  WHERE  C.nom = S.nomCinema);
END;
```

CHAPITRE 7

Indices and tables

— genindex
— modindex
— search