Cours de bases de données - SQL : fondements et pratiques

Maxime Buron

Contents

1 Introduction **2**
- 1.1 Contenu et plan du cours . . . 2
- 1.2 Apprendre avec ce cours . . . 3
- 1.3 Avancement . . . 3
- 1.4 S1: notions de base . . . 3
  - 1.4.1 Données, bases de données et SGBD . . . 4
  - 1.4.2 Modèle et couches d'abstraction . . . 7
  - 1.4.3 Les langages . . . 9

2 Le modèle relationnel **10**
- 2.1 Qu'est-ce qu'une relation . . . 10
- 2.2 Le schéma . . . 10
- 2.3 L'instance d'une relation . . . 11
- 2.4 Représentation des relations . . . 12
- 2.5 Mais que représente une relation ? . . . 13

3 SQL, langage déclaratif **14**
  - 3.0.1 SQL est-il *totalement* déclaratif ? . . . 14
- 3.1 S1: Un peu de logique . . . 15
- 3.2 S2: SQL conjonctif . . . 16
  - 3.2.1 La base des voyageurs . . . 16
  - 3.2.2 Requête mono-variable . . . 18
  - 3.2.3 Requêtes multi-variables et jointures . . . 20
- 3.3 S3: Quantificateurs et négation . . . 23
  - 3.3.1 Le quantificateur `exists` . . . 23
  - 3.3.2 Quantificateurs et négation . . . 25
- 3.4 S4: Conception d'une requête SQL . . . 26
  - 3.4.1 Conception d'une jointure . . . 27
  - 3.4.2 Conception des requêtes imbriquées . . . 31
  - 3.4.3 La disjonction . . . 31

4 SQL, compléments et supports **32**
- 4.1 S1: le bloc `select-from-where` . . . 32
  - 4.1.1 La clause `from` . . . 33
  - 4.1.2 La clause `where` . . . 34
  - 4.1.3 Valeurs manquantes: le `null` . . . 35
  - 4.1.4 La clause `select` . . . 36
  - 4.1.5 Jointure interne, jointure externe . . . 38
  - 4.1.6 Tri et élimination de doublons . . . 41
- 4.2 S2: Requêtes et sous-requêtes . . . 42
  - 4.2.1 Requêtes imbriquées . . . 43

4.2.2 Requêtes correlées . . . 45
4.2.3 Requêtes avec négation . . . 46
4.3 S3: Agrégats . . . 47
4.3.1 La clause `group by` . . . 48
4.3.2 Quelques exemples . . . 49
4.3.3 La clause `having` . . . 50

5 SQL, langage algébrique 51
5.1 S1: Les opérateurs de l'algèbre . . . 52
5.1.1 La projection, π . . . 53
5.1.2 La sélection, σ . . . 53
5.1.3 Le produit cartésien, $\times$ . . . 54
5.1.4 Renommage . . . 55
5.1.5 L'union, $\cup$ . . . 57
5.1.6 La différence, $-$ . . . 58
5.2 S2: la jointure . . . 58
5.2.1 L'opérateur $\bowtie$ . . . 58
5.2.2 Résolution des ambiguités . . . 61
5.3 S3: Expressions algébriques . . . 62
5.3.1 Sélection généralisée . . . 63
5.3.2 Requêtes conjonctives . . . 63
5.3.3 Requêtes avec $\cup$ et $-$ . . . 65
5.3.4 Complément d'un ensemble . . . 66
5.3.5 Quantification universelle . . . 66

1 Introduction

Ce support de cours s'adresse à tous ceux qui veulent découvrir le langage SQL, son utilisation en pratique et ses bases fondamentales. Il s'agit d'être capable de comprendre la structuration d'une base de données existante et de l'interroger et de comprendre quelques fondements du langage d'interrogation SQL.

Ce cours est librement inspiré du cours de Philippe Rigaux Cours de bases de données - Modèles et langages. Il est mis à disposition sous la même licence Creative Commons Attribution : `https://creativecommons.org/licenses/by-nc-sa/4.0/`

1.1 Contenu et plan du cours

Le cours est constitué de quelques chapitres consacrés au modèle de données *relationnelles* et à son interrogation via l'algèbre relationnelle et le langage SQL. Il allie la vue conceptuelle de ce modèle de données, à sa pratique dans les Systèmes de Gestion de Bases de Données (SGBD). Il couvre les modèles et langages des bases de données, et plus précisément :

- la notion de *modèle de données* qui décrit la structure d'une base de données,
- les principes des langages d'interrogation, avec les deux paradigmes principaux : *déclaratif* (on décrit ce que l'on veut obtenir sans dire comment on peut l'obtenir) et *procédural* (on applique une suite d'opérations sur la base),
- la rédaction en pratique de requêtes SQL, qui soient claires

1.2 Apprendre avec ce cours

Le cours est découpé en *chapitres*, couvrant un sujet bien déterminés, et en *sessions*. Les sessions sont structurées pour que les concepts principaux puissent être présentés dans une vidéo d'à peu près 20 minutes (réalisées par le créateur de ce support Philippe Rigaux). J'estime que chaque session demande environ 2 heures de travail personnel (bien sûr, cela dépend également de vous). Pour assimiler une session vous pouvez combiner les ressources suivantes:

- La lecture du support : celui que vous avez sous les yeux, disponibles en PDF ou en ligne.
- Le suivi du cours consacré à la session en vidéo

1.3 Avancement

- Semaine 3 : TD n°1, avant la séance, lire jusqu'à la Section 3.2 incluse.
- Semaine 4 : TP n°1, lire la Section 3.3.
- Semaine 5 : TD n°2, lire la Section 3.4
- Semaine 6 : TD n°3
- Semaine 7 : TP n°2, lire la Section 4.1
- Semaine 8 : VACANCES ! (pensez à regarder le projet)
- Semaine 9 : TD n°4, avant la séance, lire la Section 4.3 (attention, on laisse de côté une section, pour commencer les requêtes avec aggrégations !)
- Semaine 10 : TP n°3, avant la séance, lire la Section 4.2
- Semaine 11 : TD n°5, avant la séance, lire la Section 5.1
- Semaine 12 : Rien à faire (à part vos examens)
- Semaine 13 : TP n°4, lire la Section 5.2
- Semaine 14 : TD n°6, avant la séance lire la dernière Section 5.3
- Semaine 15 et 16 : Rien
- Semaine 17 : TP noté

1.4 S1: notions de base

Supports complémentaires:

- Diapositives: notions de base
- Vidéo sur les notions de base

Entrons directement dans le vif du sujet avec un premier tour d'horizon qui va nous permettre de situer les principales notions étudiées dans ce cours. Cette session présente sans doute beaucoup de concepts dont certains s'éclairciront au fur et à mesure de l'avancement dans le cours. À lire et relire régulièrement donc.

1.4.1 Données, bases de données et SGBD

Nous appellerons *donnée* toute valeur numérisée décrivant de manière élémentaire un fait, une mesure, une réalité. Ce peut être une chaîne de caractères (« bouvier »), un entier (365), une date (12/07/1998). Cette valeur est toujours associée au contexte permettant de savoir quelle information elle représente. Un mot comme « bouvier » par exemple peut désigner, entre autres, un gardien de troupeau, un aimable petit insecte, ou le nom d'un écrivain célèbre. Il ne prend un peu de sens que si l'on sait l'interpréter. Une donnée se présente toujours en association avec un contexte interprétatif qui permet de lui donner un sens.

Note

On pourrait établir une distinction (subtile) entre donnée (valeur brute) et information (valeur et contexte interprétatif). Pour ne pas compliquer inutilement les choses, on va assimiler les deux notions dans ce qui suit.

Les données ne tombent pas du ciel, et elles ne sont pas mises en vrac dans un espace de stockage. Elles sont issues d'un domaine applicatif, et décrivent des objets, des faits ou des concepts (on parle plus généralement *d'entités*). On les organise de manière à ce que ces entités soient correctement et uniformément représentées, ainsi que les liens que ces entités ont les unes avec les autres. Si je prends par exemple l'énoncé *Nicolas Bouvier est un écrivain suisse auteur du récit de voyage culte « L'usage du monde » paru en 1963*, je peux en extraire le prénom et le nom d'une personne, sa nationalité (données décrivant une première entité), et au moins un de ses ouvrages (seconde entité, décrite par un titre et une année de parution). J'ai de plus une notion d'auteur qui relie la première à la seconde. Tout cela constitue autant d'informations indissociables les uns des autres, constituant une ébauche d'une base de données consacrée aux écrivains et à leurs œuvres.

La représentation de ces données et leur association donne à la base une *structure* qui aide à distinguer précisément et sans ambiguité les informations élémentaires constituant cette base: nom, prénom, année de naissance, livre(s) publié(s), etc. Une base sans structure n'a aucune utilité. Une base avec une structure incorrecte ou incomplète est une source d'ennuis infinis. Nous verrons comment la structure doit être très sérieusement définie pendant la phase de conception.

Une *base de données* est un ensemble (potentiellement volumineux, mais pas forcément) de telles informations conformes à une structure pré-définie au moment de la conception, avec, de plus, une caractéristique essentielle : on souhaite les mémoriser de manière *persistante*. La persistance désigne la capacité d'une base à exister indépendamment des applications qui la manipulent, ou du système qui l'héberge. On peut arrêter toutes les machines un soir, et retrouver la base de données le lendemain. Cela implique qu'une base est *toujours* stockée sur un support comme les disques magnétiques qui préservent leur contenu même en l'absence d'alimentation électrique.

Important: Les supports persistants (disques, SSD) sont très lents par rapport aux capacités d'un processeur et de sa mémoire interne. La nécessité de stocker une base sur un tel support soulève donc de redoutables problèmes de performance, et a mené à la mise au point de techniques très sophistiquées, caractéristiques des systèmes de gestion de données. Ces techniques sont étudiées dans un cours consacré aux aspects systèmes (en L3).

On en arrive donc à la définition suivante:

Définition (base de données)

Une base de données est un ensemble d'informations structurées mémorisées sur un support *persistant*.

Remarquons qu'une organisation consistant à stocker nos données dans un (ou plusieurs) fichier(s) sur le disque de notre ordinateur personnel peut très bien être considéré comme con-

forme à cette définition, sous réserve qu'elles soient un tant soit peu structurées. Les fichiers produits par votre traitement de texte préféré par exemple ne font pas l'affaire: on y trouve certes des données, mais pas leur association à un contexte interprétatif non ambigu. Ecrire avec ce traitement de texte une phrase comme « L'usage du monde est un livre de Nicolas Bouvier paru en 1963 » constitue un énoncé trop flou pour qu'un système puisse automatiquement en extraire (sans recourir à des techniques très sophistiquées et en partie incertaines) le nom de l'auteur, le titre de son livre, ou sa date de parution.

Un fichier de base de données a nécessairement une structure qui permet d'une part de distinguer les données les unes des autres, et d'autre part de représenter leurs liens. Prenons l'exemple de l'une des structures les plus simples et les plus répandues, les fichiers CSV. Dans un fichier CSV, les données élémentaires sont réprésentés par des « champs » délimités par des points-virgule. Les champs sont associés les uns aux autres par le simple fait d'être placés dans une même ligne. Les lignes en revanche sont indépendantes les unes des autres. On peut placer autant de lignes que l'on veut dans un fichier, et même changer leur ordre sans que cela modifie en quoi que ce soit l'information représentée.

Voici l'exemple de nos données, représentées en CSV.

```
"Bouvier" ; "Nicolas"; "L'usage du monde" ; 1963
```

On comprend bien que le premier champ est le nom, le second le prénom, etc. Il paraît donc cohérent d'ajouter de nouvelles lignes comme:

```
"Bouvier"   ; "Nicolas"; "L'usage du monde" ; 1963
"Stevenson" ; "Robert-Louis"  ; "Voyage dans les Cévennes avec un âne" ; 1879
```

On a donné une structure régulière à nos informations, ce qui va permettre de les interroger et de les manipuler avec précision. On les stocke dans un fichier sur disque, et nous sommes donc en cours de constitution d'une véritable *base de données*. On peut en fait généraliser ce constat: *une base de données est toujours un ensemble de fichiers, stockés sur une mémoire externe comme un disque, dont le contenu obéit à certaines règles de structuration.*

Peut-on se satisfaire de cette solution et imaginer que nous pouvons construire des applications en nous appuyant directement sur des fichiers structurés, par exemple des fichiers CSV? C'est la méthode illustrée par la figure 1. Dans une telle situation, chaque utilisateur applique des programmes au fichier, pour en extraire des données, pour les modifier, pour les créer.

Cette approche n'est pas totalement inenvisageable, mais soulève en pratique de telles difficultés que *personne* (personne de censé en tout cas) n'a recours à une telle solution. Voici un petit catalogue de ces difficultés.

- *Lourdeur d'accès aux données.* En pratique, pour chaque accès, même le plus simple, il faudrait écrire un programme adapté à la structure du fichier. La production et la maintenance de tels programmes seraient extrêmement coûteuses.

- *Risques élevés pour l'intégrité et la sécurité.* Si tout programmeur peut accéder directement aux fichiers, il est impossible de garantir la sécurité et l'intégrité des données. Quelqu'un peut très bien par exemple, en toute bonne foi, faire une fausse manœuvre qui rend le fichier illisible.

- *Pas de contrôle de concurrence.* Dans un environnement où plusieurs utilisateurs accèdent aux même fichiers, comme illustré par exemple sur la Figure 1, des problèmes de concurrence d'accès se posent, notammment pour les mises à jour. Comment gérer par exemple la situation où deux utilisateurs souhaitent en même temps ajouter une ligne au fichier?

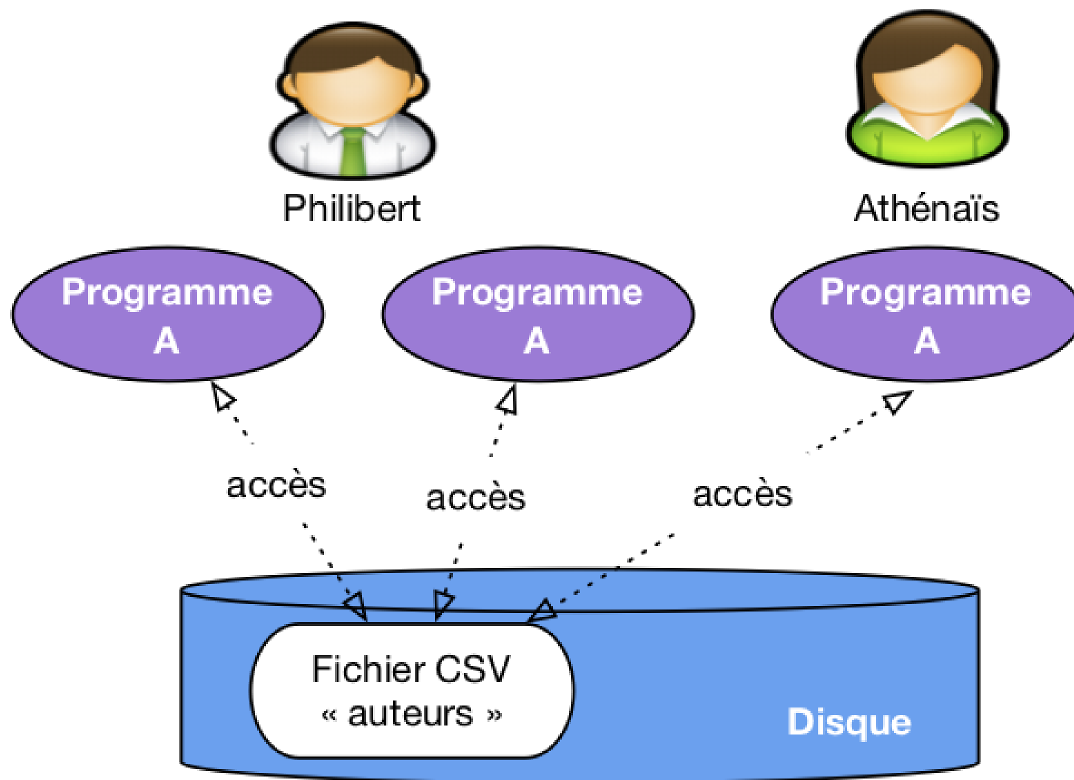


Figure 1: Une approche simpliste avec accès direct aux fichiers de la base

- *Performances.* Tant qu'un fichier ne contient que quelques centaines de lignes, on peut supposer que les performances ne posent pas de problème, mais que faire quand on atteint les Gigaoctets (1,000 Mégaoctets), ou même le Téraoctet (1,000 Gigaoctets)? Maintenir des performances acceptables suppose la mise en œuvre d'algorithmes ou de structures de données demandant des compétences très avancées, probablement hors de portée du développeur d'application qui a, de toute façon, mieux à faire.

  Chacun de ces problèmes soulève de redoutables difficultés techniques.

Leur combinaison nécessite la mise en place de systèmes d'une très grande complexité, capable d'offrir à la fois un accès simple, sécurisé, performant au contenu d'une base, et d'accomplir le tour de force de satisfaire de tels accès pour des dizaines, centaines ou même milliers d'utilisateurs simultanés, le tout en garantissant l'intégrité de la base même en cas de panne. De tels systèmes sont appelés *Systèmes de Gestion de Bases de Données*, SGBD en bref.

Définition (SGBD)
Un Système de Gestion de Bases de Données (SGBD) est un système informatique qui assure la gestion de l'ensemble des informations stockées dans une base de données. Il prend en charge, notamment, les deux grandes fonctionnalités suivantes:

1. Accès aux fichiers de la base, garantissant leur intégrité, contrôlant les opérations concurrentes, optimisant les recherches et mises à jour.

2. Interactions avec les applications et utilisateurs, grâce à des langages d'interrogation et de manipulation à haut niveau d'abstraction.

Avec un SGBD, les applications n'ont plus *jamais* accès directement aux fichiers, et ne savent d'ailleurs même pas qu'ils existent, quelle est leur structure et où ils sont situés. L'architecture

classique est celle illustrée par la figure 2. Le SGBD apparaît sous la forme d'un *serveur*, c'est-à-dire d'un processus informatique prêt à communiquer avec d'autres (les « clients ») via le réseau. Ce serveur est hébergé sur une machine (la « machine serveur ») et est le *seul* à pouvoir accéder aux fichiers contenant les données, ces fichiers étant le plus souvent stockés sur le disque de la machine serveur.

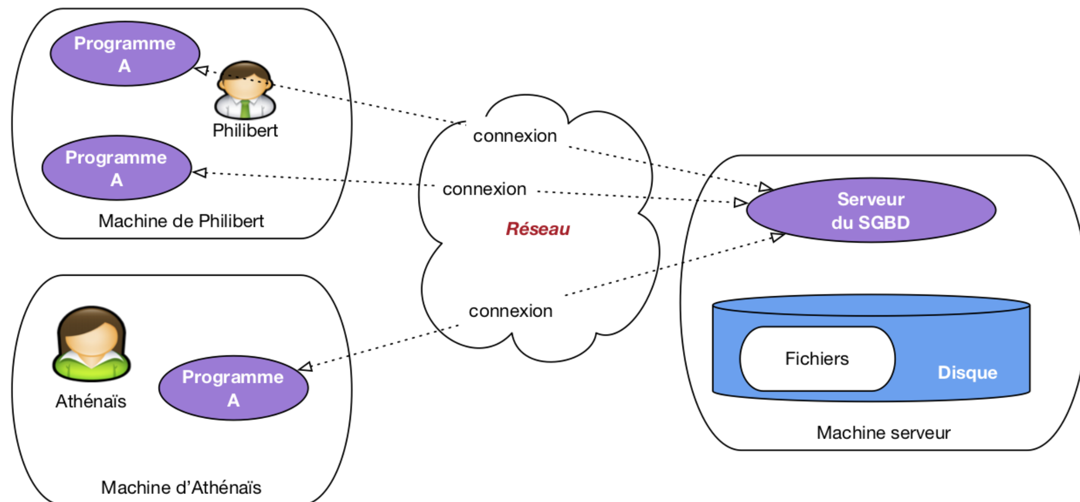


Figure 2: Architecture classique, avec serveur du SGBD

Les applications utilisateurs, maintenant, accèdent à la base *via* le programme serveur auquel elles sont connectés. Elles transmettent des commandes (d'où le nom « d'applications clientes ») que le serveur se charge d'appliquer. Ces applications bénéficient donc des puissants algorithmes implantés par le SGBD dans son serveur, comme par exemple la capacité à gérer les accès concurrents, où à satisfaire avec efficacité des recherches portant sur de très grosses bases.

Cette architecture est à peu près universellement adoptée par tous les SGDB de tous les temps et de toutes les catégories. Les notions suivantes, et le vocabulaire associé, sont donc très importantes à retenir.

Définition (architecture client serveur)

- **Programme serveur**. Un SGBD est instancié sur une machine sous la forme d'un *programme serveur* qui gère une ou plusieurs bases de données, chacune constituée de fichiers stockés sur disque. Le programme serveur est seul responsable de tous les accès à une base, et de l'utilisation des ressources (mémoire, disques) qui servent de support à ces accès.

- **Clients (programmes)**. Les *programmes (ou applications) clients* se connectent au programme serveur via le réseau, lui transmettent des *requêtes* et recoivent des données en retour. Ils ne disposent d'aucune information directe sur la base.

Malgré que ceci soit le cas dans la majorité des cas d'usage, certaines utilisations de base de données ne nécessite pas de gérer plusieurs clients en parallèle. Pour ces utilisations, des SGDBs légers existent où le client unique et serveur sont fusionnés dans le même programme. C'est notamment le cas de SQLite, que nous utiliserons en TP.

1.4.2 Modèle et couches d'abstraction

Le fait que le serveur de données s'interpose entre les fichiers et les programmes clients a une conséquence extrêmement importante: ces clients, n'ayant pas accès aux fichiers, ne voient les données que sous la forme que veut bien leur présenter le serveur. Ce dernier peut donc choisir

le mode de représentation qui lui semble le plus approprié, la seule condition étant de pouvoir aisément convertir le format des fichiers vers la représentation « publique ».

En d'autres termes, on peut s'abstraire de la complexité et de la lourdeur des formats de fichiers avec tous leurs détails compliqués de codages, de gestion de la mémoire, d'adressage, et proposer une représentation simple et intuitive aux applications. Une des propriétés les plus importantes des SGBD est donc la distinction entre plusieurs *niveaux d'abstraction* pour la réprésentation des données. Il nous suffira ici de distinguer deux niveaux: le niveau logique et le niveau physique.

Définition: Niveau physique, niveau logique

- Le *niveau physique* est celui du codage des données dans des fichiers stockés sur disque.
- Le *niveau logique* est celui de la représentation les données dans des structures abstraites, proposées aux applications clientes, obtenues par conversion du niveau physique.

Les structures du niveau logique définissent une *modélisation* des données: on peut envisager par exemple des structures de graphe, d'arbre, de listes, etc. Le *modèle relationnel* se caractérise par une modélisation basée sur une seule structure, la table. Cela apporte au modèle une grande simplicité puisque toutes les données ont la même forme et obéissent aux même contraintes. Cela a également quelques inconvénients en limitant la complexité des données représentables. Pour la grande majorité des applications, le modèle relationnel a largement fait la preuve de sa robustesse et de sa capacité d'adaptation. C'est lui que nous étudions dans l'ensemble du cours.

La Fig. 3 illustre les niveaux d'abstraction dans l'architecture d'un système de gestion de données. Les programmes clients ne voient que le niveau logique, c'est-à-dire des tables si le modèle de données est relationnel (il en existe d'autres, nous ne les étudions pas ici). Le serveur est en charge du niveau physique, de la conversion des données vers le niveau logique, et de toute la machinerie qui permet de faire fonctionner le système: mémoire, disques, algorithmes et structures de données. Tout cela est, encore une fois, invisible (et c'est tant mieux) pour les programmes clients qui peuvent se concentrer sur l'accès à des données présentées le plus simplement possible.

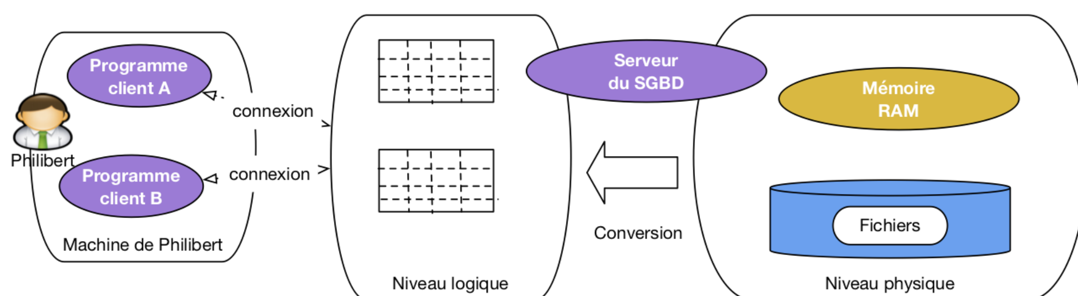


Figure 3: Illustration des niveaux logique et physique

Signalons pour finir cette courte présentation que les niveaux sont en grande partie indépendants, dans le sens où l'on peut modifier complètement l'organisation du niveau physique sans avoir besoin de changer qui que ce soit aux applications qui accèdent à la base. Cette *indépendance logique-physique* est très précieuse pour l'administration des bases de données.

1.4.3 Les langages

Un modèle, ce n'est pas seulement une ou plusieurs structures pour représenter l'information indépendamment de son format de stockage, c'est aussi un ou plusieurs langages pour interroger et, plus généralement, interagir avec les données (insérer, modifier, détruire, déplacer, protéger, etc.). Le langage permet de construire les commandes transmises au serveur.

Le modèle relationnel s'est construit sur des bases formelles (mathématiques) rigoureuses, ce qui explique en grande partie sa robustesse et sa stabilité depuis l'essentiel des travaux qui l'ont élaboré, dans les années 70-80. Deux langages d'interrogation, à la fois différents, complémentaires et équivalents, ont alors été définis:

1. Un langage *déclaratif*, basé sur la logique mathématique.
2. Un langage *procédural*, et plus précisément *algébrique*, basé sur la théorie des ensembles.

Un langage est *déclaratif* quand il permet de spécifier le résultat que l'on veut obtenir, sans se soucier des opérations nécessaires pour obtenir ce résultat. Un langage algébrique, au contraire, consiste en un ensemble d'opérations permettant de transformer une ou plusieurs tables en entrée en une table - le résultat - en sortie.

Ces deux approches sont très différentes. Elles sont cependant parfaitement complémentaires. l'approche déclarative permet de se concentrer sur le raisonnement, l'expression de requêtes, et fournit une définition rigoureuse de leur signification. L'approche algébrique nous donne une boîte à outil pour calculer les résultats.

Le langage SQL, assemblant les deux approches, a été normalisé sur ces bases. Il est utilisé depuis les années 1970 dans tous les systèmes relationnels, et il paraît tellement naturel et intuitif que même des systèmes construits sur une approche non relationnelle tendent à reprendre ses constructions.

Le terme SQL désigne plus qu'un langage d'interrogation, même s'il s'agit de son principal aspect. La norme couvre également les mises à jour, la définition des tables, les contraintes portant sur les données, les droits d'accès. SQL est donc le langage à connaître pour interagir avec un système relationnel.

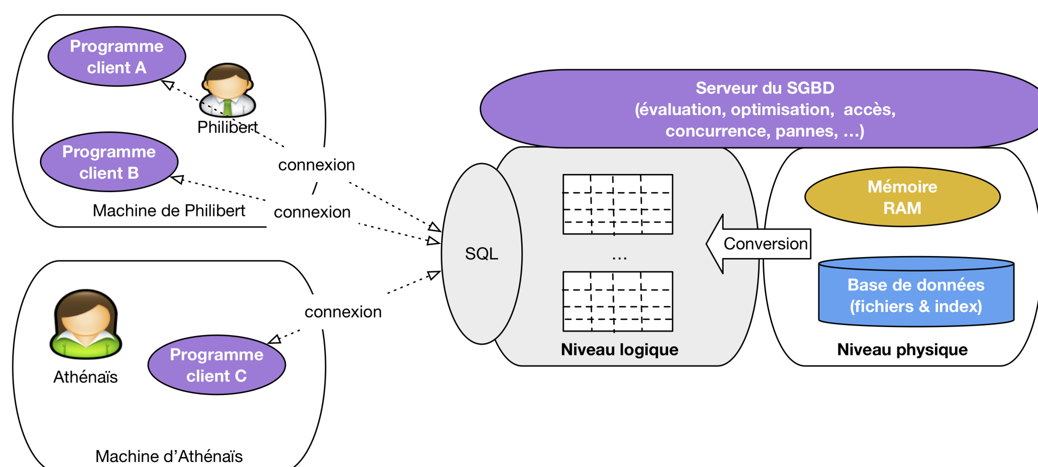


Figure 4: L'interface « modèle / langage » d'un système relationnel

La Fig. 4 étend le schéma précédent en introduisant SQL, qui apparaît comme le constituant central pour établir une communication entre une application et un système relationnel. Les parties grisées de cette figure sont celles couvertes par le cours. Nous allons donc étudier le modèle relationnel (représentation des données sous forme de table), le langage d'interrogation

SQL sous ses deux formes, déclarative et algébrique, et l'interaction avec ce langage *via* un langage de programmation permettant de développer des applications.

Tout cela consitue à peu près tout ce qu'il est nécessaire de connaître pour concevoir, implanter, alimenter et interroger une base de données relationnelle, que ce soit directement ou par l'intermédiaire d'un langage de programmation.

2 Le modèle relationnel

Qu'est-ce donc que ce fameux « modèle relationnel » ? En bref, c'est un ensemble de résultats scientifiques, qui ont en commun de s'appuyer sur une représentation tabulaire des données. Beaucoup de ces résultats ont débouché sur des mises en œuvre pratique. Ils concernent essentiellement deux problématiques complémentaires:

- *La structuration des données.* Comme vous le verrez sûrement dans le cours de L2 "système d'information", on ne peut pas se contenter de placer toute une base de données dans une seule table, sous peine de rencontrer rapidement des problèmes insurmontables. Une base de données relationnelle, c'est un ensemble de tables associées les unes aux autres. La conception du schéma (structures des tables, contraintes sur leur contenu, liens entre tables) doit obéir à certaines règles et satisfaire certaines proprietés. Une théorie solide, la *normalisation* a été développée qui permet de s'assurer que l'on a construit un schéma correct.
- *Les langages d'interrogation.* Le langage SQL que nous connaissons maintenant est issu d'efforts intenses de recherche menés dans les années 70-80. Deux approches se sont dégagées: la principale est une conception *déclarative* des langages de requêtes, basées sur la logique mathématique. Avec cette approche on formule (c'est le mot) ce que l'on souhaite, et le système décide comment calculer le résultat. La seconde est de nature plus procédurale, et identifie l'ensemble minimal des opérateurs dont le système doit disposer pour évaluer une requête. C'est cette seconde approche qui est utilisée en interne pour construire des programmes d'évaluation.

Dans ce chapitre nous introduisons le modèle relationnel, soit essentiellement la représentation des données. Deux exemples de bases, commentés, sont donnés en fin de chapitre. Les chapitres suivants seront consacrés aux différents aspects du langage SQL.

Supports complémentaires (couvrant plus de sujet que ce cours):

- Diapositives: modèle relationnel
- Vidéo sur le modèle relationnel

L'expression « modèle relationnel » a pour origine (surprise!) la notion de relation, un des fondements mathématiques sur lesquels s'appuie la théorie relationnelle. Dans le modèle relationnel, la seule structure acceptée pour représenter les données est la relation.

2.1 Qu'est-ce qu'une relation

Une *relation* est un objet mathématique pour représenter une table à deux dimensions. La table 1 représente la relation `Département`, qui contient deux colonnes : on dit qu'elle est binaire. On introduit par la suite les deux composantes d'une relation : son *schéma* et son *instance*.

2.2 Le schéma

Le *schéma* d'une relation décrire sa forme et permet d'interpréter les données qu'elle contient.

Table 1: Représentation sous forme de table de la relation `Département`

nom	code
Ardèche	7
Gard	30
Manche	50
Paris	75

Definition (Schéma):
Le schéma d'une relation est défini par :

1. Le nom de la relation.
2. Un nom distinct pour chaque colonne, dit *nom d'attribut*, noté A_i.
3. Le domaine de valeur (type) de chaque colonne, noté D_i.

Dans le monde relationnel, les possibles domaines de valeurs sont les entiers I, les réels (ou plus précisément les nombres en virgule flottante puisqu'on ne sait pas représenter une précision infinie) F, les chaînes de caractères S, les dates, etc. Ce sont des domaines de valeurs *élémentaires* en opposition à ceux de valeur *structurée*: il n'est pas possible en relationnel de placer dans une cellule un graphe, une liste, un enregistrement.

Ce schéma pour la relation nommée R s'écrit de manière concise $R(A_1 : D_1, A_2 : D_2, \cdots, A_n : D_n)$. Tous les A_i sont distincts, mais on peut bien entendu utiliser plusieurs fois le même type.

L'*arité* d'une relation est son nombre d'attribut. La relation R a pour arité n. On dit notamment que la relation est unaire pour $n = 1$, binaire pour $n = 2$ et ternaire pour $n = 3$.

Le schéma de notre table des départements est donc `Département(nom : string, code : integer)`. Le domaine de valeur ayant relativement peu d'importance, on pourra souvent l'omettre et écrire le schéma `Département(nom, code)`.

2.3 L'instance d'une relation

L'*instance* d'une relation représente son contenu. Elle est représenté par un ensemble de *nuplets*, chacun représentant une ligne dans la représentation par table. Par exemple, l'instance relation `Département`, représentée par la table 1, contient les quatre nuplets suivants :

- $(\text{Ardèche}, 7)$
- $(\text{Gard}, 30)$
- $(\text{Manche}, 50)$
- $(\text{Paris}, 75)$

On peut voir chaque nuplet comme une liste de valeurs élémentaires, où chaque position est associée à un attribut de la relation, qui les contient. Ici, les nuplets contiennent deux positions, la première est associée à l'attribut nommé `nom` et la seconde à celui nommé `code`. On dit que la valeur pour l'attribut `code` pour le nuplet $(\text{Ardèche}, 7)$ est l'entier 7.

On introduit une restriction sur les nuplets d'une instance pour qu'ils respectent le schéma : l'instance d'une relation de schéma $R(A_1 : D_1, A_2 : D_2, \cdots, A_n : D_n)$ contient des nuplets $(a_1, a_2, \cdots, a_n)$, où chaque a_i est une valeur du domaine D_i. Formellement, on obtient la définition suivante.

Définition (instance):
L'*instance* d'une relation de schéma $R(A_1 : D_1, A_2 : D_2, \cdots, A_n : D_n)$ est un sous-ensemble fini du produit cartésien $D_1 \times D_2 \times \cdots \times D_n$.

Au cas où vous ne seriez pas familiers avec cette notion, le *produit cartésien* entre deux ensembles $A \times B$ est l'ensemble de toutes les paires possibles constituées d'un élément de A et d'un élément de B. Cette définition peut se généraliser avec n ensembles (voir Wikipédia pour plus de détails). C'est une notion qui reviendra régulièrement dans ce cours.

On a ajouté dans cette définition une nouvelle restriction, à savoir que l'instance d'une relation est un ensemble *fini*, vu qu'on ne peut pas représenter un ensemble infini avec une machine.

L'ensemble des paires constituées de tous les noms des département français et de leur numéro de code pourrait être une instance de la relation `Département`: c'est un ensemble fini, sous-ensemble du produit cartésien $S \times I$. On a considéré uniquement quatre départements dans l'exemple précédent, ce qui est aussi un choix d'instance valable.

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

- *L'ordre des nuplets est indifférent* car il n'y a pas d'ordre dans un ensemble; conséquence pratique: le résultat d'une requête appliquée à une relation ne dépend pas de l'ordre des lignes dans la relation.
- *On ne peut pas trouver deux fois le même nuplet* car il n'y a pas de doublons dans un ensemble.
- Il n'y a pas (en théorie) de « cellule vide » dans la relation; toutes les valeurs de tous les attributs de chaque nuplet sont toujours connues.

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

2.4 Représentation des relations

Une relation est un objet abstrait, on peut la représenter de différentes manières. Une représentation naturelle pour les relation binaire est le graphe comme le montre la figure 5. La représentation sous forme de table s'avère beaucoup plus pratique quand la relation n'est plus binaire mais ternaire et au-delà. C'est pour cette raison que nous privilégierons cette dernière dans la suite du cours.

Et en ce qui concerne le vocabulaire, le tableau suivant montre celui, rigoureux, issu de la modélisation mathématique et celui, plus vague, correspondant à la représentation par table. Les termes de chaque ligne seront considérés comme équivalents, mais on privilégiera les premiers qui sont plus précis.

Terme du modèle	Terme de la représentation par table
Relation	Table
nuplet	ligne
Nom d'attribut	Nom de colonne
Valeur d'attribut	Cellule
Domaine	Type

Attention à utiliser ce vocabulaire soigneusement, sous peine de confusion. Ne pas confondre par exemple le nom d'attribut (qui est commun à toute la table) et la valeur d'attribut (qui est spécifique à un nuplet).

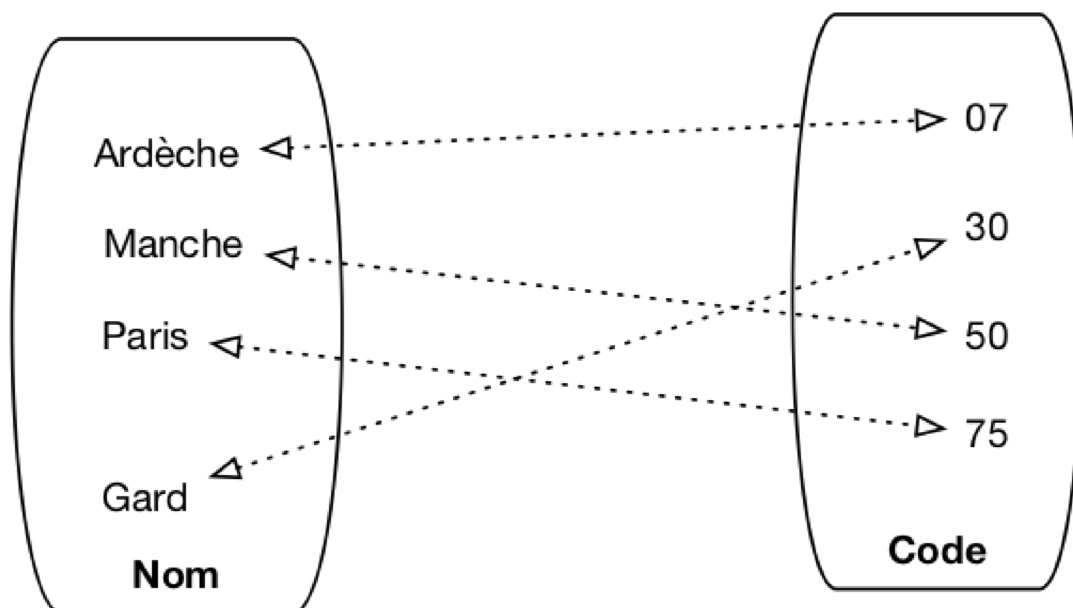


Figure 5: Une relation binaire représentée comme un graphe

La structure utilisée pour représenter les données est donc extrêmement simple. Il faut insister sur le fait que les valeurs des attributs, celles que l'on trouve dans chaque cellule de la table, sont élémentaires: entiers, chaînes de caractères, etc. On *ne peut pas* avoir une valeur d'attribut qui soit un tant soit peu construite, comme par exemple une liste, ou une sous-relation. Les valeurs dans une base de données sont dites *atomiques* (pour signifier qu'elles sont non-décomposables, rien de toxique à priori). Cette contrainte conditionne tous les autres aspects du modèle relationnel, et notamment la conception, et l'interrogation.

2.5 Mais que représente une relation ?

En première approche, une (instance de) relation est simplement un ensemble de nuplets. On peut donc lui appliquer des opérations ensemblistes: intersection, union, produit cartésien, projection, etc. Cette vision se soucie peu de la signification de ce qui est représenté, et peut mener à des manipulations dont la finalité reste obscure. Ce n'est pas forcément le meilleur choix pour un utilisateur humain, mais ça l'est pour un système qui ne se soucie que de la description opérationnelle.

Dans une seconde approche, plus « sémantique », une relation est un mécanisme permettant d'énoncer des faits sur le monde réel. Chaque nuplet correspond à un tel énoncé. Si un nuplet est présent dans la relation, le fait est considéré comme vrai, sinon il est faux.

La table des départements sera ainsi interprétée comme un ensemble d'énoncés: « Le département de l'Ardèche a pour code 07 », « Le département du Gard a pour code 30 », et ainsi de suite. Si un nuplet, par exemple, $(\text{Gers}, 32)$, n'est pas dans la base, on considère que l'énoncé « Le département du Gers a pour code 32 » est faux.

Cette approche mène directement à une manipulation des données fondée sur des raisonnements s'appuyant sur les valeurs de vérité énoncées par les faits de la base. On a alors recours à la logique formelle pour exprimer ces raisonnements de manière rigoureuse. Dans cette approche, qui est à la base de SQL, interroger une base, c'est déduire un ensemble de faits qui satisfont un énoncé logique (une « formule »). Selon ce point de vue, SQL est un langage pour écrire des formules logiques, et un système relationnel est (entre autres) une machine qui effectue des démonstrations.

3 SQL, langage déclaratif

Il est courant en informatique de disposer de plusieurs langages pour résoudre un même problème. Ces langages ont leur propre syntaxe, mais surtout ils peuvent s'appuyer sur des approches de programmation très différentes. Vous avez peut-être rencontré des langages impératifs (le C), orientés-objet (Java, Python) ou fonctionnels (Camel, Erlang).

Certains langages sont plus appropriés à certaines tâches que d'autres. Il est plus facile de vérifier les propriétés d'un programme écrit en langage fonctionnel par exemple que d'un programme C. Si l'on s'en tient aux bases de données (et particulièrement pour les bases relationnelles), deux approches sont possibles: la première est *déclarative* et la seconde *procédurale*.

L'approche procédurale est assez familière: on dispose d'un ensemble d'opérations, et on décrit le calcul à effectuer par une séquence de ces opérations. Chaque opération élémentaire peut être très simple, mais la séquence à construire pour régler des problèmes complexes peut être longue et peu claire.

L'approche déclarative est beaucoup plus simple conceptuellement: elle consiste à décrire les propriétés du point d'arrivée (le résultat) en fonction de celles du point de départ (les données de la base, dans notre cas). La description de ces propriétés se fait classiquement par des formules logiques qui indiquent comment l'existence d'un fait f_1 au départ implique l'existence d'un fait f_2 à l'arrivée.

Cela peut paraître abstrait, et de fait ça l'est puisqu'aucun calcul n'est spécifié. On s'appuie simplement sur le fait que l'informatique *sait* effectuer des calculs spécifiés par des formules logiques (dans le cas particulier des bases de données en tout cas) apparemment indépendantes de tout processus calculatoire. Il se trouve que SQL *est* un langage déclaratif, et qu'il l'était même exclusivement dans sa version initiale.

Note

Il existe de très bonnes raisons pour privilégier le caractère déclaratif des langages de requêtes, liées à l'indépendance entre le niveau logique et le niveau physique dont nous avons déjà parlé, et à l'opportunité que cette indépendance laisse au SGBD pour déterminer la meilleure manière d'évaluer une requête. Cela n'est possible que si l'expression de cette dernière est assez abstraite pour n'imposer aucun choix de calcul à priori.

Avec SQL, on ne dit rien sur la manière dont le résultat doit être calculé: c'est le problème du SGBD, qui sait d'ailleurs trouver la solution bien mieux que nous puisqu'on ne connaît pas l'organisation des données. On se contente avec SQL d'énoncer les propriétés de la relation de sortie en fonction des propriétés de la base en entrée. Pour bien utiliser SQL, il faut bien comprendre la *signification* de ce que l'on exprime, ce qui est rigoureusement défini par une formulation logique basée sur le calcul des prédicats. Vous aurez l'occasion de voir en détail cette formulation plus tard dans vos études, ce cours se concentre sur la formulation des requêtes en SQL.

On rencontre parfois l'argument que SQL est, à l'inverse d'un langage de programmation, accessible à un non-initié, car il est proche de la manière dont on exprimerait naturellement une recherche. Ce n'est vrai que si on sait *formuler* cette dernière de manière rigoureuse, et c'est exactement ce que nous allons apprendre dans ce chapitre.

3.0.1 SQL est-il *totalement* déclaratif ?

Au fil des années et des normes successives, SQL s'est étendu pour incorporer un autre langage relationnel, l'algèbre, que nous étudierons dans le prochain chapitre. Est-ce à dire que la forme déclarative n'était pas suffisante? Non: tous ces ajouts sont redondants et auraient pu être omis sans affecter l'expressivité du langage.

On se retrouve à l'heure actuelle avec un langage très riche dans lequel on peut exprimer des requêtes de manière soit déclarative, soit procédurale, soit par un mélange des deux. Cela ne

contribue pas forcément à la facilité d'apprentissage, et introduit une certaine confusion sur la portée de telle ou telle formulation, et sa possible équivalence avec une autre.

En présentant successivement les deux approches, et en montrant ensuite comment elles sont parfaitement équivalentes l'une à l'autre, ce cours a choisi de tenter de clarifier la situation.

3.1 S1: Un peu de logique

La logique est l'art de raisonner, autrement dit de construire des argumentations rigoureuses permettant d'induire ou déduire de nouveaux faits à partir de faits existants (ou considérés comme tels). La logique mathématique est la partie de la logique qui présente les règles de raisonnement de manière formelle. C'est une branche importante des mathématiques, qui s'est fortement développée au début du XXe siècle, et constitue un fondement majeur de la science informatique.

Commençons par donner l'intuition de la logique sous-jacente à SQL pour formuler et interpréter les requêtes. Pour se familiariser à la logique, il faut recourir à des textes spécialisés. Pour une passionante introduction historico-scientifique, je vous recommande d'ailleurs la bande dessinée (mais oui) *Logicomix*, parue chez Vuivert en 2009.

Dans les bases de données, on souhaite représenter des énoncés complexes, qui peuvent avoir une signification logique. On peut souhaiter représenter les énoncés « Mozart a composé Don Giovanni », « Mozart a composé Cosi fan tutte », et « Bach a composé la Messe en si » en mettant en avant qu'ils déclarent le même type de propriété (le fait de composer une œuvre) liant des entités (Mozart, Bach, leurs œuvres).

Pour énoncer une propriété, on utilise une relation pour lier des entités. On peut ici définir une relation `Compose(compositeur, oeuvre)` dans laquelle chaque nuplet (X, Y) énonce "X a composé Y". Stockées dans une relation, les trois propositions suivantes deviennent :

compositeur	oeuvre
Mozart	Don Giovanni
Mozart	Cosi fan tutte
Bach	Messe en si

Il existe virtuellement une infinité de nuplets énoncables avec une relation. Certains sont faux, d'autres vrais.

Quand on modélise le monde réel, les nuplets vrais doivent être énoncés explicitement comme, dans l'exemple ci-dessus, les compositeurs et leurs œuvres. Une base de données n'est rien d'autre que l'ensemble des nuplets considérés comme vrais pour des relations applicatives, tous les autres étant considérés comme faux. Un système pourra nous dire que le $(\text{Bach}, \text{Don Giovanni})$ suivant est faux (il n'est pas dans la base), alors que le nuplet $(\text{Mozart}, \text{Don Giovanni})$ est vrai (il appartient à la base).

De tels nuplets forment des requêtes très simples dont est la réponse Vrai ou Faux. Nous restons pour l'instant dans un système assez restreint où tous les nuplets font référence à des entités connues, dit autrement, tous les attributs de la relation ont une valeur fixée. De tels nuplets sont dits *fermés*.

Mais on peut également manipuler des nuplets dits *ouverts* dans lesquels certaines valeurs d'attributs sont inconnus. dans lesquels certains objets sont inconnus, et remplacés par des variables habituellement dénotés x, y, z, etc. On obtient un langage beaucoup plus puissant. Dans le nuplet ouvert suivant, le nom du compositeur est remplacé par une variable.

$$(x, \text{Don Giovanni})$$

Intuitivement, ce nuplet ouvert représente concisément tous les nuplets fermés exprimant qu'un musicien x a composé une œuvre intitulée Don Giovanni. En affectant à x toutes les valeurs possibles (une variable est supposée couvrir un domaine de valeurs), on énumère tous

les nuplets de ce type. La plupart sont faux (ceux qui ne sont pas dans la base), certains sont vrais.

Interroger une base relationnelle, c'est simplement demander au système les valeurs de x pour lesquelles $(x, \text{Don Giovanni})$ est vrai. La réponse est `Mozart`, dans notre cas.

Voici la requête SQL correspondante, on y précise que l'on souhaite obtenir les valeurs de l'attribut `compositeur` dans les nuplets de la relation `Compose` dont la valeur de l'attribut `oeuvre` est Don Giovanni :

```
select compositeur
from Compose
where oeuvre='Don Giovanni'
```

On peut étendre l'expressivité de ce langage de requêtes à l'aide de connecteurs logiques, aussi présent en SQL. Notamment les connecteurs booléens "et", "ou" et la négation dont la signification est commune avec celle des autres langages de programmation. La requête SQL suivante demande l'ensemble des oeuvres composées soit par Mozart ou par Bach. Cette requête retourne trois réponses sur l'instance précédente.

```
select oeuvre
from Compose
where compositeur='Mozart' or compositeur='Bach'
```

Un autre exemple, où on demande les compositeurs issus des nuplets différents de $(\text{Mozart}, \text{Don Giovanni})$.

```
select compositeur
from Compose
where not (compositeur='Mozart' and oeuvre='Don Giovanni')
```

3.2 S2: SQL conjonctif

Supports complémentaires:

- Diapositives: SQL conjonctif
- Vidéo sur la première partie de SQL

Cette session présente le langage SQL dans sa version déclarative. La base de données est constituée d'un ensemble de relations. Ces relations contiennent des nuplets (fermés, sans variable).

Pour illustrer les requêtes et leur interprétation, nous prenons la base des voyageurs. Vous pouvez expérimenter toutes les requêtes présentées (et d'autres) directement sur note site `http://deptfod.cnam.fr/bd/tp`. Voir également l'atelier SQL proposé en fin de chapitre.

Cette session se limite à la partie dite « conjonctive » de SQL, celle où toutes les requêtes peuvent s'exprimer sans négation. La prochaine session complètera le langage.

3.2.1 La base des voyageurs

Cette base de données décrit les pérégrinations de quelques voyageurs plus ou moins célèbres. Ces voyageurs occupent occasionnellement des logements pendant des périodes plus ou moins longues, et y exercent (ou pas) quelques activités.

Voici le schéma de la base.

- Voyageur (idVoyageur, nom, prénom, ville, région)
- Séjour (idSéjour, idVoyageur, codeLogement, début, fin)
- Logement (code, nom, capacité, type, lieu)

- Activité (codeLogement, codeActivité, description)

La table des voyageurs

Dans la table `Voyageur`, les voyageurs sont identifiés par un numéro séquentiel nommé `idVoyageur`, incrémenté de 10 en 10. On indique la ville et la région de résidence.

idVoyageur	nom	prénom	ville	région
10	Fogg	Phileas	Ajaccio	Corse
20	Bouvier	Nicolas	Aurillac	Auvergne
30	David-Néel	Alexandra	Lhassa	Tibet
40	Stevenson	Robert Louis	Vannes	Bretagne

La table Logement

La table `Logement` est également très simple, voici son contenu.

code	nom	capacité	type	lieu
pi	U Pinzutu	10	Gîte	Corse
ta	Tabriz	34	Hôtel	Bretagne
ca	Causses	45	Auberge	Cévennes
ge	Génépi	134	Hôtel	Alpes

L'information nommée `région` dans la table des voyageurs s'appelle maintenant `lieu` dans la table `Logement`. Ce n'est pas tout à fait cohérent, mais corrrespond à des situations couramment rencontrées où la même information apparaît sous des noms différents.

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 (on fait simple, ce n'est pas une base pour de vrai).

idSéjour	idVoyageur	codeLogement	début	fin
1	10	pi	20	20
2	20	ta	21	22
3	30	ge	2	3
4	20	pi	19	23
5	20	ge	22	24
6	10	pi	10	12
7	30	ca	13	18
8	20	ca	21	22

`Séjour` référence le logement et le voyageur par leur identifiant. On peut voir que la valeur de `idVoyageur` (ou `codeLogement`) dans cette relation est *toujours* la valeur de l'un des identifiants de `Voyageur` (respectivement `Logement`).

Connaissant un séjour, je connais donc les identifiants du logement et du voyageur, et je peux trouver la description complète de ces derniers dans leur table respective.

La table Activité

Cette table contient les activités associées aux logements.

codeLogement	codeActivité	description
pi	Voile	Pratique du dériveur et du catamaran
pi	Plongée	Baptèmes et préparation des brevets
ca	Randonnée	Sorties d'une journée en groupe
ge	Ski	Sur piste uniquement
ge	Piscine	Nage loisir non encadrée

3.2.2 Requête mono-variable

Dans les requêtes relationnelles, les variables ne désignent pas des valeurs individuelles, mais des nuplets libres. Une variable-nuplet t a donc des composants $a_1, a_2, \dots a_n$ que l'on désigne par $t.a_1, t.a_2, \cdots, t.a_n$. Par souci de simplicité, on nomme souvent les variables comme les attributs du schéma, mais ce n'est pas une obligation.

Commençons par étudier les requêtes utilisant une seule variable. Leur forme générale est

```
select [distinct] t.a1, t.a2, ..., t.an
from T as t
where <condition>
```

Ce « bloc » SQL comprend trois clauses: le `from` définit la variable libre et ce que nous appellerons la *portée* de cette variable, le `where` exprime les conditions sur la variable libre, enfin le `select`, accompagné du mot-clé optionnel `distinct`, construit le nuplet constituant le résultat.

L'interprétation est la suivante: je veux constituer tous les nuplets *fermés* $(t.a_1, t.a_2, \cdots, t.a_n)$ dont les valeurs satisfont les deux points suivants:

- La première, la variable t est un nuplet de la relation T. Nous appelons donc cette partie la *portée*.
- La seconde, les conditions sur t définies le `where` , c'est à dire une formule logique sur t, que nous appellons la *condition*.

Important

La portée définit les variables libres de la formule, celles pour lesquelles on va chercher l'affectation qui satisfait la condition, et à partir desquelles on va construire le nuplet-résultat.

À propos du `distinct` Une relation ne contient pas de doublon. La présence de doublons (deux unités d'information indistinguables l'une de l'autre) dans un système d'information est une anomalie. Pour prendre quelques exemples applicatifs, on ne veut pas envoyer deux fois le même message, on ne veut pas produire deux fois la même facture, on ne veut pas afficher deux fois le même document, etc. Vous pouvez vérifier que votre moteur de recherche préféré applique ce principe.

Les relations de la base sont sans doublons. Qu'en est-il des relations calculées, autrement dit le résultat des requêtes ? Supposons que l'on souhaite connaître tous les types de logements. Voici la requête SQL sans `distinct`:

```
select type
from Logement
```

On obtient une relation avec deux nuplets identiques.

type
Gîte
Hôtel
Auberge
Hôtel

Sans `distinct`, SQL peut produire des relations avec doublons. Du point de vue logique, cela montre simplement que l'on a établi le même fait de deux manières différentes, mais cela ne sert à rien d'afficher ce fait deux fois (ou plus). Si on ajoute `distinct`

```
select distinct type
from Logement
```

on obtient

type
Gîte
Hôtel
Auberge

Pourquoi SQL n'élimine-t-il pas systématiquement les doublons? En premier lieu parce que cette élimination implique un algorithme potentiellement coûteux si la relation en entrée est très grande. Il faut en effet effectuer un tri suivi d'une élimination des nuplets identiques. Sur des petites relations, la différence en temps d'exécution est indiscernable, mais elle peut devenir significative quand on a des centaines de milliers de nuplets ou plus. Les concepteurs du langage SQL ont fait le choix, par défaut, d'éviter d'appliquer cet algorithme, ce qui revient à accepter de produire éventuellement des doublons.

Une seconde raison pour ne pas appliquer systématiquement l'algorithme d'élimination de doublons est que certaines requêtes, par construction, produisent un résultat sans doublons. Voici un exemple très simple

```
select code, type
from Logement
```

Inutile dans ces cas-là d'utiliser `distinct`. En d'autres termes: SQL nous laisse la charge de décider quand une requête risque de produire des doublons, et si nous souhaitons les éliminer. *Dans tout ce cours nous utilisons* `distinct` chaque fois que c'est nécessaire pour toujours obtenir un résultat sans doublon.

Il est par ailleurs très utile, quand on exprime une requête, de réfléchir à la possibilité qu'elle produise ou non des doublons et donc à la nécessité d'utiliser `distinct`. Si une requête produit potentiellement des doublons, il est sans doute pertinent de se demander quel est le sens du résultat obtenu.

Exemples Voici une première requête concrète sur notre base. On veut le nom et le type des logements corses.

```
select t.code, t.nom, t.type
from Logement as t
where t.lieu = 'Corse'
```

Note

Pour distinguer les chaînes de caractères des noms d'attribut, on les encadre par des apostrophes simples.

Note

SQL permet, quand c'est possible, quelques légères simplifications syntaxiques. La forme simplfiée de la requête précédente est donnée ci-dessous.

```
select code, nom, type
from Logement
where lieu = 'Corse'
```

On peut donc omettre de spécifier le nom de la variable quand il n'y a pas d'ambiguité, notamment l'interprétation du nom des champs.

Elle s'interprète de la manière suivante: on cherche les affectations d'une variable t parmi les nuplets de la relation `Logement`, telle que `t.lieu` ait pour valeur « Corse ».

De cette interprétation, assez évidente pour l'instant, il faut retenir qu'une table mentionnée dans le `from` de SQL définit en fait une variable dont la portée est la table (ici, `Logement`). Parmi toutes les affectations possibles de cette variable (c'est à dire les nuplets de `Logement`), on ne conserve que celles qui satisfont la condition exprimée par le reste de la formule.

Le système d'évaluation peut donc considérer que t est affectée à n'importe lequel des nuplets de la table, et évaluer si cette affectation satisfait la condition. Dans la table ci-dessous, la croix indique à quel nuplet t est affectée. Ici, la condition n'est clairement pas satisfaite.

t	code	nom	capacité	type	lieu
	pi	U Pinzutu	10	Gîte	Corse
	ta	Tabriz	34	Hôtel	Bretagne
X	ca	Causses	45	Auberge	Cévennes
	ge	Génépi	134	Hôtel	Alpes

En revanche, quand l'affectation est faite comme indiquée ci-dessous, la condition est satisfaite et sert à construire le nuplet-résultat.

t	code	nom	capacité	type	lieu
X	pi	U Pinzutu	10	Gîte	Corse
	ta	Tabriz	34	Hôtel	Bretagne
	ca	Causses	45	Auberge	Cévennes
	ge	Génépi	134	Hôtel	Alpes

Voici quelques exemples. Cherchons d'abord quels hôtels sont dans les Alpes. La requête SQL est:

```
select t.code, t.nom
from Logement as t
where t.type = 'Hôtel' and t.lieu = 'Alpes'
```

La condition à satisfaire pour un nuplet de la relation `Logement` est $t.type$ = 'Hôtel' and $t.lieu$ = 'Alpes'. C'est seulement le cas pour le dernier nuplet. Cherchons maintenant les hôtels qui, soit sont en Bretagne, soit ont au moins 100 chambres. La version SQL:

```
select t.code, t.nom
from Logement as t
where t.type = 'Hôtel' and (t.lieu = 'Bretagne' or t.capacité >= 100)
```

3.2.3 Requêtes multi-variables et jointures

Voyons maintenant le cas général où on s'autorise à utiliser plusieurs variables. Pour simplifier la notation, nous allons étudier les requêtes avec exactement deux variables. Il est facile ensuite de généraliser. Voici la forme d'une telle requête SQL.

```
select [distinct] t1.a1, ..., t1.an, t2.b1, ..., t2.bm
from T1 as t1, T2 as t2
where <condition>
```

L'interprétation est exactement la même que pour les requêtes mono-variables, légèrement généralisée: *parmi toutes les affectations possibles des variables t_1 et t_2, on ne conserve que celles qui satisfont la condition exprimée par le reste de la formule.*

Il n'y a rien de plus à comprendre. Il suffit de considérer toutes les affectations possibles de t_1 et t_2 et de ne garder que celles pour lesquelles la formule de condition est satisfaite.

Voici quelques exemples. On veut les noms des logements où on peut pratiquer le ski. Nous avons besoin de deux variables:

- la première s'affecte aux nuplets de la table `Activité`; on ne veut que ceux dont le code est `Ski`.
- la seconde s'affecte aux nuplets de la table `Logement`

Enfin, une condition doit lier les deux variables: on veut qu'elles soient relatives au même logement, et donc que le code logement soit identique. C'est ce qu'on appelle une *jointure*.

Voici la requête SQL. Remarquons au passage que le nom que l'on donne aux variables n'a aucune importance. Nous utilisons `l` pour le logement, `a` pour l'activité.

```
select l.code, l.nom
from Logement as l, Activité as a
where l.code = a.codeLogement
and   a.codeActivité = 'Ski'
```

Les seules affectations de l et a satisfaisant la formule sont marquées par des croix dans les tables ci-dessous (les champs concernés ont de plus été mis en gras). Prenez, si nécessaire, le temps de bien comprendre que d'une part la formule de condition est bien satisfaite, et d'autre part qu'il n'y a pas d'autre solution possible.

l	code	nom	capacité	type	lieu
	pi	U Pinzutu	10	Gîte	Corse
	ta	Tabriz	34	Hôtel	Bretagne
	ca	Causses	45	Auberge	Cévennes
X	**ge**	Génépi	134	Hôtel	Alpes

a	codeLogement	codeActivité	description
	pi	Voile	Pratique du dériveur et du catamaran
	pi	Plongée	Baptèmes et préparation des brevets
	ca	Randonnée	Sorties d'une journée en groupe
X	**ge**	**Ski**	Sur piste uniquement
	ge	Piscine	Nage loisir non encadrée

A partir de ces deux affectations, on construit le résultat.

code	nom
ge	Génépi

Pour maîtriser cette partie de SQL (sans doute la plus couramment utilisée), il faut bien comprendre le mécanisme mis en œuvre. Pour construire un nuplet du résultat, nous avons besoin de 1, 2 ou plus nuplets provenant de la base. Il faut identifier ces nuplets, les conditions qu'ils doivent satisfaire, et les valeurs qu'ils partagent. Ici:

- nous avons besoin d'un nuplet de la relation `Activité`, tel que le code soit `Ski`;
- nous avons besoin d'un nuplet de la relation `Logement`, puisque nous souhaitons obtenir le nom du logement en sortie;
- enfin ces nuplets doivent être relatifs au même logement, et partager donc la même valeur sur l'attribut qui identifie ce logement, respectivement `code` dans `Logement` et `codeLogement` dans `Activité`.

Ce raisonnement est très général et permet d'exprimer des requêtes SQL puissantes. Les seules conditions sont de formuler rigoureusement la requête et de comprendre le schéma de la base.

Prenons un autre exemple montrant que l'on peut utiliser la même portée pour des variables différentes. On veut obtenir les paires de logements qui sont du même type. Puisqu'il nous faut deux logements, nous avons besoin de deux variables, ayant chacune pour portée la table `Logement`. Ces deux variables doivent partager la même valeur pour l'attribut `type`. Les deux variables ont été nommées respectivement l_1 et l_2. La syntaxe SQL est donnée ci-dessous.

```
select distinct l1.nom as nom1, l2.nom as nom2
from Logement as l1, Logement as l2
where l1.type = l2.type
```

Note

Dans la syntaxe SQL, il faut résoudre les ambiguités éventuelles sur les noms d'attributs avec `as`. Ici, on a nommé le nom du premier logement `nom1` et celui du second `nom2` pour obtenir en sortie une relation de schéma `(nom1, nom2)`.

Il existe plusieurs affectations de `l1` et `l2` pour lesquelles la formule est satisfaite. La première est donnée ci-dessous: `l1` est affectée à la seconde ligne et `l2` à la quatrième.

l1	l2	code	nom	capacité	type	lieu
		pi	U Pinzutu	10	Gîte	Corse
X		ta	Tabriz	34	Hôtel	Bretagne
		ca	Causses	45	Auberge	Cévennes
	X	ge	Génépi	134	Hôtel	Alpes

Mais la formule est également satisfaite si on inverse les affectations: `l1` est à la quatrième ligne et `l2` à la seconde.

l1	l2	code	nom	capacité	type	lieu
		pi	U Pinzutu	10	Gîte	Corse
	X	ta	Tabriz	34	Hôtel	Bretagne
		ca	Causses	45	Auberge	Cévennes
X		ge	Génépi	134	Hôtel	Alpes

Et, surprise, elle est également satisfaite si les deux variables sont affectées au *même* nuplet.

l1	l2	code	nom	capacité	type	lieu
X	X	pi	U Pinzutu	10	Gîte	Corse
		ta	Tabriz	34	Hôtel	Bretagne
		ca	Causses	45	Auberge	Cévennes
		ge	Génépi	134	Hôtel	Alpes

Pour éviter les inversions et auto-égalités, on peut ajouter une condition:

```
select distinct l1.nom as nom1, l2.nom as nom2
from Logement as l1, Logement as l2
where l1.type = l2.type
and l1.nom < l2.nom
```

Le résultat de cette requête est alors:

nom1	nom2
Génépi	Tabriz

Interprétation d'une requête SQL

En résumé, *quelle que soit sa complexité*, l'interprétation d'une requête SQL peut *toujours* se faire de la manière suivante.

- Chaque variable du `from` peut être affectée à tous les nuplets de sa portée.
- Le `where` définit une condition sur ces variables: seules les affectations satisfaisant cette condition sont conservées
- Le nuplet résultat est construit à partir de ces affectations

Remarquez que ce mode d'interrogation n'indique en aucune manière, même de très loin, comment le résultat est calculé. On est (pour insister) dans une approche purement *déclarative* où le système est totalement libre de déterminer la méthode d'évaluation de chaque requête la plus efficace.

3.3 S3: Quantificateurs et négation

Supports complémentaires:

- Diapositives: SQL: quantificateurs et négation
- Vidéo sur les quantificateurs et la négation dans SQL

Jusqu'à présent les seules variables que nous utilisons sont des variables libres de la formule, définies dans la clause `from` de la syntaxe SQL. Nous n'avons pas encore rencontré de variable liée parce que nous n'avons pas utilisé les quantificateurs.

SQL propose uniquement le quantificateur existentiel. Le quantificateur universel peut être obtenu en le combinant avec la négation. Rappelons que les quantificateurs servent à exprimer des conditions sur l'ensemble d'une relation (qui peut être une relation en base, ou une relation calculée). Ils sont particulièrement utiles pour les requêtes qui comportent des *négations* (« je *ne* veux *pas* des objets qui ont telle ou telle propriété dans mon résultat »).

3.3.1 Le quantificateur `exists`

Reprenons simplement la requête qui demande les logements où l'on peut faire du ski. La requête donnée précédemment est la suivante:

```
select l.nom
from Logement as l, Activité as a
where l.code = a.codeLogement
and   a.codeActivité = 'Ski'
```

On remarque que la variable libre *a* n'est pas utilisée dans la construction du nuplet-résultat (qui ne contient que `l.nom`). On pourrait donc affecter le nuplet `a` à une variable liée, ce qui revient à formuler la requête légèrement différemment: « donnez-moi le nom des logements pour lesquels *il existe une activité* Ski ». Voici la syntaxe en SQL, où l'on utilise le mot-clé `exists` pour vérifier qu'une sous-requête a au moins un résultat.

```
select distinct l.nom
from Logement as l
where exists (select ''
              from Activité as a
              where l.code = a.codeLogement
              and a.codeActivité = 'Ski')
```

On a introduit la sous-requête suivante.

```
select ''
from Activité as a
where l.code = a.codeLogement
and a.codeActivité = 'Ski'
```

Cette sous-requête retourne un résultat dès que l'on a trouvé *au moins* un nuplet qui satisfait les conditions demandées, à savoir un code activité égal à `Ski`, et le même code logement que celui de la variable l.

Le résultat est construit à partir du `select` de premier niveau, qui ne peut accéder qu'à la variable `l`, et pas à la variable (liée) `a`.

Note

La clause du `select` imbriquée ne sert donc absolument à rien d'autre qu'à respecter la syntaxe SQL, et on peut utiliser `select ''`, `select *` ou n'importe quoi d'autre.

Cet exemple montre qu'il est possible d'exprimer une même requête avec des syntaxes différentes, que ce soit au niveau de la formulation en langage naturel ou de l'expression formelle (logique ou SQL).

Les quantificateurs permettent d'imbriquer des formules dans des formules, sans limitation de profondeur. En SQL, on peut de même avoir des imbrications de requêtes sans limitation. La lisibilité et la compréhension en sont quand même affectées.

Prenons une requête un peu plus complexe: je veux les noms des voyageurs qui sont allés dans les Alpes. Une première formulation, complètement « à plat » est la suivante:

```
select distinct v.prénom, v.nom
from Voyageur as v, Séjour as s, Logement as l
where v. idVoyageur=s.idVoyageur
and   s.codeLogement = l.code
and   l.lieu = 'Alpes'
```

Ni la variable `s`, ni la variable `l` ne sont utilisées pour construire le nuplet-résultat. On peut donc l'exprimer ainsi: « je veux les noms des voyageurs pour lesquels il existe un séjour dans les Alpes ». Ce qui donne:

```
select distinct v.prénom, v.nom
from Voyageur as v
where exists (select ''
              from Séjour as s, Logement as l
              where v. idVoyageur=s.idVoyageur
              and   s.codeLogement = l .code
              and   l.lieu = 'Alpes')
```

On pourrait même aller encore plus loin dans l'imbrication avec la requête suivante:

```
select distinct v.prénom, v.nom
from Voyageur as v
where exists (select ''
              from Séjour as s
              where v. idVoyageur=s.idVoyageur
              and exists (select ''
                          from  Logement as l
                           where s.codeLogement = l .code
                           and   l.lieu = 'Alpes')
               )
```

La troisième version correspond à la formulation « Les voyageurs tels *qu'il existe* un de leurs séjours tels que le logement *existe* dans les Alpes ». Elle n'est pas très naturelle, et, de plus, probablement la plus difficile à comprendre, ce qui ne plaide pas en sa faveur.

3.3.2 Quantificateurs et négation

Il nous reste à découvrir les requêtes probablement les plus complexes, celle où l'on exprime une négation. Voici un premier exemple : on veut les logements qui ne proposent pas de Ski. En reprenant la requête « positive » étudiée précédemment, il suffit d'ajouter une négation devant le quantificateur existentiel.

```
select distinct l.nom
from Logement as l
where not exists (select ''
                  from Activité as a
                  where l.code = a.codeLogement
                  and a.codeActivité = 'Ski')
```

C'est la seule manière de l'exprimer correctement. Elle donne le résultat suivant:

nom
Causses
U Pinzutu
Tabriz

Vous devriez être convaincus que la requête suivante est très différente (et ne correspond pas à ce que l'on souhaite). L'opérateur `!=` signifie *différent de* en SQL.

```
select l.nom
from Logement as l
where  exists (select ''
                  from Activité as a
                  where l.code = a.codeLogement
                  and a.codeActivité != 'Ski')
```

Dont le résultat est:

nom
Causses
Génépi
U Pinzutu

Réfléchissez au sens de cette requête, trouvez le résultat sur notre petite base. Rappelez-vous que les quantificateurs servent à exprimer une condition sur un ensemble de nuplets, pas sur chaque nuplet en particulier.

Le `not exists` est la porte d'entrée pour exprimer le quantificateur universel, c'est à dire vérifier qu'une propriété soit satisfaite par tous les nuplets d'un ensemble. Supposons que l'on cherche les voyageurs qui sont allés dans *tous* les logements. On reformule cette requête avec deux négations: on cherche les voyageurs tels *qu'il n'existe pas* de logement où *ils ne sont pas* allés.

```
select distinct v.prénom, v.nom
from Voyageur as v
where  not exists (select ''
                   from Logement as l
                   where not exists  (select ''
                                      from Séjour as s
                                      where l.code = s.codeLogement
                                      and   v.idVoyageur = s.idVoyageur)
                   )
```

Vous devriez obtenir:

prénom	nom
Nicolas	Bouvier

Vous savez maintenant tout sur la version déclarative de SQL, qui n'est rien d'autre qu'une syntaxe concrète pour exprimer des formules ouvertes sur une base de données. Tout ce qui peut s'exprimer par une formule logique est exprimable en SQL. Ni plus, ni moins. Inversement, tout ce qui ne s'exprime pas par une formule (boucles, incrémentations, etc.) ne s'exprime pas en SQL.

Dans un prochain chapitre, nous verrons la version procédurale, mais il est important de préciser qu'elle n'apporte *rien* en terme de possibilités d'expression. En d'autres termes, vous avez déjà, avec ce que nous venons d'étudier, la capacité d'exprimer toutes les requêtes possibles (à l'exception des agrégations). La version procédurale n'est qu'une manière alternative de concevoir l'interrogation d'une base relationnelle.

Prenez le temps de bien maîtriser ce qui précède, car la compréhension du *sens* de ce que l'on exprime avec les formules de logique des prédicats est la condition nécessaire et suffisante pour utiliser correctement SQL.

3.4 S4: Conception d'une requête SQL

Supports complémentaires:

- Diapositives: SQL: construction d'une requête
- Vidéo sur la construction d'une requête SQL

Vous devriez à ce stade connaître et comprendre l'interprétation d'une requête SQL. Redonnons-la encore une fois sous une forme un peu différente:

- Le *résultat* d'une requête est une relation constituée de nuplets.
- Chaque nuplet du résultat est construit à partir d'un ensemble de n nuplets $t_1, t_2, \cdots, t_n$ *provenant de la base de données.*
- Ces n nuplets doivent satisfaire un ensemble de *conditions.*

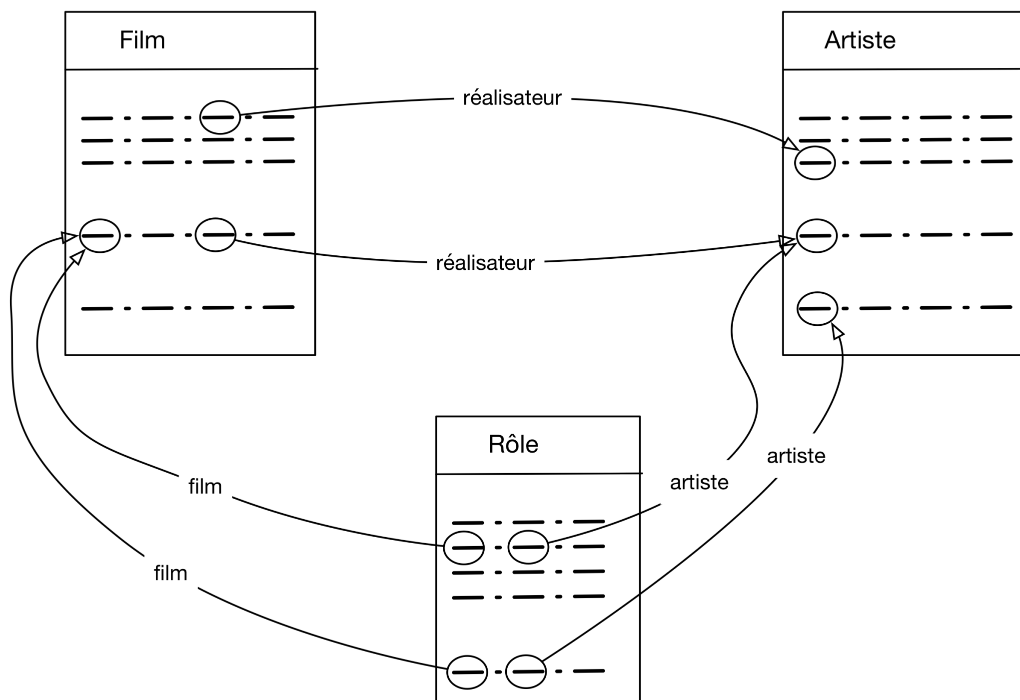
La *construction* d'une requête consiste :

- à indiquer de quels nuplets $t_1, t_2, \cdots, t_n$ nous avons besoin, et d'où chacun provient (c'est la clause `from`)
- à exprimer les conditions avec la clause `where`
- à indiquer comment on construit un nuplet du résultat avec la clause `select`.

C'est tout. Le système pour sa part se charge de trouver toutes les combinaisons possibles des $t_1, t_2, \cdots, t_n$, de tester les conditions, de construire le résultat. Le tout en choisissant la méthode la plus efficace.

Nous sommes maintenant en mesure de tenter de décrire le processus mental qui nous permet de construire une requête SQL pour répondre à un besoin donné. Le processus que nous décrivons s'appuie sur une vision de la structure de la base qui comprend, au minimum, la liste des tables, leurs clés primaires et les clés étrangères. On établit cette vision à partir du schéma, comme le montre par exemple la Fig. 7 pour trois tables de la base des films (vu en TP). La bonne connaissance du schéma, et sa compréhension, sont des pré-requis pour exprimer des requêtes SQL correctes.

les liens entre les attributs des différentes relations



Commençons par les requêtes conjonctives, dans lesquelles la principale difficulté est de construire les jointures.

3.4.1 Conception d'une jointure

Le mécanisme de base consiste donc à se représenter les nuplets qui permettront de construire un des nuplets du résultat. Dans les cas les plus simples, un seul suffit. Pour la requête « Quelle est l'année de naissance de G. Depardieu » par exemple, on construit un nuplet du résulat à partir d'un nuplet de la table Artiste, dont l'attribut « nom » est « Depardieu », et dont l'attribut « âge » est l'information qui nous intéresse. On désigne ce nuplet par un nom, par exemple a. L'image mentale à construire est celle de la Fig. 6.

C'est très élémentaire (pour l'instant) mais toute la requête SQL est déjà codée dans cette représentation.

- Chaque nuplet désigné doit être défini dans le `from`.
- Les contraintes satisfaites par ce nuplet constituent le `where` (nom="Depardieu").
- La clause `select` est toujours triviale (on choisit les attributs à conserver).

Ce qui donne sur ce premier exemple:

```
select annéeNaissance
from Artiste as a
where a.nom='Depardieu'
```

Entrons dans le vif du sujet avec la requête « Titre des films avec pour acteur Depardieu ». Cette fois l'image mentale à construire est celle de la Fig. 7. Nous avons besoin, pour construire chaque nuplet du résultat, de trois nuplets de la base: un film, un artiste, un rôle. Dès que nous avons plusieurs nuplets, il faut indiquer de quelle manière ils sont liés: ici les liens sont (comme à peu près toujours) définis par le critère d'égalité des clés primaires et clés étrangères.

On a donné un nom à chaque nuplet, soit f, r et a. La construction de la requête s'ensuit quasiment automatiquement.

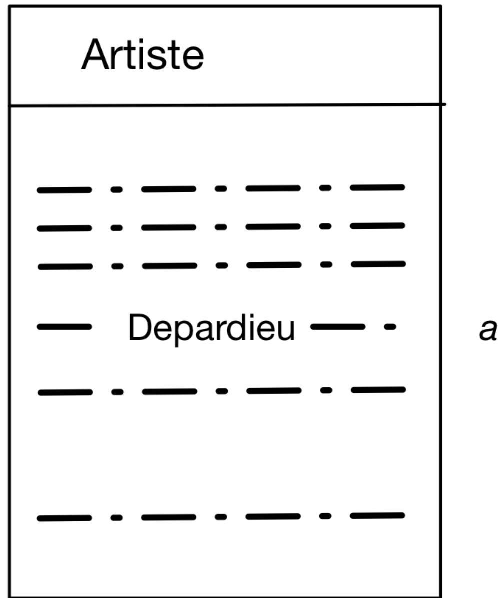


Figure 6: Interrogation avec un seul nuplet

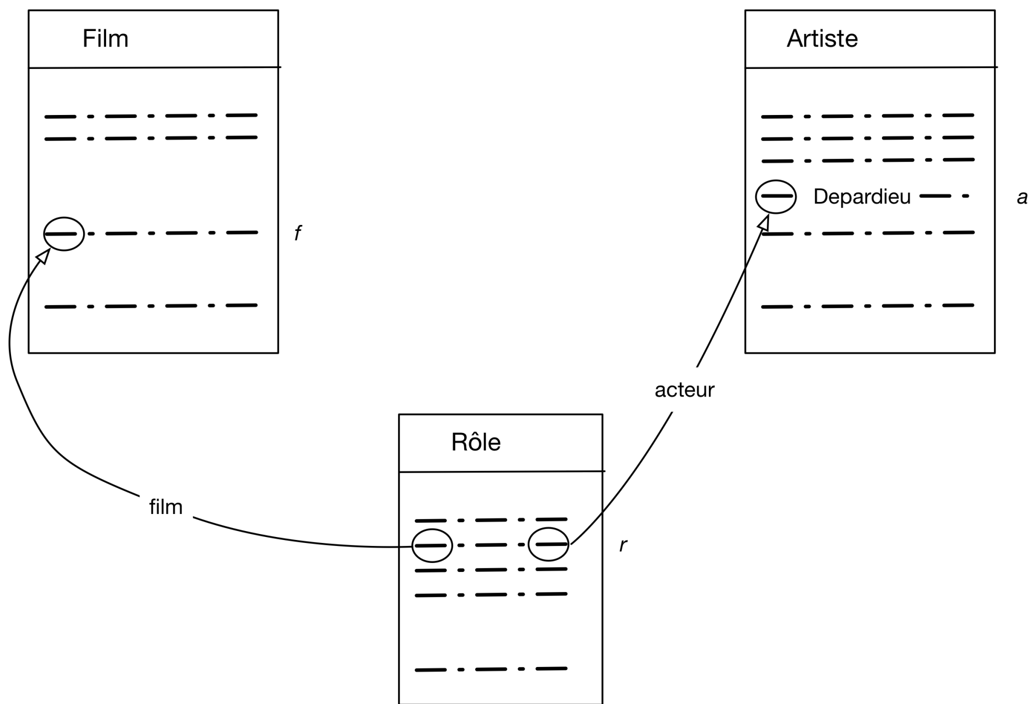


Figure 7: Les nuplets impliqué dans la recherche des films avec Depardieu

```sql
select f.titre
from Artiste as a, Rôle as r, Film as f
where a.nom='Depardieu'
and a.idArtiste = r.idActeur
and r.idFilm = f.idFilm
```

Notez que les contraintes sur les nuplets sont soit des égalités entre attributs, soit l'égalité entre un attribut et une constante. Quand nous ajouterons la négation, un troisième type de contrainte apparaîtra, celui de l'existence ou non d'un résultat pour une sous-requête.

Remarquez également comment on se repose sur l'interpéteur SQL pour faire l'essentiel du travail: trouver les nuplets satisfaisant les constraintes, énumérer toutes les combinaisons valides à partir de la base, et construire le résultat.

Voici un exemple un peu plus compliqué qui ne change rien au raisonnement: on veut les titres de film avec Depardieu et Deneuve. L'image à construire est celle de la Fig. 8. Ici il faut concevoir qu'il nous faut deux nuplets de la table Artiste, l'un avec pour nom Depardieu (*a1*), et l'autre avec pour nom Deneuve (*a2*). Ces deux nuplets sont liés à deux nuplets *distincts* de la table Rôle, nommons-les *r1* et *r2*. Ces deux derniers nuplets sont liés au *même* film *f* .

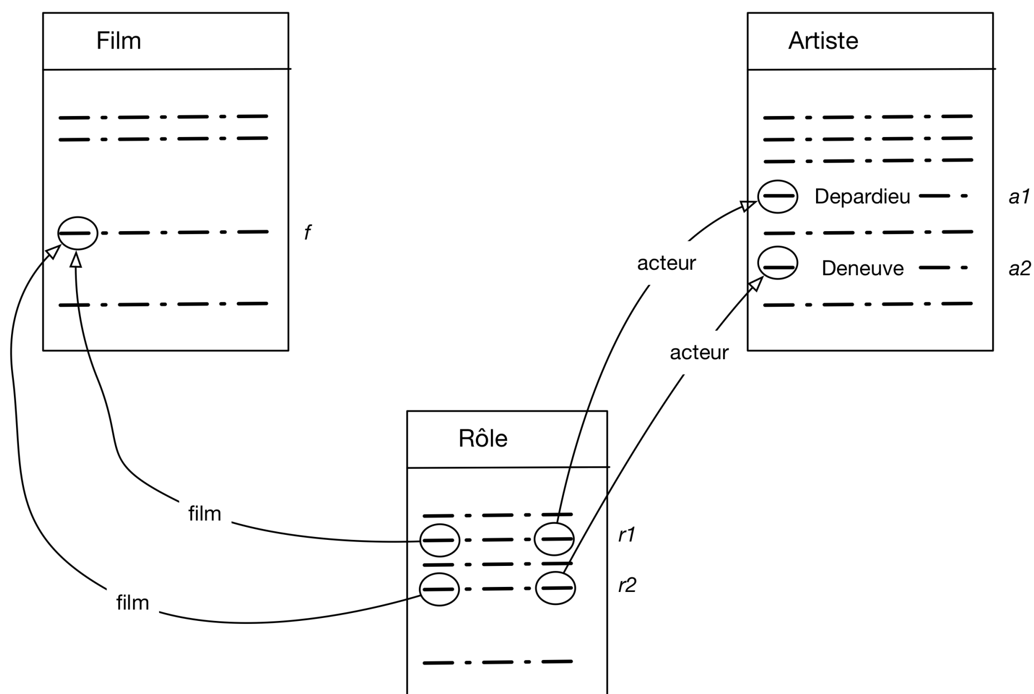


Figure 8: Les nuplets impliqué dans la recherche des films avec Depardieu et Deneuve

À partir de la Fig. 8, la construction syntaxique de la requête SQL est encore une fois directe: énumération des variables-nuplets dans le `from`, contraintes dans le `where`, clause `select` selon les besoins.

```sql
select *
from Artiste as a1, Artiste as a2, Rôle as r1, Rôle as r2, Film as f
where a1.nom='Depardieu'
and a2.nom='Deneuve'
and a1.idArtiste = r1.idActeur
and a2.idArtiste = r2.idActeur
and r1.idFilm = f.idFilm
and r2.idFilm = f.idFilm
```

Voici deux exemples complémentaires. Le premier recherche les films réalisés par Q. Tarantino en 1994. L'image mentale est celle de la Fig. 9.

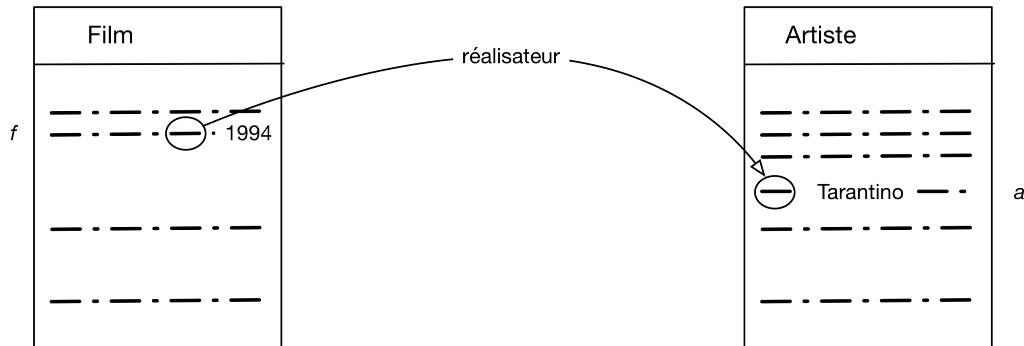


Figure 9: Recherche les films réalisés par Q. Tarantino en 1994

La requête correspondante est bien entendu celle-ci.

```
select *
from Artiste as a, Film as f
where a.nom='Tarantino'
and  f.année = 1994
and a.idArtiste = f.idRéalisateur
```

Le second exemple recherche les films réalisés par Q. Tarantino en 1994 dans lesquels il joue lui-même dans tant qu'acteur. Je vous laisse étudier et interpréter la Fig. 10 et exprimer vous-même la requête SQL.

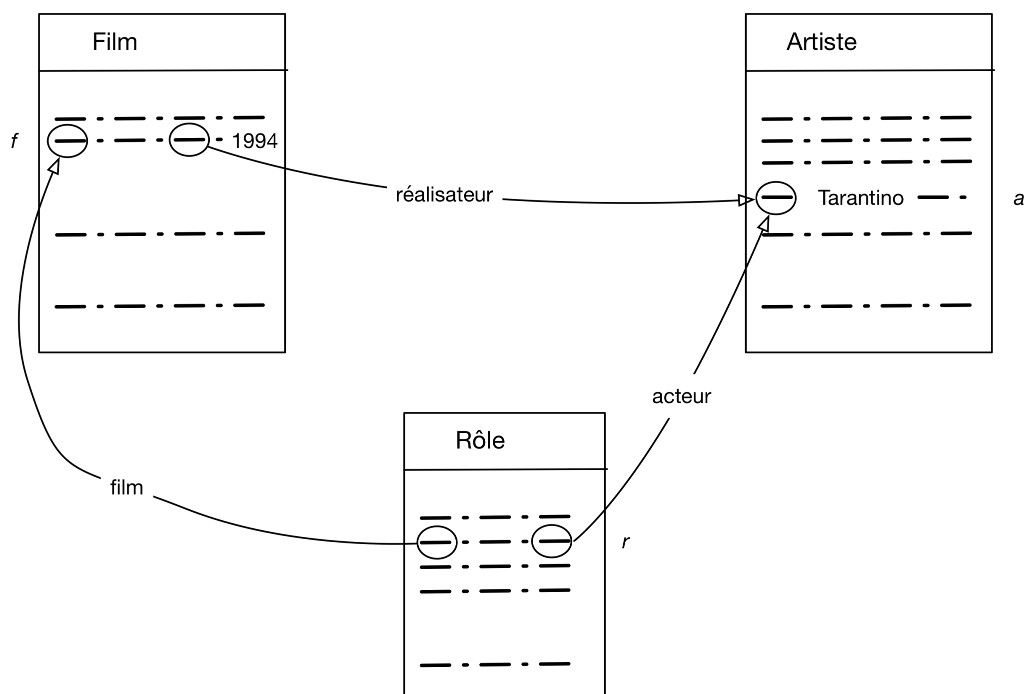


Figure 10: Recherche les films réalisés par Q. Tarentino en 1994 dans lequels il joue

3.4.2 Conception des requêtes imbriquées

Que se passe-t-il en cas de requête imbriquée, et surtout en cas de nécessité d'exprimer une négation? Les principes précédents restent valables: on identifie les nuplets de la base qui permettent de produire un nuplet du résultat, on construit la requête comme précédemment, *et la requête imbriquée n'est qu'une contrainte supplémentaire sur ces nuplets.* La seule particularité des requêtes imbriquées est que la contrainte porte sur un ensemble, et pas sur une valeur atomique.

Prenons un exemple: je veux les titres de film avec Catherine Deneuve mais sans Gérard Depardieu. On commence par la solution partielle qui consiste à trouver les films avec Deneuve

```
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
and r.idActeur = a.idArtiste
and a.nom='Deneuve'
```

Maintenant on ajoute la contrainte suivante sur le film *f*: dans l'ensemble des acteurs du film **f**, on ne doit pas trouver Gérard Depardieu. L'ensemble des acteurs du film *f* qui se nomment Depardieu est obtenu par une requête fonction de *f*, cette requête est ajoutée dans le `where` et on obtient la requête complète

```
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
and r.idActeur = a.idArtiste
and a.nom='Deneuve'
and not exists (select * from Rôle as r2, Artiste as a2
                where f.idFilm=r2.idFilm and r2.idActeur=a2.idActeur
                and a2.nom='Depardieu')
```

Il faut bien être conscient que cette condition supplémentaire porte sur le film *f*, et que *f* doit impérativement intervenir dans la requête imbriquée. La requête suivante par exemple est fausse:

```
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
and r.idActeur = a.idArtiste
and a.nom='Deneuve'
and not exists (select * from Rôle as r2, Artiste as a2
                where r2.idActeur=a2.idActeur
                and a2.nom='Depardieu')
```

La requête imbriquée est ici indépendante des nuplets de la variable principale, et on peut donc évaluer son résultat dès le début: soit il existe un acteur nommé Depardieu (*quel que soit le film*), le `not exists` est toujours faux et le résultat est toujours vide; soit il n'en existe pas, le `not exists` est toujours vrai et ne sert donc à rien.

3.4.3 La disjonction

Reste à discuter de la disjonction. Il existe une propriété assez utile des formules logiques: on peut toujours les mettre sous une forme dite « normale disjonctive », autrement dit comme la disjonction de conjonctions. En pratique cela implique que toute requête comprenant un « ou » peut s'écrire comme l'union de requêtes conjonctives écrites sans « ou ». Cherchons les films avec Deneuve ou Depardieu.

```
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
```

```
and r.idActeur = a.idArtiste
and a.nom='Deneuve'
  union
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
and r.idActeur = a.idArtiste
and a.nom='Depardieu'
```

Ce n'est pas très concis. Il est à peu près toujours possible de trouver une formulation plus condensée avec le « or ». Ici ce serait:

```
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
and r.idActeur = a.idArtiste
and (a.nom='Deneuve' or nom='Depardieu')
```

Il n'existe pas de règle générale permettant de trouver la bonne formulation sans réfléchir. La bonne maîtrise des principes de logique, d'équivalence de formule et d'interprétation sont les connaissances clés.

Les principes exposés ici sont très importants. Même s'ils peuvent vous sembler parfois éloignés de vos objectifs pratiques, tout ce qui précède devrait j'espère vous convaincre que maîtriser SQL, c'est d'abord être capable d'aborder la formulation des requêtes de manière rigoureuse, pas de produire une syntaxe finalement relativement simple. À vous de jouer.

4 SQL, compléments et supports

Ce chapitre présente les compléments du langage d'interrogation SQL (la partie dite *Langage de Manipulation de Données* ou LMD) dans le cadre d'un récapitulatif. Ces compléments présentent peu de difficulté dans la mesure où la véritable complexité réside d'une part dans l'interprétation des requêtes complexes qui font parfois appel à des logiques sophistiquées et d'autre part dans la multiplicité des variantes syntaxiques qui peut parfois troubler.

Le chapitre précédent devrait avoir réglé ces problèmes d'interprétation. Vous savez maintenant que le paradigme d'interrogation déclaratif de SQL, vous découvrez l'autre procédural au cours du chapitre suivant. Dans ce chapitre, nous allons compléter quelques capacités de SQL non abordés avec l'approche déclarative.

La base prise comme exemple dans ce chapitre est celle des immeubles. Elle est accessible ici.

4.1 S1: le bloc `select-from-where`

Supports complémentaires:

- Diapositives: le bloc 'select-from-where
- Vidéo sur le bloc 'select-from-where

Dans cette session, nous étudions les compléments à la forme de base d'une requête SQL, que nous appelons *bloc*, résumée ainsi:

```
select liste_expressions
from   relations_sources
[where liste_conditions]
[order by critère_de_tri]
```

Parmi les quatre clauses `select`, `from`, `where` et `order by`, les deux dernières sont optionnelles. 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 des Martyrs
2	Barabas	2 allée du Grand Turc

L'ordre des trois clauses `select from` et `where` est trompeur pour la signification d'une requête. Comme nous l'avons déjà détaillé dans les chapitres qui précédent l'intepprétation s'effectue *toujours* de la manière suivante:

- la clause `from` définit l'espace de recherche en fonction d'un ensemble de sources de données;
- la clause `where` exprime un ensemble de conditions sur la source: seuls les nuplets pour lesquels ces conditions sont satisfaites sont conservés;
- enfin la clause `select` construit un nuplet-résultat grâce à une liste d'expressions appliquées aux nuplets de la source ayant passé le filtre du `where`.

4.1.1 La clause `from`

L'espace de recherche est défini dans la clause `from` par une ou 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.

Voici une première requête qui ramène les immeubles dont l'id vaut 1.

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

Il n'aura pas échappé au lecteur attentif que le résultat est lui-même une table (calculée et non basée). 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.

L'interprétation du `from` est indépendante de l'origine des tables: tables basées, tables calculées. Comme nous l'avons vu dans les chapitres précédents, il existe deux manières de spécifier l'espace de recherche avec le `from`. La première est la forme déclarative dans laquelle on sépare le nom des tables par des virgules.

```
select * from Immeuble as i, Appart as a
```

Dans ce cas, le nom d'une table sert à définir une variable nuplet (voir chapitre SQL, langage déclaratif) à laquelle on peut affecter tous les nuplets de la table. Les variables peuvent être explicitement nommées avec la mot-clé `as` (elles s'appellent `i` et `a` dans la requête ci-dessus).

On peut aussi omettre le `as`, dans ce cas le nom de la variable est (implcitement) le nom de la table.

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

Un cas où le `as` est obligatoire est l'auto-jointure: on veut désigner deux nuplets de la même table. Exemple: on veut les paires d'appartement du même immeuble.

```
select a1.no, a2.no
from Appart as a1, Appart as a2
where a1.idImmeuble = a2.idImmeuble
```

En l'absence du `as` et de l'utilisation du nom de la variable comme préfixe, il y aurait ambiguité sur le nom des attributs.

La deuxième forme du `from` définit l'espace de recherche par une opération algébrique, nous aborderons ce point dans le chapitre suivant.

Dernière précision au sujet du `from`: l'ordre dans lequel on énumère les tables n'a aucune importance.

4.1.2 La clause `where`

La clause `where` permet d'exprimer des conditions portant sur les nuplets désignés par la clause `from`. Ces conditions 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 de fonctions, et Θ est l'un des opérateurs de comparaison classique $<$ $>$ $<=$ $>=$ $!=$ ou $=$.

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	prénom	nom	profession	idAppart
4	Barnabé	Simplet	Acteur	102
5	Alphonsine	Joyeux	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 des Martyrs

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 niveau.

```
select * from Personne
where idAppart in (select id from Appart where niveau > 3)
```

id	prénom	nom	profession	idAppart
2	Alice	Grincheux	Cadre	103
3	Léonie	Atchoum	Stagiaire	100

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.

4.1.3 Valeurs manquantes: le `null`

En théorie, dans une table relationnelle, tous les attributs ont une valeur. En pratique, certaines valeurs peuvent être inconnues ou manquantes: on dit qu'elles sont à `null`. Le `null` *n'est pas une valeur spéciale*, c'est une absence de valeur.

Note

Les valeurs à `null` sont une source de problème, car elles rendent parfois le résultat des requêtes difficile à comprendre. Mieux vaut les éviter si c'est possible.

Il est impossible de déterminer quoi que ce soit à partir d'une valeur à `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 tous les nuplets.

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

Mais la présence d'un `null` empêche l'inclusion du nuplet correspondant dans le résultat.

id	prénom	nom	profession	idAppart
2	Alice	Grincheux	Cadre	103
3	Léonie	Atchoum	Stagiaire	100
4	Barnabé	Simplet	Acteur	102
5	Alphonsine	Joyeux	Rentier	201
6	Brandon	Timide	Rentier	104
7	Don-Jean	Dormeur	Musicien	200

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

```
select *
from Personne
where prénom 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`, = min(val1 val2)
- `val1 or val2` = max(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, soit en utilisant le test `is null` (ou son complément `is not null`). La requête ci-dessous ramène tous les nuplets de la table, même en présence de `null`.

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

id	prénom	nom	profession	idAppart
1		Prof	Enseignant	202
2	Alice	Grincheux	Cadre	103
3	Léonie	Atchoum	Stagiaire	100
4	Barnabé	Simplet	Acteur	102
5	Alphonsine	Joyeux	Rentier	201
6	Brandon	Timide	Rentier	104
7	Don-Jean	Dormeur	Musicien	200

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

4.1.4 La clause `select`

Finalement, une fois obtenus les nuplets du `from` qui satisfont le `where`, on crée à partir de ces nuplets le résultat final avec les expressions du `select`.

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 concat(prénom, ' ', nom) as 'nomComplet'
from Personne
```

nomComplet
null
Alice Grincheux
Léonie Atchoum
Barnabé Simplet
Alphonsine Joyeux
Brandon Timide
Don-Jean Dormeur

Le résultat montre que l'une des valeurs est à `null`. 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`. Dans ce cas, il faut utiliser une fonction (spécifique à chaque système) à qui remplace la valeur à `null` par une valeur de remplacement. Voici la version MySQL (fonction `ifnull(attribut, remplacement)`).

```
select concat(ifnull(prénom,' '), ' ', nom) as 'nomComplet'
from Personne
```

Une « expression » dans la clause `select` 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, niveau, 18 as 'EurosParm2'
from Appart
```

surface	niveau	EurosParm2
150	14	18
50	15	18
200	2	18
50	5	18
75	3	18
150	0	18
250	1	18
250	2	18

Les attributs `surface` et `niveau` proviennent de *Appart* alors que 18 est une constante qui sera répétée autant de fois qu'il y a de nuplets 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 le niveau.

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

no	surface	niveau	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, niveau, surface,
        case when surface <= 50 then 'Petit'
             when surface > 50 and surface <= 100 then 'Moyen'
             else 'Grand'
        end as categorie
from Appart
```

no	niveau	surface	categorie
1	14	150	Grand
34	15	50	Petit
51	2	200	Grand
52	5	50	Petit
43	3	75	Moyen
10	0	150	Grand
1	1	250	Grand
2	2	250	Grand

4.1.5 Jointure interne, jointure externe

La jointure est une opération indispensable dès que l'on souhaite combiner des données réparties dans plusieurs tables. Nous avons déjà étudié en détail la conception et l'expression des jointures. On va se contenter ici de montrer quelques exemples en forme de récapitulatif, sur notre base d'immeubles.

Note

Il existe beaucoup de manières différentes d'exprimer les jointures en SQL. 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.

Jointure interne Prenons l'exemple d'une requête cherchant la surface et le niveau de l'appartement de M. Barnabé Simplet.

```
select p.nom, p.prénom,  a.surface, a.niveau
from Personne as p, Appart as a
where prénom='Barnabé'
and nom='Simplet'
and    a.id = p.idAppart
```

nom	prénom	surface	niveau
Simplet	Barnabé	200	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 variables-nuplet dont ils proviennent.

Notez que la levée de l'ambiguité en préfixant par le nom de la variable-nuplet 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.

Comme dans la très grande majorité des cas la jointure consiste à exprimer une égalité entre un identifiant d'une table et l'attribut correspondant dans une autre table. Mais rien n'empêche d'exprimer des conditions de jointure sur n'importe quel attribut.

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

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

La requête exprimant ces conditions est donc:

```
select a1.id as idAppart1, a1.surface as surface1, a1.niveau as niveau1,
       a2.id as idAppart2, a2.surface as surface2, a2.niveau as niveau2
from Appart a1, Appart a2
where a1.id != a2.id
and a1.surface = a2.surface
and a1.idImmeuble = a2.idImmeuble
```

Ce qui donne le résultat suivant:

idAppart1	surface1	niveau1	idAppart2	surface2	niveau2
103	50	5	101	50	15
101	50	15	103	50	5
202	250	2	201	250	1
201	250	1	202	250	2

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`.

Voici quelques exemples complémentaires de jointure.

Qui habite un appartement de plus de 200 m2?

```
select prénom, nom, profession
from Personne, Appart
where idAppart = 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 prénom, p.nom, no, surface, niveau
from   Personne as p, Appart as a, Immeuble as i
where  p.idAppart=a.id
and    a.idImmeuble=i.id
and    i.nom='Barabas'
```

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

```
select prénom, nom, quotePart
from   Personne as p, Propriétaire as p2, Appart as a
where  p.id=p2.idPersonne /* p est propriétaire */
and    p2.idAppart=a.id   /* de l'appartement a */
and    p.idAppart=a.id   /* et il y habite      */
```

De quel(s) appartement(s) Alice Grincheux 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, niveau, surface
from  Personne as p, Appart as a, Immeuble as i, Propriétaire as p2
where  p.id=p2.idPersonne /* Jointure PersonnePropriétaire */
and    p2.idAppart = a.id /* Jointure PropriétaireAppart */
and    a.idImmeuble= i.id /* Jointure AppartImmeuble */
and    p.nom='Grincheux' and p.prénom='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`).

L'approche déclarative d'expression des jointures 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` (voir chapitre suivant) ou (pas toujours) requêtes imbriquées. À l'exception notable des jointures externes, elles n'apportent aucune expressivité supplémentaire. Toutes ces variantes constituent des moyens plus ou moins commodes d'exprimer différemment la jointure.

Jointure externe Qu'est-ce qu'une jointure externe? Effectuons la requête qui affiche tous les appartements avec leur occupant.

```
select idImmeuble, no, niveau, surface, nom, prénom
from  Appart as a, Personne as p
where  p.idAppart=a.id
```

Voici ce que l'on obtient:

idImmeuble	no	niveau	surface	nom	prénom
2	2	2	250	Prof	null
1	52	5	50	Grincheux	Alice
1	1	14	150	Atchoum	Léonie
1	51	2	200	Simplet	Barnabé
2	1	1	250	Joyeux	Alphonsine
1	43	3	75	Timide	Brandon
2	10	0	150	Dormeur	Don-Jean

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 tous ses nuplets, même ceux qui ne trouvent pas de correspondant dans la table de droite, seront prises en compte. Les nuplets de la table de droite sont en revanche optionnels.

Si pour un nuplet de la table de gauche on trouve un nuplet 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 `left` est optionnel.

```sql
select idImmeuble, no niveau, surface, nom, prénom
from  Appart as a left outer join Personne as p on (p.idAppart=a.id)
```

Note

Notez bien que l'expression `Appart as a left outer join Personne as p on (p.idAppart=a.id)` définie le calcul de la jointure externe entre les deux relations. Nous verrons plus en détails cette syntaxe au chapitre suivant.

On obtient le résultat suivant:

idImmeuble	no	niveau	surface	nom	prénom
1	1	14	150	Atchoum	Rachel
1	34	15	50	null	null
1	51	2	200	Simplet	Barnabé
1	52	5	50	Grincheux	Alice
2	1	1	250	Joyeux	Alphonsine
2	2	2	250	Prof	null

Notez les deux attributs `prénom` 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.

```sql
select i.nom as nomImmeuble, no, niveau, surface, p.nom as nomPersonne, prénom
from  Immeuble  as i,
      (Appart as a left outer join Personne as p
                   on (p.idAppart=a.id))
where i.id=a.idImmeuble
order by i.id, a.no
```

4.1.6 Tri et élimination de doublons

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

```sql
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 nuplets dans le résultat intermédiaire après exécution des clauses `from` et `where`. En général, on ne souhaite pas conserver ces nuplets identiques dont la répétition n'apporte aucune information. Le mot-clé `distinct` placé juste après le `select` permet d'éliminer ces doublons.

```sql
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 niveau.

```
select *
from Appart
order by surface, niveau
```

id	surface	niveau	idImmeuble	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, niveau desc
```

id	surface	niveau	idImmeuble	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.

4.2 S2: Requêtes et sous-requêtes

Supports complémentaires:

Pas de support vidéo pour cette session qui ne fait que récapituler les différentes syntaxes équivalentes pour exprimer une même requête. Ne vous laissez pas troubler par la multiplicité des options offertes par SQL. En choisissant un dialecte et un seul (vous avez compris que je vous recommande la partie déclarative, logique de SQL) vous pourrez tout exprimer sans avoir à vous poser des questions sans fin. Vos requêtes n'en seront que plus cohérentes et lisibles.

Dans tout ce qui précède, les requêtes étaient « à plat », avec un seul bloc `select-from-where`. SQL est assez riche (ou assez inutilement compliqué, selon les goûts) pour permettre des expressions complexes combinant plusieurs blocs. On a dans ce cas une requête principale, et des sous-requêtes, ou requêtes imbriquées.

Disons-le tout de suite: à l'exception des requêtes avec négation `not exists`, toutes les requêtes imbriquées peuvent s'écrire de manière équivalente à plat, et on peut juger que c'est préférable pour des raisons de lisibilité et de cohérence d'écriture. Cette session essaie en tout cas de clarifier les choses.

4.2.1 Requêtes imbriquées

Reprenons l'exemple de la requête trouvant la surface et le niveau de l'appartement de M. Simplet. On peut l'exprimer avec une requête imbriquée de deux manières. La première est la forme déclarative classique.

```
select surface, niveau
from Appart as a, Personne as p
where p.prénom='Barnabé' and p.nom='Simplet'
and a.id = p.idAppart
```

On remarque qu'aucun attribut de la table `Personne` n'est utilisé pour construire le résultat. On peut donc utiliser une sous-requête (ou requête imbriquée).

```
select surface, niveau
from Appart
where id  in (select idAppart
              from Personne
              where prénom='Barnabé' and nom='Simplet')
```

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 prénom, nom, surface, niveau
from   Appart as a, Personne as p
where  a.id = p.idAppart
and    (p.id, p.idAppart)
         in (select idPersonne, idAppart from Propriétaire)
```

prénom	nom	surface	niveau
null	Prof	250	2
Alice	Grincheux	50	5
Alphonsine	Joyeux	250	1

Il est bien entendu assez direct de réécrire la requête ci-dessus comme une jointure classique (exercice). 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 idImmeuble from Appart where surface=50)
```

id	nom	adresse
1	Koudalou	3 rue des Martyrs

La requête directement réécrite en jointure donne le résultat suivant:

```
select  i.*
from    Immeuble as i, Appart as a
where   i.id=a.idImmeuble
and     surface=50
```

id	nom	adresse
1	Koudalou	3 rue des Martyrs
1	Koudalou	3 rue des Martyrs

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 de placer 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. Cela revient, dans le cadre formel qui soutient SQL, à appliquer une quantification sur une collection constituée par une requête.

Ces conditions sont les suivantes:

- `exists R`: renvoie `true` si R n'est pas vide `false` sinon.
- `t in R` où est un nuplet 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 un* des nuplets de la table R renvoie `true`.
- `v cmp all R` où *cmp* est un comparateur SQL ($<$ $>$ $=$ etc.): renvoie `true` si la comparaison avec *tous* les nuplets 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 `all` peut se réécrire avec 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 le niveau le plus élevé de l'immeuble 1.

```
select * from Appart
    where idImmeuble=1
    and     niveau >= all (select niveau from Appart where idImmeuble=1)
```

Le `all` exprime une comparaison qui vaut pour *toutes* les nuplets ramenés par la requête imbriquée. La formulation avec `any` s'écrit:

```
select * from Appart
    where idImmeuble=1
    and    not (niveau < any (select niveau from Appart where idImmeuble=1))
```

Rien de nouveau du point de vue expressif: on peut prendre les étages tels *qu'il n'existe pas* un niveau supérieur.

```
select * from Appart as a1
    where a1.idImmeuble=1
    and     not exists (select *
                  from Appart as a2
                  where a2.idImmeuble=1 and a1.niveau < a2.niveau)
```

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.

Note

Dans certains systèmes de base de données, l'opérateur `ALL` et `ANY` n'est pas défini. C'est le cas de SQLite, le système utilisé pour les TPs, il faut donc privilégier la forme équivalente avec `NOT EXISTS`.

4.2.2 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. Barnabé Simplet exprimée avec `exists`:

```
select * from Appart
where  exists  (select * from Personne
                where  prénom='Barnabé' and nom='Simplet'
                and    Personne.idAppart=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 Barnabé Simplet. 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.

Les jointures dans lesquelles le résultat est construit à partir d'une seule table 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 prénom, nom, profession
from Personne
where idAppart in (select id from Appart where surface >= 200)
```

Avec `exists`:

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

Qui habite le Barabas?

Avec `in`:

```
select prénom, nom, no, surface , niveau
from   Personne as p, Appart as a
where  p.idAppart=a.id
and a.idImmeuble in
         (select id from Immeuble
          where  nom='Barabas')
```

Avec `exists`:

```
select prénom, nom, no, surface, niveau
from Personne as p, Appart as a
where  p.idAppart=a.id
and exists (select * from Immeuble i
            where  i.id=a.idImmeuble
            and i.nom='Barabas')
```

Important

Dans une sous-requête associée à la clause `exists` peu importent les attributs du `select` puisque la condition se résume à: cette requête ramène-t-elle au moins un nuplet ou non? On peut donc systématiquement utiliser `select *` ou `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 Grincheux est-elle propriétaire et dans quel immeuble? » écrite avec plusieurs niveaux.

```
select  i.nom, no, niveau, surface
from  Immeuble as i, Appart as a
where  a.idImmeuble= i.id
and    a.id in
             (select idAppart
              from Propriétaire
              where idPersonne in
                       (select id
                        from Personne
                        where  nom='Grincheux'
                        and prénom='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.

4.2.3 Requêtes avec négation

Les sous-requêtes 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 idAppart from Personne)
```

On obtient comme résultat.

id	no	surface	niveau	idImmeuble
101	34	50	15	1

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 le niveau maximal d'un immeuble, et on utilisera cet niveau pour sélectionner un ou plusieurs appartements, le tout avec une requête correlée pour ne comparer que des appartements situés dans le même immeuble.

```sql
select *
from Appart as a1
where niveau =  (select max(niveau) from Appart as a2
                 where a1.idImmeuble=a2.idImmeuble)
```

id	surface	niveau	idImmeuble	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* de niveau 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.

```sql
select *
from Appart as a1
where not exists  (select * from Appart as a2
                   where a2.niveau > a1.niveau
                   and a1.idImmeuble = a2.idImmeuble)
```

Autre manière d'exprimer la même chose: si le niveau est le plus élevé, tous les autres sont situés à un niveau 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.

```sql
select *
from Appart as a1
where niveau >= all (select niveau from Appart as a2
                 where a1.idImmeuble=a2.idImmeuble)
```

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`.

```sql
select *
from Personne
where id not in (select idPersonne from Propriétaire)

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

4.3 S3: Agrégats

Supports complémentaires:

- Diapositives: agrégats
- Vidéo sur les agrégats

Les requêtes d'agrégation en SQL consistent à effectuer des regroupements de nuplets 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 nuplets. 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 nuplets 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(*) as nbPersonnes, count(prénom) as nbPrénoms, count(nom) as nbNoms
from Personne
```

Ce qui donne:

nbPersonnes	nbPrénoms	nbNoms
7	6	7

On obtient 7 pour le nombre de nuplets, 6 pour le nombre de prénoms, et 7 pour le nombre de noms. En effet, l'attribut `prénom` est à `null` pour la première personne et n'est en conséquence pas pris en compte par la fonction d'agrégation. Pour compter tous les nuplets, on doit utiliser `count(*)`. On peut aussi compter le nombre de valeurs distinctes dans un groupe avec `count(distinct <expression>)`.

4.3.1 La clause `group by`

Le rôle du `group by` est de partitionner le résultat d'un bloc `select from where` en fonction d'un critère (un ou plusieurs attributs, 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   idAppart, sum(quotePart) as totalQP
from     Propriétaire
group by idAppart
```

idAppart	totalQP
100	100
101	100
102	100
103	100
104	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 nuplet, d'une part la (ou les) valeur(s) du (ou des)

attribut(s) de partitionnement (ici `idAppart`), d'autre part l'ensemble de nuplets dans lesquelles on trouve cette valeur. Ces nuplets « imbriqués » sont séparés par des points-virgule dans la représentation ci-dessous.

idAppart	Groupe	count
100	(idPersonne=1 quotePart=33 ; idPersonne=5 quotePart=67)	2
101	(idPersonne=1 quotePart=100)	1
102	(idPersonne=5 quotePart=100)	1
103	(idPersonne=2 quotePart=100)	1
104	(idPersonne=2 quotePart=100)	1
201	(idPersonne=5 quotePart=100)	1
202	(idPersonne=1 quotePart=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 nuplets en une valeur par application d'une fonction d'agrégation. La fonction `count()` compte le nombre de nuplets dans chaque groupe, `max()` donne la valeur maximale d'un attribut parmi l'ensemble des nuplets du groupe, etc. La liste des fonctions d'agrégation est donnée ci-dessous:

- `count(expression)`, Compte le nombre de nuplets pour lesquels `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(niveau) as niveauMax
from     Appart
group by surface
```

sera 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.

4.3.2 Quelques exemples

Note

Vous pouvez exécuter ces requêtes sur le site : `http://deptfod.cnam.fr/bd/tp`.

Calculons la surface totale des appartements, groupés par immeuble. Décomposons: nous avons d'abord besoin du bloc « select - from - where » avec les identifants d'immeubles et les surfaces d'appartement.

```
select idImmeuble, surface
from Appart
```

On ajoute à cette requête la clause `group by` pour grouper par immeuble. On obtient alors (en phase intermédiaire) deux groupes d'appartements, un pour chaque immeuble. Il reste à appliquer une fonction d'agrégation pour ramener ces groupes à une valeur atomique.

```sql
select idImmeuble, sum(surface) as totalSurface
from Appart
group by idImmeuble
```

On pourrait aussi appliquer d'autres fonctions d'agrégation:

```sql
select idImmeuble, min(niveau) as minEtage, max(niveau) as maxEtage,
         sum(surface) as totalSurface
from Appart
group by idImmeuble
```

Revenons un moment à nos voyageurs et à leurs séjours. La requête ci-dessous doit être claire. Exécutez-la sur le site: on constate qu'un voyageur a effectué plusieurs séjours et qu'un logement a reçu plusieurs voyageurs.

```sql
select v.nom as nomVoyageur, l.nom as nomLogement
from Voyageur as v, Séjour as s, Logement as l
where v.idVoyageur = s.idVoyageur
and l.code = s.codeLogement
```

En ajoutant une clause `group by` on produit des statistiques sur le résultat de cette requête. Par exemple, en groupant sur les voyageurs

```sql
select v.nom as nomVoyageur, count(*) as 'nbSéjours'
from Voyageur as v, Séjour as s, Logement as l
where v.idVoyageur = s.idVoyageur
and l.code = s.codeLogement
group by v.idVoyageur
```

Ou en groupant sur les logements

```sql
select l.nom as nomLogement, count(*) as 'nbVoyageurs'
from Voyageur as v, Séjour as s, Logement as l
where v.idVoyageur = s.idVoyageur
and l.code = s.codeLogement
group by l.code
```

On peut aussi regrouper sur plusieurs attributs. Pour obtenir par exemple le nombre de séjours effectués par un voyageur dans un même logement.

```sql
select l.nom as nomLogement, v.nom as 'nomVoyageur', count(*) as 'nbSéjours'
from Voyageur as v, Séjour as s, Logement as l
where v.idVoyageur = s.idVoyageur
and l.code = s.codeLogement
group by l.code, v.idVoyageur
```

Moralité: à partir d'une requête SQL `select - from -- where`, aussi complexe que nécessaire, vous produisez un résultat (une table). Le `group by` permet d'effectuer des regroupements et des agrégations (simples) sur ce résultat. Il s'agit vraiment d'un complément au SQL.

4.3.3 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`. Par exemple, on peut sélectionner les appartements pour lesquels on connaît au moins deux copropriétaires.

```sql
select  idAppart, count(*) as nbProprios
from    Propriétaire
group by idAppart
having count(*) >= 2
```

On voit que la condition porte ici sur une propriété de l"/ensemble/ des nuplets du groupe et pas de chaque nuplet pris individuellement. La clause `having` est donc toujours exprimée sur le résultat de fonctions d'agrégation, par opposition avec la clause `where` qui ne peut exprimer des conditions que sur les nuplets pris un à un.

Important // La requête ci-dessus pourrait s'exprimer en utilisant l'alias pour éviter d'avoir à répeter deux fois le `count(*)` (et pour la rendre plus claire).

```
select  idAppart, count(*) as nbProprios
from    Propriétaire
group by idAppart
having nbProprios >= 2
```

Il n'est malheureusement pas sûr que l'utilisation de l'alias dans le `group by` soit acceptée dans tous les systèmes.

Quelques exemples pour conclure. Toujours sur la base des voyageurs: quels voyageurs ont effectué au moins deux séjours ?

```
select v.nom as nomVoyageur, count(*) as 'nbSéjours'
from Voyageur as v, Séjour as s, Logement as l
where v.idVoyageur = s.idVoyageur
and l.code = s.codeLogement
group by v.idVoyageur
having count(*) > 1
```

Quels logements proposent moins de deux activités.

```
select l.nom
from Logement as l, Activité as a
where l.code = a.codeLogement
group by l.code
having count(*) < 2
```

Voici enfin une requête un peu complexe (sur la base des immeubles) 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 prénom nom,
       sum(quotePart * surface / 100) as 'surfacePossédée'
from Personne as p1, Propriétaire as p2, Appart as a
where p1.id=p2.idPersonne
and  a.id=p2.idAppart
and  idImmeuble = 1
group by p1.id
order by sum(quotePart * surface / 100)
```

On obtient le résultat suivant.

nom	surfacePossédée
null	99.5000
Alice	125.0000
Alphonsine	300.5000

5 SQL, langage algébrique

Le second 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 des nuplets la relation, effectuer des *produits cartésiens* ou des *projections*, etc.

Important
nous voyons maintenant les relations comme des ensembles, au sens mathématique du terme, avec au moins deux conséquences importantes:

- il n'y a pas de doublon dans un ensemble, donc tous les opérateurs éliminent implicitement tout potentiel doublon dans le résultat ;
- un ensemble n'est pas ordonné: en aucun cas on ne peut donc s'appuyer sur l'hypothèse d'un ordre sur les nuplets.

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. C'est donc plus une approche d'informaticien que de logicien. Elle correspond moins naturellement à la manière dont on *pense* une requête. À l'origine, le langage SQL était d'ailleurs entièrement construit sur la logique mathématique, comme nous l'avons vu dans le chapitre SQL, langage déclaratif, à l'exception de l'union et de l'intersection. L'algèbre n'était utilisée que comme un moyen de décrire les opérations à effectuer pour évaluer une requête. Petit à petit, les évolutions de la norme SQL ont introduit dans le langage les opérateurs de l'algèbre. Il est maintenant possible de les retrouver tous et d'exprimer toutes les requêtes (plus ou moins facilement) avec cette approche. C'est ce que nous étudions dans ce chapitre.

Note

La base utilisée comme exemple dans ce chapitre est celle de nos intrépides voyageurs.

5.1 S1: Les opérateurs de l'algèbre

Supports complémentaires:

- Diapositives: les opérateurs de l'algèbre
- Vidéo sur les opérateurs de l'algèbre

L'algèbre se compose d'un ensemble d'opérateurs, parmi lesquels 6 sont nécessaires et suffisants et permettent de définir les autres par composition. Une propriété fondamentale de chaque opérateur est qu'il prend une ou deux relations en entrée, et produit une relation en sortie. Cette propriété (dite de *clôture*) permet de *composer* des opérateurs : 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 toutes les requêtes relationnelles à l'aide d'un petit nombre d'opérations de base.

Ces opérations sont donc:

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

Les trois 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. C'est une opération essentielle, nous lui consacrons la prochaine session.

Ces opérateurs sont maintenant présentés tour à tour.

5.1.1 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 tous les nuplets de R, dans lesquels seuls les attributs $A_1, A_2, \ldots A_k$ sont conservés. La requête suivante construit une relation avec le nom des logements et leur lieu.

$$\pi_{nom,lieu}(Logement)$$

On obtient le résultat suivant, après suppression des colonnes `id`, `capacité` et `type` :

nom	lieu
Causses	Cévennes
Génépi	Alpes
U Pinzutu	Corse
Tabriz	Bretagne

En SQL, le projection s'exprime avec le `select` suivi de la liste des attributs à projeter.

```
select nom, lieu
from Logement
```

C'est un habillage syntaxique direct de la projection.

Si on souhaite conserver tous les attributs, on peut éviter d'en énumérer la liste en la remplaçant par `*`.

```
select *
from Logement
```

Note

En algèbre cette requête est tout simplement l'identité: R

5.1.2 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 tous les logements en Corse.*

$$\sigma_{lieu='Corse'}(Logement)$$

On obtient donc le résultat :

code	nom	capacité	type	lieu
pi	U Pinzutu	10	Gîte	Corse

La sélection a pour effet de supprimer des nuplets, mais chaque nuplet garde l'ensemble de ses attributs. Il ne peut pas y avoir de problème de doublon (pourquoi?) et il ne faut donc surtout par appliquer un `distinct`.

En SQL, les critères de sélection sont exprimés par la clause `where`.

```
select *
from Logement
where lieu = 'Corse'
```

Les chaînes de caractères doivent impérativement être encadrées par des apostrophes simples, sinon le système ne verrait pas la différence avec un nom d'attribut. Ce n'est pas le cas pour les numériques, car aucun nom d'attribut ne peut commencer par un chiffre.

```
select *
from Logement
where capacité = 134
```

Note

Vous noterez que SQL appelle `select` la projection, et `where` la sélection, ce qui est pour le moins infortuné. Dans des langages modernes comme XQuery (pour les modèles basés sur XML) le, `select` est remplacé par `return`. En ce qui concerne SQL, la question a donné lieu (il y a longtemps) à des débats mais il était déjà trop tard pour changer.

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

En lui-même, le produit cartésien ne présente pas un grand intérêt puisqu'il associe aveuglément chaque nuplet de R à chaque nuplet 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.

En SQL, le produit cartésien est un opérateur `cross join` intégré à la clause `from`.

```
select *
from R cross join S
```

C'est la première fois que nous rencontrons une expression à l'intérieur du `from` en lieu et place de la simple énumération par une virgule. Il y a une logique certaine à ce choix: dans la mesure où `R cross join S` définit une nouvelle relation, la requête SQL peut être vue comme une requête sur cette seule relation, et nous sommes ramenés au cas le plus simple.

Comme illustration de ce principe, voici le résultat du produit cartésien $Logement \times Activité$ (en supprimant l'attribut `description` pour gagner de la place).

code	nom	capacité	type	lieu	codeLogement	codeActivité
ca	Causses	45	Auberge	Cévennes	ca	Randonnée
ge	Génépi	134	Hôtel	Alpes	ca	Randonnée
pi	U Pinzutu	10	Gîte	Corse	ca	Randonnée
ta	Tabriz	34	Hôtel	Bretagne	ca	Randonnée
ca	Causses	45	Auberge	Cévennes	ge	Piscine
ge	Génépi	134	Hôtel	Alpes	ge	Piscine
pi	U Pinzutu	10	Gîte	Corse	ge	Piscine
ta	Tabriz	34	Hôtel	Bretagne	ge	Piscine
ca	Causses	45	Auberge	Cévennes	ge	Ski
ge	Génépi	134	Hôtel	Alpes	ge	Ski
pi	U Pinzutu	10	Gîte	Corse	ge	Ski
ta	Tabriz	34	Hôtel	Bretagne	ge	Ski
ca	Causses	45	Auberge	Cévennes	pi	Plongée
ge	Génépi	134	Hôtel	Alpes	pi	Plongée
pi	U Pinzutu	10	Gîte	Corse	pi	Plongée
ta	Tabriz	34	Hôtel	Bretagne	pi	Plongée
ca	Causses	45	Auberge	Cévennes	pi	Voile
ge	Génépi	134	Hôtel	Alpes	pi	Voile
pi	U Pinzutu	10	Gîte	Corse	pi	Voile
ta	Tabriz	34	Hôtel	Bretagne	pi	Voile

C'est une relation (tout est relation en relationnel) et on peut bien imaginer interroger cette relation comme n'importe quelle autre. C'est exactement ce que fait la requête SQL suivante.

```
select *
from Logement cross join Activité
```

Jusqu'à présent, le `from` ne contenait que des relations « *basées* » (c'es-à-dire stockées dans la base). Maintenant, on a placé une relation *calculée*. Le principe reste le même. Rappelons que l'algèbre est un langage *clos*: il s'applique à des relations et produit une relation en sortie. Il est donc possible d'appliquer à nouveau des opérateurs à cette relation-résultat. C'est ainsi que l'on construit des expressions, comme nous allons le voir dans la session suivante. Nous retrouverons une autre application de cette propriété extrêmement utile quand nous étudierons les vues.

5.1.4 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 relation résultat en donnant un nom distinct à chaque attribut.

Voici par exemple une relation 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 relation 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	o	p
x	y	m	n
x	y	o	p

Cette convention pose quelques problèmes quand on crée des expressions complexes. Il existe une seconde possibilité, plus générale, 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

En SQL, le renommage est obtenu avec le mot-clé `as`. Il peut s'appliquer soit à la relation, soit aux attributs (ou bien même aux deux). Le résultat suivant est donc obtenu avec la requête:

```
select *
from R as R1 cross join R as R2
```

On obtient une relation de schéma `(R1.A, R1.B, R1.A, R2.B)`, avec des noms d'attribut qui ne sont en principe pas acceptés par la norme SQL. Il reste à spécifier ces noms en ajoutant dans `as` dans la clause de projection.

```
select R1.a as premier_a, R1.b as premier_b, R2.a as second_a, R2.b as second_b
from R as R1 cross R as R2
```

Ce qui donnera donc le résultat:

premier_a	premier_b	second_a	second_b
a	b	a	b
a	b	x	y
x	y	a	b
x	y	x	y

Sur notre schéma, le renommage s'impose par exemple si on effectue le produit cartésien entre `Voyageur` et `Séjour` car l'attribut `idVoyageur` apparaît dans les deux tables. Essayez la requête:

```
select Voyageur.idVoyageur, Séjour.idVoyageur
from Voyageur cross join Séjour
```

Elle vous renverra une erreur comme *Encountered duplicate field name: "idVoyageur"*. Il faut nommer explicitement les attributs pour lever l'ambiguité.

```
select Voyageur.idVoyageur as idV1, Séjour.idVoyageur as idV2
from Voyageur cross join Séjour
```

5.1.5 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.

L'union est un des opérateurs qui existe dans SQL depuis l'origine. La requête suivante effectue l'union des lieux de la table `Logement` et des régions de la table `Voyageur`. Pour unifier les schémas, on a projeté sur cet unique attribut, et on a effectué un renommage.

```
select lieu from Logement
  union
select région as lieu from Voyageur
```

On obtient le résultat suivant.

lieu
Cévennes
Alpes
Corse
Bretagne
Auvergne
Tibet

Notez que certains noms comme « Corse » apparaîssent deux fois: vous savez maintenant comment éliminer les doublons avec SQL.

5.1.6 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

En SQL, la différence est obtenue avec `except`.

```
select A, B from R
   except
select C as A, D as B from S
```

La différence est le seul opérateur algébrique qui permet d'exprimer des requêtes comportant une négation (on veut « *rejeter* » quelque chose, on « *ne veut pas* » des nuplets ayant telle propriété). La contrainte d'identité des schémas rend cet opérateur très peu pratique à utiliser, et on lui préfère le plus souvent la construction logique du SQL « déclaratif », `not exists`.

Note

L'opérateur `except` n'est même pas proposé par certains systèmes comme MYSQL.

5.2 S2: la jointure

Supports complémentaires:

- Diapositives: la jointure algébrique
- Vidéo sur la jointure algébrique

Toutes les requêtes exprimables avec l'algèbre relationnelle peuvent se construire avec les 6 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.

5.2.1 L'opérateur ⋈

Afin de comprendre l'intérêt de cet opérateur, regardons le produit cartésien Logement×Activité, dont le résultat est rappelé ci-dessous.

code	nom	capacité	type	lieu	codeLogement	codeActivité
ca	Causses	45	Auberge	Cévennes	ca	Randonnée
ge	Génépi	134	Hôtel	Alpes	ca	Randonnée
pi	U Pinzutu	10	Gîte	Corse	ca	Randonnée
ta	Tabriz	34	Hôtel	Bretagne	ca	Randonnée
ca	Causses	45	Auberge	Cévennes	ge	Piscine
ge	Génépi	134	Hôtel	Alpes	ge	Piscine
pi	U Pinzutu	10	Gîte	Corse	ge	Piscine
ta	Tabriz	34	Hôtel	Bretagne	ge	Piscine
ca	Causses	45	Auberge	Cévennes	ge	Ski
ge	Génépi	134	Hôtel	Alpes	ge	Ski
pi	U Pinzutu	10	Gîte	Corse	ge	Ski
ta	Tabriz	34	Hôtel	Bretagne	ge	Ski
ca	Causses	45	Auberge	Cévennes	pi	Plongée
ge	Génépi	134	Hôtel	Alpes	pi	Plongée
pi	U Pinzutu	10	Gîte	Corse	pi	Plongée
ta	Tabriz	34	Hôtel	Bretagne	pi	Plongée
ca	Causses	45	Auberge	Cévennes	pi	Voile
ge	Génépi	134	Hôtel	Alpes	pi	Voile
pi	U Pinzutu	10	Gîte	Corse	pi	Voile
ta	Tabriz	34	Hôtel	Bretagne	pi	Voile

Si vous regardez attentivement cette relation, vous noterez que le résultat comprend manifestement un grand nombre de nuplets qui ne nous intéressent pas. C'est le cas de toutes les lignes pour lesquelles le `code` (provenant de la table `Logement`) et le `codeLogement` (provenant de la table `Activité`) sont distincts. Cela ne présente pas beaucoup de sens (à priori) de rapprocher des informations sur l'hôtel Génépi, dans les Alpes, avec l'activité de plongée en Corse.

Note

Il est bien sûr arbitraire de dire qu'un résultat « n'a pas de sens » ou « ne présente aucun intérêt ». Nous nous plaçons ici dans un contexte où l'on cherche à reconstruire une information sur certaines entités du monde réel, dont la description a été distribuée dans plusieurs tables par la normalisation. C'est l'utilisation sans doute la plus courante de SQL.

Si, en revanche, on considère le produit cartésien comme un *résultat intermédiaire*, on voit qu'il permet d'associer des nuplets initialement répartis dans des tables distinctes. Sur notre exemple, on rapproche les informations générales sur un logement et la liste des activités de ce logement.

La sélection qui effectue une rapprochement pertinent est celle qui ne conserve que les nuplets partageant la même valeur pour les attributs `code` et `codeLogement`, soit:

$$\sigma_{code=codeLogement}(\text{Logement} \times \text{Activité})$$

Prenez bien le temps de méditer cette opération de sélection: nous ne voulons conserver que les nuplets de Logement × Activité pour lesquelles l'identifiant du logement (provenant de `Logement`) 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 nuplets qui ont un sens: chacune contient des informations sur un logement et sur une activité dans ce *même* logement.

On obtient le résultat ci-dessous.

code	nom	capacité	type	lieu	codeLogement	codeActivité
ca	Causses	45	Auberge	Cévennes	ca	Randonnée
ge	Génépi	134	Hôtel	Alpes	ge	Piscine
ge	Génépi	134	Hôtel	Alpes	ge	Ski
pi	U Pinzutu	10	Gîte	Corse	pi	Plongée
pi	U Pinzutu	10	Gîte	Corse	pi	Voile

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

$$\text{Logement} \bowtie_{\text{code}=\text{codeLogement}} \text{Activité}$$

La jointure consiste donc à rapprocher les nuplets 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) l'identifiant de l'une des relations et (2) une référence à cet identifiant dans l'autre relation. Dans l'exemple ci-dessus, c'est le cas pour `code` (identifiant de *Logement*) et `codeLogement` (référence dans *Activité*).

Note

Le logement Tabriz, 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 dans le chapitre final sur SQL que ce dernier propose une variante, la *jointure externe*, qui permet de la contourner.

La notation de la jointure, $R \bowtie_F S$, est un racourci pour $\sigma_F(R \times S)$.

Note

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.

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

Initialement, SQL ne proposait pour effectuer la jointure que la version déclarative.

```
select *
from Logement as l, Activité as a
where l.code=a.codeLogement
```

En 1992, la révision de la norme a introduit l'opérateur algébrique qui, comme le produit cartésien, et pour les mêmes raisons, prend place dans le `from`.

```
select *
from Logement join Activité on (code=codeLogement)
```

Il s'agit donc d'une manière alternative *d'exprimer* une jointure. Laquelle est la meilleure? Aucune, puisque toutes les deux ne sont que des spécifications, et n'imposent en aucun cas au système une méthode particulière d'exécution. Il est d'ailleurs exclu pour un système d'appliquer aveuglément la définition de la jointure et d'effectuer un produit cartésien, puis une sélection, car il existe des algorithmes d'évaluation bien plus efficaces.

5.2.2 Résolution des ambiguités

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, ou bien on renomme des attributs avant d'effectuer la jointure.

Supposons que l'on veuille obtenir les voyageurs et les séjours qu'ils ont effectués. La jointure s'exprime en principe comme suit:

```
select *
from Voyageur join Séjour on (idVoyageur=idVoyageur)
```

Le système renvoie une erreur: La clause de jointure `on (idVoyageur=idVoyageur)` est clairement ambigüe. Pour MySQL, le message est par exemple *Column "idVoyageur" in on clause is ambiguous*. Nouvelle tentative:

```
select *
from Voyageur join Séjour on (Voyageur.idVoyageur=Séjour.idVoyageur)
```

Nouveau message d'erreur (cette fois, sous MySQL: *Encountered duplicate field name: "idVoyageur"*). La liste des noms d'attribut dans le nuplet-résultat obtenu avec `select *` comprend encore deux fois `idVoyageur`.

Première solution: on renomme les attributs du nuplet résultat. Cela suppose d'énumérer tous les attributs.

```
select V.idVoyageur as idV1, V.nom, S.idVoyageur as idV2, début, fin
from Voyageur as V join Séjour as S on (V.idVoyageur=S.idVoyageur)
```

Cette première solution consiste à effectuer un renommage *après* la jointure. Une autre solution est d'effectuer le renommage *avant* la jointure.

```
select *
from (select idVoyageur as idV1, nom from Voyageur) as V
            join
     (select idVoyageur as idV2, début, fin from Séjour) as S
         on (V.idV1=S.idV2)
```

En algèbre, la requête ci-dessus correspond à l'expression suivante:

$$(\rho_{idVoyageur \to idV1}(\pi_{idVoyageur,nom} Voyageur) \bowtie_{idV1=idV2} \rho_{idVoyageur \to idV2}(\pi_{idVoyageur,début,fin} Séjour))$$

On voit que le `from` commence à contenir des expressions de plus en plus complexes. Dans ses premières versions, SQL ne permettait pas des constructions algébriques dans le `from`, ce qui avait l'avantage d'éviter des constructions qui ressemblent de plus en plus à de la programmation. Rappelons qu'il existe une syntaxe alternative à la requête ci-dessus, dans la forme déclarative de SQL étudiée au chapitre précédent.

```
select V.idVoyageur as idV1, V.nom, S.idVoyageur as idV2, début, fin
from Voyageur as V, Séjour as S
where V.idVoyageur= S.idVoyageur
```

Bref, vous commencez à avoir l'embarras du choix.

La jointure dite « naturelle »

Il reste à vrai dire, avec SQL, un troisième choix, la jointure dite « naturelle ». Elle s'applique uniquement quand les attributs de jointure ont des noms identiques dans les deux tables. C'est le cas ici, (l'attribut de jointure est `idVoyageur`, que ce soit dans `Logement` ou dans `Séjour`). La jointure naturelle s'effectue alors automatiquement sur ces attributs communs, et ne conserve que l'un des attributs dans le résultat, ce qui élimine l'ambiguité. La syntaxe devient alors très simple.

```
select *
from Voyageur as V natural join Séjour
```

Si les attributs de jointures sont nommés différemment, la jointure naturelle devient plus délicate à utiliser puisqu'il faut au préalable effectuer des renommages pour faire coïncider les noms des attributs à comparer.

À partir de là, vous savez comment effectuer plusieurs jointures. Un exemple devrait suffire: supposons que l'on veuille les noms des voyageurs et les noms des logements qu'ils ont visités. La requête algébrique devient un peu compliquée. On va s'autoriser une construction en plusieurs étapes.

Tout d'abord on effectue un renommage sur la table `Voyageur` pour éviter les futures ambiguités.

$$V2 := \rho_{idVoyageur \to idV, nom \to nomVoyageur}(Voyageur)$$

Opération semblable sur les logements.

$$L2 := \rho_{nom \to nomLogement}(Logement)$$

Et finalement, voici la requête algébrique complète, utilisant `V2` et `L2`.

$$\pi_{nomVoyageur, nomLogement}(\text{L2}) \bowtie_{\text{code=codeLogement}} \text{Séjour} \bowtie_{\text{idVoyageur=idV}} \text{V2}$$

En SQL, il faut tout écrire avec une seule requête. Allons-y

```
select nomVoyageur, nomLogement
from ( (select idVoyageur as idV, nom as nomVoyageur from Voyageur) as V
              join
            Séjour as S on idV=idVoyageur)
           join
         (select code, nom as nomLogement from Logement) as L
           on codeLogement = code
```

Ce n'est pas très lisible... Pour comparaison, la version déclarative de ces jointures.

```
select V.nom as nomVoyageur, L.nom as nomLogement
from   Voyageur as V, Séjour as S, Logement as L
where  V.idVoyageur = S.idVoyageur
and    S.codelogement = L. code
```

À vous de voir quel style (ou mélange des styles) vous souhaitez adopter.

5.3 S3: Expressions algébriques

Supports complémentaires:

- Diapositives: expressions algébriques
- Vidéo sur les expressions algébriques

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.

Note

Les expressions sont seulement données dans la forme concise de l'algèbre. La syntaxe SQL

équivalente est à faire à titre d'exercices.

5.3.1 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 nuplets satisfaisant *un* critère de sélection, par exemple : « les logements de type "Hôtel" ». Maintenant supposons que l'on veuille retrouver les hôtels dont la capacité est supérieure à 100. On peut exprimer cette requête par une composition :

$$\sigma_{capacit\acute{e}>100}(\sigma_{type='H\hat{o}tel'}(Logement))$$

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".

$$\sigma_{capacit\acute{e}>100 \wedge type='H\hat{o}tel'}(Logement)$$

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 logements qui sont en Corse, *ou* dont la capacité est supérieure à 100.

$$\sigma_{capacit\acute{e}>100}(Logement) \cup \sigma_{lieu='Corse'}(Logement)$$

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

$$\sigma_{capacit\acute{e}>100\ \vee\ lieu='Corse'}(Logement)$$

Enfin la *différence* permet d'exprimer la *négation* et « d'éliminer » des nuplets. Par exemple, voici la requête qui sélectionne les logements dont la capacité est supérieure à 200 mais qui ne sont *pas* en Corse. .

$$\sigma_{capacit\acute{e}>100}(Logement) - \sigma_{lieu='Corse'}(Logement)$$

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

$$\sigma_{capacit\acute{e}>100 \wedge lieu\neq'Corse'}(Logement)$$

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, l'exemple précédent étant une exception. 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.

5.3.2 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 logements en Corse : $\pi_{nom}(\sigma_{lieu='Corse'}(Logement))$

- Code des logements où l'on pratique la voile. $\pi_{codeLogement}(\sigma_{codeActivit='Voile'}(Activit))$
- Nom et prénom des clients corses $\pi_{nom,prnom}(\sigma_{rgion='Corse'}(Voyageur))$

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 relations. 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 le lieu des logements où l'on pratique la voile ». Une analyse très simple suffit pour constater que l'on a besoin des attributs `lieu` et `nom` qui apparaîssent dans la relation `Logement`, et de `codeActivité` qui apparaît dans `Activité`.

Donc il faut faire une jointure, de manière à rapprocher les nuplets de `Logement` 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 `Logement` et *Activité*. Elle s'effectue donc par appariement de la identifiant d'une part (dans `Logement`), de la référence à cet identifiant d'autre part.

$$\pi_{nom,lieu}(Logement \bowtie_{code=codeLogement} (\sigma_{codeActivité='Voile'}(\text{Activit})))$$

En pratique, la grande majorité des opérations de jointure s'effectue sur des attributs qui sont des identifiants dans une relations, et une référence à cet identifiant 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 un logement et ses activités, ou un logement et ses clients), mais qui ont été réparties dans plusieurs relations au moment de la conception de la base.

Voici quelques autres exemples qui illustrent cet état de fait :

- Nom des clients qui sont allés à Tabriz (en supposant connu le code, `ta`, de cet hôtel) :

$$\pi_{nom}(\text{Voyageur} \bowtie_{idVoyageur=idVoyageur} \sigma_{codeLogement='ta'}(\text{Séjour}))$$

- Quels lieux a visité le client 30 :

$$\pi_{lieu}(\sigma_{idVoyageur=30}(\text{Sjour}) \bowtie_{codeLogement=code} (\text{Logement}))$$

- Nom des clients qui ont eu l'occasion de faire de la voile :

$$\pi_{nom}(\texttt{Voyageur} \bowtie_{idVoyageur=idVoyageur} (\text{Séjour} \bowtie_{\text{codeLogement=codeLogement}} \sigma_{\text{codeActivité='Voile'}}(\text{Activité})))$$

Note

Pour simplifier un peu l'expression, on a considéré ci-dessus que l'ambiguité sur l'attribut de jointure `idVoyageur` était effacée par la projection finale sur `nom`. En toute rigueur, la relation obtenue par

$$\texttt{Voyageur} \bowtie_{idVoyageur=idVoyageur} (\texttt{Séjour} \bowtie_{codeLogement=codeLogement} \sigma_{codeActivité='Voile'}(\texttt{Activité}))$$

comporte des noms d'attributs doublés auxquels il faudrait appliquer un renommage.

La dernière requête comprend deux jointures, portant à chaque fois sur des identifiants. Encore une fois ce sont les identifiants et leurs références 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 lieu de résidence, ainsi que le nom de ce lieu.

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

$$\pi_{nom,lieu}(\text{Voyageur} \bowtie_{idVoyageur=idVoyageur \wedge région=lieu} (\text{Séjour} \bowtie_{codeLogement=code} \text{Logement}))$$

Les jointures avec la relation `Séjour` se font sur les couples (identifiant, référence), mais on a en plus un critère de rapprochement relatif à l'attribut `lieu` de `Voyageur` et de `Logement`.

5.3.3 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.

Illustration concrète sur la base de données avec la requête suivante : quels sont les codes des logements qui *ne* proposent *pas* de voile ?

$$\pi_{code}(\text{Logement}) - \pi_{codeLogement}(\sigma_{codeActivité='Voile'}(\text{Activité}))$$

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 qui sélectionne tous les O *qui satisfont* p.
- Finalement, faire $A - B$.

Les requêtes A et B peuvent bien entendu être arbitrairement complexes et mettre en œuvre des jointures, des sélections, etc. La seule contrainte est que le résultat de A et de B comprenne le même nombre d'attributs (et, en théorie, les mêmes noms, mais on peut s'affranchir de cette contrainte).

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 logements qui ne proposent pas de voile

$$\pi_{codeLogement}(\sigma_{codeActivité \neq 'Voile'}(\text{Activité}))$$

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 logement.

$$\pi_{région}(\text{Voyageur}) - \pi_{région}(\rho_{lieu \to région}(\text{Logement}))$$

- Identifiant des logements qui n'ont pas reçu de client tibétain.

  $$\pi_{code}(\text{Logement}) - \pi_{codeLogement}(\text{Séjour} \bowtie_{idVoyageur=idVoyageur} \sigma_{région='Tibet'}(\text{Voyageur}))$$

- Id des clients qui ne sont pas allés en Corse.

  $$\pi_{idVoyageur}(\text{Voyageur}) - \pi_{idVoyageur}(\sigma_{lieu='Corse'}(\text{Logement}) \bowtie_{code=codeLogement} \text{Séjour})$$

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

5.3.4 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 logements où ils ne sont pas allés. En d'autres termes, parmi toutes les associations Voyageur/Logement 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 $-$.

$$(\pi_{idVoyageur}(\text{Voyageur}) \times \pi_{code}(\text{Logement})) - \pi_{idVoyageur,codeLogement}(\text{Séjour})$$

5.3.5 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. À 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 : quels sont les clients dont *tous* les séjours ont eu lieu en Corse? On l'exprime également par "quels sont clients pour lesquels *il n'existe pas* de séjour dans un lieu qui soit différent de la Corse. Ce qui donne l'expression suivante :

$$\pi_{idVoyageur}(\text{Séjour}) - \pi_{idVoyageur}(\sigma_{lieu\neq'Corse'}(\text{Séjour}))$$

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 *tous* les logements.

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

$$\pi_{idVoyageur}(\text{Voyageur}) - \pi_{idVoyageur}((\pi_{idVoyageur}(\text{Voyageur}) \times \pi_{code}(\text{Logement})) - \pi_{idVoyageur,idLogement}(\text{Séjour}))$$

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

$$\pi_{idVoyageur}(\text{Voyageur}) \times \pi_{code}(\text{Logement})) - \pi_{idVoyageur,idLogement}(\text{Séjour})$$

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

$$\pi_{idVoyageur}(\text{Voyageur}) - B$$

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