Introduction au Système de Gestion de Base de Données et aux Base de Données

Formation
« Gestion des données scientifiques : stockage et consultation en utilisant des bases de données »
24 au 27 /06/08

Dernière modif. : 17/06/2008	**Version :** 1.0	**Pages:** 47

INTRODUCTION

Aujourd'hui, la disponibilité de systèmes de gestion de base de données fiables permet aux organisations de toutes tailles de gérer des données efficacement, de déployer des applications utilisant ces données et de les stocker. Les bases de données sont actuellement au cœur du système d'information des entreprises.

Les bases de données relationnelles constituent l'objet de ce cours. Ces bases sont conçues suivant le modèle relationnel, dont les fondations théoriques sont solides, et manipulées en utilisant l'algèbre relationnelle. Il s'agit, à ce jour, de la méthode la plus courante pour organiser et accéder à des ensembles de données.

Cependant, il est difficile de modéliser un domaine directement sous une forme base de données relationnelle. Une modélisation intermédiaire est généralement indispensable. Le modèle entités-associations ou le diagramme des classes permettent une description naturelle du monde réel à partir des concepts d'entité et d'association ou classes/association.

Après une courte introduction (chapitre 1), nous présenterons le modèle relationnel, le passage du modèle conceptuel au modèle relationnel et enfin l'algèbre relationnelle (chapitre 2).

Le chapitre 3 est entièrement consacré au langage SQL (Structured Query Language) qui peut être considéré comme le langage d'accès normalisé aux bases de données relationnelles. Ce langage est supporté par la plupart des systèmes de gestion de bases de données commerciaux (comme Oracle) et du domaine libre (comme PostgreSQL).

Nous détaillons dans ce chapitre les instructions du langage de définition de données et celles du langage de manipulation de données.

Ce document a été réalisé à partir du support du cours « Base de Données et langage SQL » dispensé aux étudiants du département d'informatique de l'institut universitaire de technologie de Villetaneuse (http://laurent-audibert.developpez.com/Cours-BD/).

I - Introduction aux bases de données

1.1 - Qu'est ce qu'une base de données

Une base de données (BD) *est un ensemble structuré et organisé permettant le stockage de grandes quantités d'informations afin d'en faciliter l'exploitation (ajout, mise à jour, recherche de données).*

1.2 – Système de Gestion de Base de Données

1.2.1 Définition et principes de fonctionnement

Un système de gestion de base de données (SGBD) *est un ensemble de programmes qui permet la gestion et l'accès à une base de données.* Il héberge généralement plusieurs bases de données, qui sont destinées à des logiciels ou des thématiques différentes.

On distingue couramment les SGBD classiques, dits **SGBD relationnels** (SGBD-R), des **SGBD orientés objet** (SGBD-O). En fait, un SGBD est caractérisé par **le modèle de description des données** qu'il supporte (relationnel, objet etc.). Les données sont décrites sous la forme de ce modèle, grâce à un Langage de Description des Données (LDD). Cette description est appelée schéma.
Une fois la base de données spécifiée, on peut y insérer des données, les récupérer, les modifier et les détruire. Les données peuvent être manipulées non seulement par un Langage spécifique de Manipulation des Données (LMD) mais aussi par des langages de programmation classiques.

Actuellement, la plupart des SGBD fonctionnent selon un mode client/serveur. Le serveur (sous entendu la machine qui stocke les données) reçoit des requêtes de plusieurs clients et ceci de manière concurrente. Le serveur analyse la requête, la traite et retourne le résultat au client.

Quelque soit le modèle, un des problèmes fondamentaux à prendre en compte est la cohérence des données. Par exemple, dans un environnement où plusieurs utilisateurs peuvent accéder concurremment à une colonne d'une table par exemple pour la lire ou pour l'écrire, il faut s'accorder sur la politique d'écriture. Cette politique peut être : les lectures concurrentes sont autorisées mais dès qu'il y a une écriture dans une colonne, l'ensemble de la colonne est envoyée aux autres utilisateurs l'ayant lue pour qu'elle soit rafraîchie.

1.2.2 Objectifs

Les principaux objectifs fixés aux SGBD afin de résoudre les problèmes causés par une gestion sous forme de fichiers à plat sont les suivants :

- **Indépendance physique** : La façon dont les données sont définies doit être indépendante des structures de stockage utilisées.
- **Indépendance logique** : Un même ensemble de données peut être vu différemment par des utilisateurs différents. Toutes ces visions personnelles des données doivent être intégrées dans une vision globale.
- **Accès aux données** : L'accès aux données se fait par l'intermédiaire d'un Langage de Manipulation de Données (LMD). Il est crucial que ce langage permette d'obtenir des réponses aux requêtes en un temps « raisonnable ». Le LMD doit donc être optimisé, minimiser le nombre d'accès disques, et tout cela de façon totalement transparente pour l'utilisateur.
- **Administration centralisée des données (intégration)** : Toutes les données doivent être centralisées dans un réservoir unique commun à toutes les applications. En effet, des visions

différentes des données (entre autres) se résolvent plus facilement si les données sont administrées de façon centralisée.

- **Non redondance des données** : Afin d'éviter les problèmes lors des mises à jour, chaque donnée ne doit être présente qu'une seule fois dans la base.
- **Cohérence des données** : Les données sont soumises à un certain nombre de contraintes d'intégrité qui définissent un état cohérent de la base. Elles doivent pouvoir être exprimées simplement et vérifiées automatiquement à chaque insertion, modification ou suppression des données. Les contraintes d'intégrité sont décrites dans le Langage de Description de Données (LDD).
- **Partage des données** : Il s'agit de permettre à plusieurs utilisateurs d'accéder aux mêmes données au même moment de manière transparente. Si ce problème est simple à résoudre quand il s'agit uniquement d'interrogations, cela ne l'est plus quand il s'agit de modifications dans un contexte multi-utilisateurs car il faut : permettre à deux (ou plus) utilisateurs de modifier la même donnée « en même temps » et assurer un résultat d'interrogation cohérent pour un utilisateur consultant une table pendant qu'un autre la modifie.
- **Sécurité des données** : Les données doivent pouvoir être protégées contre les accès non autorisés. Pour cela, il faut pouvoir associer à chaque utilisateur des droits d'accès aux données.
- **Résistance aux pannes** : Que se passe-t-il si une panne survient au milieu d'une modification, si certains fichiers contenant les données deviennent illisibles ? Il faut pouvoir récupérer une base dans un état « sain ». Ainsi, après une panne intervenant au milieu d'une modification deux solutions sont possibles : soit récupérer les données dans l'état dans lequel elles étaient avant la modification, soit terminer l'opération interrompue.

1.2.3 Niveaux de description des données ANSI/SPARC

Pour atteindre certains de ces objectifs (surtout les deux premiers), trois niveaux de description des données ont été définis par la norme ANSI/SPARC :

- **Le niveau externe** correspond à la perception de tout ou partie de la base par un groupe donné d'utilisateurs, indépendamment des autres. On appelle cette description le schéma externe ou vue. Il peut exister plusieurs schémas externes représentant différentes vues sur la base de données avec des possibilités de recouvrement. Le niveau externe assure l'analyse et l'interprétation des requêtes en primitives de plus bas niveau et se charge également de convertir éventuellement les données brutes, issues de la réponse à la requête, dans un format souhaité par l'utilisateur.
- **Le niveau conceptuel** décrit la structure de toutes les données de la base, leurs propriétés (i.e. les relations qui existent entre elles : leur sémantique inhérente), sans se soucier de l'implémentation physique ni de la façon dont chaque groupe de travail voudra s'en servir. Dans le cas des SGBD relationnels, il s'agit d'une vision tabulaire où la sémantique de l'information est exprimée en utilisant les concepts de relation, attributs et de contraintes d'intégrité. On appelle cette description le schéma conceptuel.
- **Le niveau interne ou physique** s'appuie sur un système de gestion de fichiers pour définir la politique de stockage ainsi que le placement des données. Le niveau physique est donc responsable du choix de l'organisation physique des fichiers ainsi que de l'utilisation de telle ou telle méthode

1.3 Quelques SGBD connus et utilisés

Il existe de nombreux systèmes de gestion de bases de données, en voici une liste non exhaustive :

ACCESS : plate-forme Windows, mono-poste, licence commerciale
SQL SERVER : plate-forme Windows, mode client/serveur, licence commerciale
ORACLE : plate-formes Windows et Linux, mode client/serveur, licence commerciale
SYBASE : plate-formes Windows et Linux, mode client/serveur, licence commerciale
POSTGRESQL : plate-formes Windows et Linux, mode client/serveur, licence libre
MYSQL : plate-formes Windows et Linux, mode client/serveur, licence libre
Comparatif de SGBD : http://fadace.developpez.com/sgbdcmp/

1.4 PostgreSQL

Les systèmes traditionnels de gestion de bases de données relationnelles (SGBDR) offrent un modèle de données composé d'une collection de relations contenant des attributs relevant chacun d'un type spécifique.
PostgreSQL apporte une puissance additionnelle substantielle en incorporant les quatre concepts de base suivants afin que les utilisateurs puissent facilement étendre le système : classes, héritage, types, fonctions. D'autres fonctionnalités accroissent la puissance et la souplesse : contraintes, déclencheurs, règles, intégrité des transactions.
Ces fonctionnalités placent PostgreSQL dans la catégorie des bases de données relationnelles objet mais bien que PostgreSQL possède certaines fonctionnalités orientées objet, il appartient avant tout au monde des SGBDR. C'est essentiellement l'aspect SGBDR de PostgreSQL qui sera abordé dans ce cours.
L'une des principales qualités de PostgreSQL est d'être un logiciel libre, c'est-à-dire gratuit et dont les sources sont disponibles. Il est possible de l'installer sur les systèmes Unix/Linux et Win32.

PostgreSQL fonctionne selon une architecture client/serveur, il est ainsi constitué :

– d'une partie serveur, c'est-à-dire une application fonctionnant sur la machine hébergeant la base de données (le serveur de bas es de données) capable de traiter les requêtes des clients ;
– d'une partie client (psql) devant être installée sur toutes les machines nécessitant d'accéder au serveur de base de données (un client peut éventuellement fonctionner sur le serveur lui-même).

II - SGBD relationnel

2.1 - Introduction au modèle relationnel

Edgar Frank Codd, chercheur chez IBM, étudiait à la fin des années 1960 de nouvelles méthodes pour gérer de grandes quantités de données car les modèles et les logiciels de l'époque ne le satisfaisaient pas. Mathématicien de formation, il était persuadé qu'il pourrait utiliser des branches spécifiques des mathématiques (la théorie des ensembles et la logique des prédicats du premier ordre) pour résoudre des difficultés telles que la redondance des données, l'intégrité des données ou l'indépendance de la structure de la base de données avec sa mise en œuvre physique.
En 1970, Codd publia un article où il proposait de stocker des données hétérogènes dans des tables, permettant d'établir des relations entre elles.
Un premier prototype de Système de gestion de bases de données relationnelles a été construit dans les laboratoires d'IBM. Depuis les années 80, cette technologie a mûri et a été adoptée par l'industrie. En 1987, le langage SQL, qui étend l'algèbre relationnelle, a été standardisé.

Dans ce modèle, les données sont représentées par des tables, sans préjuger de la façon dont les informations sont stockées dans la machine. Les tables constituent donc la structure logique du modèle relationnel. Au niveau physique, le système est libre d'utiliser n'importe quelle technique de stockage dès lors qu'il est possible de relier ces structures à des tables au niveau logique. Les tables ne représentent donc qu'une abstraction de l'enregistrement physique des données en mémoire.

Le succès du modèle relationnel auprès des chercheurs, concepteurs et utilisateurs est dû à la puissance et à la simplicité de ses concepts. En outre, contrairement à certains autres modèles, il repose sur des bases théoriques solides, notamment la théorie des ensembles et la logique des prédicats du premier ordre.

Les objectifs du modèle relationnel sont :
- proposer des schémas de données faciles à utiliser ;
- améliorer l'indépendance logique et physique;
- mettre à la disposition des utilisateurs des langages de haut niveau ;
- optimiser les accès à la base de données ;
- améliorer l'intégrité et la confidentialité ;
- fournir une approche méthodologique dans la construction des schémas.

De façon informelle, on peut définir le modèle relationnel de la manière suivante :
- les données sont organisées sous forme de tables à deux dimensions, encore appelées relations, dont les lignes sont appelées n-uplet ou tuple en anglais ;
- les données sont manipulées par des opérateurs de l'algèbre relationnelle ;
- l'état cohérent de la base est défini par un ensemble de contraintes d'intégrité.

2.2 Éléments du modèle relationnel

Un attribut *est un identificateur (un nom) décrivant une information stockée dans une base.*
Ex : l'âge d'une personne, le nom d'une personne, le numéro de sécurité sociale.

Le domaine d'un attribut *est l'ensemble, fini ou infini, de ses valeurs possibles.*
Par exemple, l'attribut numéro de sécurité sociale a pour domaine l'ensemble des combinaisons de quinze chiffres et nom a pour domaine l'ensemble des combinaisons de lettres (chaîne de caractère).

Une relation *est un sous-ensemble du produit cartésien de n domaines d'attributs (n > 0).* Elle est représentée sous la forme d'un tableau à deux dimensions dans lequel les n attributs correspondent aux titres des n colonnes.

Un schéma de relation *précise le nom de la relation ainsi que la liste des attributs avec leurs domaines.*

N° Sécu	Nom	Prénom
354338532195874	Durand	Caroline
345353545435811	Dubois	Jacques
173354684513546	Dupont	Lisa
973564213535435	Dubois	Rose-Marie

Exemple du schéma de relation Personne(N° sécu : Entier, Nom : Chaîne, Prénom : Chaîne)

Le degré d'une relation *est son nombre d'attributs.*

Une occurrence, *ou* ***n-uplets,*** *ou* ***tuples****, est un élément de l'ensemble figuré par une relation.* Autrement dit, une occurrence est une ligne du tableau qui représente la relation.

La ***cardinalité d'une relation*** *est son nombre d'occurrences.*

Une clé candidate *d'une relation est un ensemble minimal des attributs de la relation dont les valeurs identifient à coup sûr une occurrence.*

La valeur d'une clé candidate est donc distincte pour toutes les tuples de la relation. La notion de clé candidate est essentielle dans le modèle relationnel.

Toute relation a au moins une clé candidate et peut en avoir plusieurs. Ainsi, il ne peut jamais y avoir deux tuples identiques au sein d'une relation. Les clés candidates d'une relation n'ont pas forcément le même nombre d'attributs. Une clé candidate peut être formée d'un attribut arbitraire, utilisé à cette seule fin.

La clé primaire d'une relation est une de ses clés candidates. Pour signaler la clé primaire, ses attributs sont généralement soulignés.
Une clé étrangère dans une relation est formée d'un ou plusieurs attributs qui constituent une clé primaire dans une autre relation.

Un schéma relationnel est constitué par l'ensemble des schémas de relation.

Une base de données relationnelle est constituée par l'ensemble des n-uplets des différentes relations du schéma relationnel

2.3 Algèbre relationnelle

2.3.1 Introduction

L'algèbre relationnelle est un support mathématique cohérent sur lequel repose le modèle relationnel. L'objet de cette section est d'aborder l'algèbre relationnelle dans le but de décrire les opérations qu'il est possible d'appliquer sur des relations pour produire de nouvelles relations. L'approche suivie est donc plus opérationnelle que mathématique.

On peut distinguer trois familles d'opérateurs relationnels :

- **Les opérateurs unaires (Sélection, Projection)** : ce sont les opérateurs les plus simples, ils permettent de produire une nouvelle table à partir d'une autre table.
- **Les opérateurs binaires ensemblistes (Union, Intersection Différence)** : ces opérateurs permettent de produire une nouvelle relation à partir de deux relations de même degré et de même domaine.
- **Les opérateurs binaires ou n-aires (Produit cartésien, Jointure, Division)** : ils permettent de produire une nouvelle table à partir de deux ou plusieurs autres tables.

Les notations ne sont pas standardisées en algèbre relationnelle. Ce cours utilise des notations courantes mais donc pas forcément universelles.

2.3.2 Sélection

La sélection génère une relation regroupant exclusivement toutes les occurrences de la relation R qui satisfont l'expression logique E, on la note $\sigma_{(E)}R$.

Il s'agit d'une opération unaire essentielle dont la signature est :

relation * expression logique -> relation

En d'autres termes, la sélection permet de choisir (i.e. sélectionner) des lignes dans le tableau. Le résultat de la sélection est donc une nouvelle relation qui a les mêmes attributs que R. Si R est vide (i.e. ne contient aucune occurrence), la relation qui résulte de la sélection est vide.

Le tableau 2 montre un exemple de sélection.

Numéro	**Nom**	**Prénom**
5	Durand	Caroline
1	Germain	Stan
12	Dupont	Lisa
3	Germain	Rose-Marie

Tableau 1 : Exemple de relation *Personne*

Numéro	**Nom**	**Prénom**
5	Durand	Caroline
12	Dupont	Lisa

Tableau 2 : Exemple de sélection sur la relation *Personne* du tableau 1 : $\sigma_{(Numéro \geq 5)}Personne$

2.3.3 Projection

La projection consiste à supprimer les attributs autres que $A_1, \ldots A_n$ d'une relation et à éliminer les n-uplets en double apparaissant dans la nouvelle relation ; on la note $\Pi_{(A1, \ldots An)}R$

Il s'agit d'une opération unaire essentielle dont la signature est :

relation * liste d'attributs -> relation

En d'autres termes, la projection permet de choisir des colonnes dans le tableau. Si R est vide, la relation qui résulte de la projection est vide, mais pas forcément équivalente (elle contient généralement moins d'attributs).

Nom
Durand
Germain
Dupont

Tableau 3: Exemple de projection sur la relation *Personne* du tableau 1 : $\Pi_{(Nom)}Personne$

2.3.4 Union

L'union est une opération portant sur deux relations R_1 et R_2 ayant le même schéma et construisant une troisième relation constituée des n-uplets appartenant à chacune des deux relations R_1 et R_2 sans doublon, on la note $R_1 \cup R_2$.

Il s'agit une opération binaire ensembliste commutative essentielle dont la signature est :

relation * relation -> relation

Comme nous l'avons déjà dit, R1 et R2 doivent avoir les mêmes attributs et si une même occurrence existe dans R1 et R2, elle n'apparaît qu'une seule fois dans le résultat de l'union. Le résultat de l'union est une nouvelle relation qui a les mêmes attributs que R1 et R2. Si R1 et R2 sont vides, la relation qui résulte de l'union est vide. Si R1 (respectivement R2) est vide, la relation qui résulte de l'union est identique à R2 (respectivement R1).

Relation R_1		**Relation** R_2		**Relation** R	
Nom	**Prénom**	**Nom**	**Prénom**	**Nom**	**Prénom**
Durand	Caroline	Dupont	Lisa	Durand	Caroline
Germain	Stan	Juny	Carole	Germain	Stan
Dupont	Lisa	Fourt	Lisa	Dupont	Lisa
Germain	Rose-Marie			Germain	Rose-Marie
				Juny	Carole
				Fourt	Lisa

Tableau 4: Exemple d'union : $R = R_1 \cup R_2$

2.3.5 Intersection

L'intersection est une opération portant sur deux relations R_1 et R_2 ayant le même schéma et construisant une troisième relation dont les n-uplets sont constitués de ceux appartenant aux deux relations, on la note $R_1 \cap R_2$.

Il s'agit une opération binaire ensembliste commutative dont la signature est :

relation * relation -> relation

Comme nous l'avons déjà dit, R1 et R2 doivent avoir les mêmes attributs. Le résultat de l'intersection est une nouvelle relation qui a les mêmes attributs que R1 et R2. Si R1 ou R2 ou les deux sont vides, la relation qui résulte de l'intersection est vide.

Relation R_1

Nom	**Prénom**
Durand	Caroline
Germain	Stan
Dupont	Lisa
Germain	Rose-Marie
Juny	Carole

Relation R_2

Nom	**Prénom**
Dupont	Lisa
Juny	Carole
Fourt	Lisa
Durand	Caroline

Relation R

Nom	**Prénom**
Durand	Caroline
Dupont	Lisa
Juny	Carole

Tableau 5: Exemple d'intersection : $R = R_1 \cap R_2$

2.3.6 Différence

La différence est une opération portant sur deux relations R_1 et R_2 ayant le même schéma et construisant une troisième relation dont les n-uplets sont constitués de ceux ne se trouvant que dans la relation R_1 ; on la note $R_1 - R_2$.

Il s'agit une opération binaire ensembliste non commutative essentielle dont la signature est :

relation * relation -> relation

Comme nous l'avons déjà dit, R1 et R2 doivent avoir les mêmes attributs. Le résultat de la différence est une nouvelle relation qui a les mêmes attributs que R1 et R2. Si R1 est vide, la relation qui résulte de la différence est vide. Si R2 est vide, la relation qui résulte de la différence est identique à R1.

Relation R_1	
Nom	**Prénom**
Durand	Caroline
Germain	Stan
Dupont	Lisa
Germain	Rose-Marie
Juny	Carole

Relation R_2	
Nom	**Prénom**
Dupont	Lisa
Juny	Carole
Fourt	Lisa
Durand	Caroline

Relation R	
Nom	**Prénom**
Germain	Stan
Germain	Rose-Marie

Tableau 6 : Exemple de différence : $R = R_1 - R_2$

2.3.7 Produit cartésien

Le produit cartésien est une opération portant sur deux relations R_1 et R_2 et qui construit une troisième relation regroupant exclusivement toutes les possibilités de combinaison des occurrences des relations R_1 et R_2, on la note $R_1 \times R_2$.

Il s'agit une opération binaire commutative essentielle dont la signature est :

relation * relation -> relation

Le résultat du produit cartésien est une nouvelle relation qui a tous les attributs de R1 et tous ceux de R2. Si R1 ou R2 ou les deux sont vides, la relation qui résulte du produit cartésien est vide. Le nombre d'occurrences de la relation qui résulte du produit cartésien est le nombre d'occurrences de R1 multiplié par le nombre d'occurrences de R2.

Relation *Amie*	
Nom	**Prénom**
Fourt	Lisa
Juny	Carole

Relation *Cadeau*	
Article	**Prix**
livre	45
poupée	25
montre	87

Relation *R*			
Nom	**Prénom**	**Article**	**Prix**
Fourt	Lisa	livre	45
Fourt	Lisa	poupée	25
Fourt	Lisa	montre	87
Juny	Carole	livre	45
Juny	Carole	poupée	25
Juny	Carole	montre	87

Tableau 7 : Exemple de produit cartésien : $R = Amie \times Cadeau$

2.3.8 Jointure, theta-jointure, equi-jointure, jointure naturelle

a) Jointure

La jointure est une opération portant sur deux relations R_1 et R_2 qui construit une troisième relation regroupant exclusivement toutes les possibilités de combinaison des occurrences des relations R_1 et R_2 qui satisfont l'expression logique E. La jointure est notée $R_1 \triangleright\!\triangleleft_E R_2$.

Il s'agit d'une opération binaire commutative dont la signature est :

relation × relation × expression logique ⟶ relation

Si R_1 ou R_2 ou les deux sont vides, la relation qui résulte de la jointure est vide.

En fait, la jointure n'est rien d'autre qu'un produit cartésien suivi d'une sélection :

$R_1 \triangleright\!\triangleleft_E R_2 = \sigma_E (R_1 \times R_2)$

Relation *Famille*

Nom	**Prénom**	**Age**
Fourt	Lisa	6
Juny	Carole	42
Fidus	Laure	16

Relation *Cadeau*

AgeC	**Article**	**Prix**
99	livre	30
6	poupée	60
20	baladeur	45
10	déguisement	15

Relation *R*

Nom	**Prénom**	**Age**	**AgeC**	**Article**	**Prix**
Fourt	Lisa	6	99	livre	30
Fourt	Lisa	6	20	baladeur	45
Fourt	Lisa	6	10	déguisement	15
Juny	Carole	42	99	livre	30
Fidus	Laure	16	99	livre	30
Fidus	Laure	16	20	baladeur	45

Tableau 8: Exemple de jointure : $R = Famille \triangleright\!\triangleleft_{((Age \leq AgeC) \wedge (Prix < 50))} Cadeau$

b)Theta-jointure

Une theta-jointure est une jointure dans laquelle l'expression logique E est une simple comparaison entre un attribut A_1 de la relation R_1 et un attribut A_2 de la relation R_2. La theta-jointure est notée $R_1 \triangleright\!\triangleleft_E R_2$.

c) Equi-jointure

Une equi-jointure est une theta-jointure dans laquelle l'expression logique E est un test d'égalité entre un attribut A_1 de la relation R_1 et un attribut A_2 de la relation R_2. L'equi-jointure est notée $R_1 \triangleright\!\triangleleft_{A1,A2} R_2$.

Remarque : Il vaut mieux écrire $R_1 \triangleright\!\triangleleft_{A1=A2} R_2$ que $R_1 \triangleright\!\triangleleft_{A1,A2} R_2$ car cette dernière notation peut prêter à confusion avec une jointure naturelle explicite

d) Jointure naturelle

Une jointure naturelle est une jointure dans laquelle l'expression logique E est un test d'égalité entre les attributs qui portent le même nom dans les relations R_1 et R_2. Dans la relation construite, ces attributs ne sont pas dupliqués mais fusionnés en une seul colonne par couple d'attributs. La jointure naturelle est notée $R_1 \triangleright\!\triangleleft R_2$. On peut préciser explicitement les attributs communs à R_1 et R_2 sur lesquels porte la jointure : $R_1 \triangleright\!\triangleleft_{A1,\ldots,An} R_2$.

Généralement, R_1 et R_2 n'ont qu'un attribut en commun. Dans ce cas, une jointure naturelle est équivalente à une *equi-jointure* dans laquelle l'attribut de R_1 et celui de R_2 sont justement les deux attributs qui portent le même nom.

Lorsque l'on désire effectuer une jointure naturelle entre R_1 et R_2 sur un attribut A_1 commun à R_1 et R_2, il vaut mieux écrire $R_1 \triangleright\!\triangleleft_{A1} R_2$ que $R_1 \triangleright\!\triangleleft R_2$. En effet, si R_1 et R_2 possèdent deux attributs portant un nom commun, A_1 et A_2, $R_1 \triangleright\!\triangleleft_{A1} R_2$ est bien une jointure naturelle sur l'attribut A_1, mais $R_1 \triangleright\!\triangleleft R_2$ est une jointure naturelle sur le couple d'attributs A_1, A_2, ce qui produit un résultat très différent !

Relation *Famille*

Nom	**Prénom**	**Age**
Fourt	Lisa	6
Juny	Carole	40
Fidus	Laure	20
Choupy	Emma	6

Relation *Cadeau*

Age	**Article**	**Prix**
40	livre	45
6	poupée	25
20	montre	87

Relation *R*

Nom	**Prénom**	**Age**	**Article**	**Prix**
Fourt	Lisa	6	poupée	25
Juny	Carole	40	livre	45
Fidus	Laure	20	montre	87
Choupy	Emma	6	poupée	25

Tableau 9: Exemple de jointure naturelle : $R = Famille \triangleright\!\triangleleft Cadeau$ ou $R = Famille \triangleright\!\triangleleft_{Age} Cadeau$

2.3.9 Division

La division est une opération portant sur deux relations R_1 et R_2, telles que le schéma de R_2 est strictement inclus dans celui de R_1, qui génère une troisième relation regroupant toutes les parties d'occurrences de la relation R_1 qui sont associées à toutes les occurrences de la relation R_2 ; on la note $R_1 \div R_2$.

Il s'agit d'une opération binaire non commutative dont la signature est :

relation × relation ⟶ relation

Autrement dit, la division de R_1 par R_2 ($R_1 \div R_2$) génère une relation qui regroupe tous les n-uplets qui, concaténés à chacun des n-uplets de R_2, donne toujours un n-uplet de R_1.

La relation R_2 ne peut pas être vide. Tous les attributs de R_2 doivent être présents dans R_1 et R_1 doit posséder au moins un attribut de plus que R_2 (inclusion stricte). Le résultat de la division est une

nouvelle relation qui a tous les attributs de R_1 sans aucun de ceux de R_2. Si R_1 est vide, la relation qui résulte de la division est vide.

Relation *Enseignement*

Enseignant	**Etudiant**
Germain	Dubois
Fidus	Pascal
Robert	Dubois
Germain	Pascal
Fidus	Dubois
Germain	Durand
Robert	Durand

Relation *Etudiant*

Nom
Dubois
Pascal

Relation *R*

Enseignant
Germain
Fidus

Tableau 10 : Exemple de division : *R = Enseignement ÷ Etudiant*. La relation *R* contient donc tous les enseignants de la relation *Enseignement* qui enseignent à tous les étudiants de la relation *Etudiant*.

III – Langage SQL

3.1 Introduction SQL

Le langage SQL (Structured query language) peut être considéré comme le langage d'accès standard et normalisé, destiné à interroger ou à manipuler une base de données relationnelle

Il a fait l'objet de plusieurs normes ANSI/ISO dont la plus répandue aujourd'hui est la norme SQL2 qui a été définie en 1992.

Le succès du langage SQL est dû essentiellement à sa simplicité et au fait qu'il s'appuie sur le schéma conceptuel pour énoncer des requêtes en laissant le SGBD responsable de la stratégie d'exécution. Le langage SQL propose un langage de requêtes ensembliste et assertionnel. Néanmoins, le langage SQL ne possède pas la puissance d'un langage de programmation : entrées/sorties, instructions conditionnelles, boucles et affectations. Pour certains traitements, il est donc nécessaire de coupler le langage SQL avec un langage de programmation plus complet.
De manière synthétique, on peut dire que SQL est un langage relationnel, il manipule donc des tables (i.e. des relations, c'est-à-dire des ensembles) par l'intermédiaire de requêtes qui produisent également des tables.

3.2 Catégories d'instructions

Les instructions SQL sont regroupées en catégories en fonction de leur utilité et des entités manipulées. Nous pouvons distinguer cinq catégories, qui permettent :

- un langage de définition de données (LDD)
- un langage de manipulation de données (LMD)
- un langage de contrôle de données (LCD),
- un langage de contrôle des transactions (LCT)
- et d'autres modules destinés notamment à écrire des routines (procédures, fonctions ou déclencheurs) et interagir avec des langages externes.

3.2.1 Langage de définition de données

Le **langage de définition de données** (LDD, ou Data Definition Language, soit DDL en anglais) est un langage orienté au niveau de la structure de la base de données. Le LDD permet de créer, modifier, supprimer des objets. Il permet également de définir le domaine des données (nombre, chaîne de caractères, date, booléen, . . .) et d'ajouter des contraintes de valeur sur les données. Il permet enfin d'autoriser ou d'interdire l'accès aux données et d'activer ou de désactiver l'audit pour un utilisateur donné.

3.2.2 Langage de manipulation de données

Le **langage de manipulation de données** (LMD, ou Data Manipulation Language, soit DML en anglais) est l'ensemble des commandes concernant la manipulation des données dans une base de données. Le LMD permet l'ajout, la suppression et la modification de lignes, la visualisation du contenu des tables et leur verrouillage.

3.2.3 Langage de protections d'accès

Le **langage de protections d'accès** (ou Data Control Language, soit DCL en anglais) s'occupe de gérer les droits d'accès aux tables.

3.2.4 Langage de contrôle de transaction

Le **langage de contrôle de transaction** (ou Transaction Control Language, soit TCL en anglais) gère les modifications faites par le LMD, c'est-à-dire les caractéristiques des transactions et la validation et l'annulation des modifications.

3.2.5 SQL intégré

Le **SQL intégré** (Embedded SQL) permet d'utiliser SQL dans un langage de troisième génération (C, Java, Cobol, etc.) :
– déclaration d'objets ou d'instructions ;
– exécution d'instructions ;
– gestion des variables et des curseurs ;
– traitement des erreurs.

3.3 Définir/créer une base – Langage de Définition des Données (LDD)

3.3.1 Introduction aux contraintes d'intégrité

Soit le schéma relationnel minimaliste suivant:

- Acteur (Num-Act, Nom, Prénom)
- Jouer (Num-Act, Num-Film)
- Film (Num-Film, Titre, Année)

Contrainte d'intégrité de domaine

Toute comparaison d'attributs n'est acceptée que si ces attributs sont définis sur le même domaine. Le SGBD doit donc constamment s'assurer de la validité des valeurs d'un attribut. C'est pourquoi la commande de création de table doit préciser, en plus du nom, le type de chaque colonne.

Par exemple, pour la table *Film*, on précisera que le *Titre* est une chaîne de caractères et l'*Année* une date. Lors de l'insertion de n-uplets dans cette table, le système s'assurera que les différents champs du n-uplet satisfont les contraintes d'intégrité de domaine des attributs précisées lors de la création de la base. Si les contraintes ne sont pas satisfaites, le n-uplet n'est, tout simplement, pas inséré dans la table.

Contrainte d'intégrité de table (ou de relation ou d'entité)

Lors de l'insertion de n-uplets dans une table (*i.e.* une relation), il arrive qu'un attribut soit inconnu ou non défini. On introduit alors une valeur conventionnelle notée `NULL` et appelée *valeur nulle*.

Cependant, une clé primaire ne peut avoir une valeur nulle. De la même manière, une clé primaire doit toujours être unique dans une table. Cette contrainte forte qui porte sur la clé primaire est appelée contrainte d'intégrité de relation.

Tout SGBD relationnel doit vérifier l'unicité et le caractère défini (`NOT NULL`) des valeurs de la clé primaire.

Contrainte d'intégrité de référence

Dans tout schéma relationnel, il existe deux types de relation :

- les relations qui représentent des entités de l'univers modélisé ; elles sont qualifiées de statiques, ou d'indépendantes ; les relations *Acteur* et *Film* en sont des exemples ;
- les relations dont l'existence des n-uplets dépend des valeurs d'attributs situées dans d'autres relations ; il s'agit de relations dynamiques ou dépendantes ; la relation *Jouer* en est un exemple.

Lors de l'insertion d'un n-uplet dans la relation *Jouer*, le SGBD doit vérifier que les valeurs `Num-Act` et `Num-Film` correspondent bien, respectivement, à une valeur de `Num-Act` existant dans la relation *Acteur* et une valeur `Num-Film` existant dans la relation *Film*.

Lors de la suppression d'un n-uplet dans la relation *Acteur*, le SGBD doit vérifier qu'aucun n-uplet de la relation *Jouer* ne fait référence, par l'intermédiaire de l'attribut `Num-Act`, au n-uplet que l'on cherche à supprimer. Le cas échéant, c'est-à-dire si une, ou plusieurs, valeur correspondante de `Num-Act` existe dans *Jouer*, quatre possibilités sont envisageables :

- interdire la suppression ;
- supprimer également les n-uplets concernés dans *Jouer* ;
- avertir l'utilisateur d'une incohérence ;
- mettre les valeurs des attributs concernés à une valeur nulle dans la table *Jouer*, si l'opération est possible (ce qui n'est pas le cas si ces valeurs interviennent dans une clé primaire) ;

3.3.2 Créer une table : `CREATE TABLE`

Introduction

Une table est un ensemble de lignes et de colonnes. La création consiste à définir (en fonction de l'analyse) le nom de ces colonnes, leur format (*type*), la valeur par défaut à la création de la ligne (`DEFAULT`) et les règles de gestion s'appliquant à la colonne (`CONSTRAINT`).

Création simple

La commande de création de table la plus simple ne comportera que le nom et le type de chaque colonne de la table. A la création, la table sera vide, mais un certain espace lui sera alloué. La syntaxe est la suivante :

```
CREATE TABLE nom_table (nom_col1 TYPE1, nom_col2 TYPE2, ...)
```

Quand on crée une table, il faut définir les contraintes d'intégrité que devront respecter les données que l'on mettra dans la table (cf. section 3.3.3).

Les types de données

Les types de données peuvent être :

`INTEGER :`
: Ce type permet de stocker des entiers signés codés sur 4 octets.

`BIGINT :`
: Ce type permet de stocker des entiers signés codés sur 8 octets.

`REAL :`
: Ce type permet de stocker des réels comportant 6 chiffres significatifs codés sur 4 octets.

`DOUBLE PRECISION :`
: Ce type permet de stocker des réels comportant 15 chiffres significatifs codés sur 8 octets.

`NUMERIC[(précision, [longueur])] :`
: Ce type de données permet de stocker des données numériques à la fois entières et réelles avec une précision de 1000 chiffres significatifs. `longueur` précise le nombre maximum de chiffres significatifs stockés et `précision` donne le nombre maximum de chiffres après la virgule.

`CHAR(longueur) :`
: Ce type de données permet de stocker des chaînes de caractères de longueur fixe. `longueur` doit être inférieur à 255, sa valeur par défaut est 1.

`VARCHAR(longueur) :`
: Ce type de données permet de stocker des chaînes de caractères de longueur variable. `longueur` doit être inférieur à 2000, il n'y a pas de valeur par défaut.

`DATE :`
: Ce type de données permet de stocker des données constituées d'une date.

`TIMESTAMP :`
: Ce type de données permet de stocker des données constituées d'une date et d'une heure.

`BOOLEAN :`
: Ce type de données permet de stocker des valeurs Booléenne.

`MONEY :`
: Ce type de données permet de stocker des valeurs monétaires.

`TEXT :`
: Ce type de données permet des stocker des chaînes de caractères de longueur variable.

3.3.3 Contraintes d'intégrité

Syntaxe

A la création d'une table, les contraintes d'intégrité se déclarent de la façon suivante :

```
CREATE TABLE nom_table (
nom_col_1 type_1 [CONSTRAINT nom_1_1] contrainte_de_colonne_1_1
                 [CONSTRAINT nom_1_2] contrainte_de_colonne_1_2
                 ...                  ...
                 [CONSTRAINT nom_1_m] contrainte_de_colonne_2_m,
nom_col_2 type_2 [CONSTRAINT nom_2_1] contrainte_de_colonne_2_1
                 [CONSTRAINT nom_2_2] contrainte_de_colonne_2_2
                 ...                  ...
                 [CONSTRAINT nom_2_m] contrainte_de_colonne_2_m,
...
nom_col_n type_n [CONSTRAINT nom_n_1] contrainte_de_colonne_n_1
                 [CONSTRAINT nom_n_2] contrainte_de_colonne_n_2
                 ...                  ...
                 [CONSTRAINT nom_n_m] contrainte_de_colonne_n_m,
[CONSTRAINT nom_1] contrainte_de_table_1,
[CONSTRAINT nom_2] contrainte_de_table_2,
...                 ...
[CONSTRAINT nom_p] contrainte_de_table_p
)
```

Contraintes de colonne

Les différentes contraintes de colonne que l'on peut déclarer sont les suivantes :

`NOT NULL` ou `NULL` :
: Interdit (`NOT NULL`) ou autorise (`NULL`) l'insertion de valeur `NULL` pour cet attribut.

`UNIQUE` :
: Désigne l'attribut comme clé secondaire de la table. Deux n-uplets ne peuvent recevoir des valeurs identiques pour cet attribut, mais l'insertion de valeur `NULL` est toutefois autorisée. Cette contrainte peut apparaître plusieurs fois dans l'instruction.

`PRIMARY KEY` :
: Désigne l'attribut comme clé primaire de la table. La clé primaire étant unique, cette contrainte ne peut apparaître qu'une seule fois dans l'instruction. La définition d'une clé primaire composée se fait par l'intermédiaire d'une contrainte de table. En fait, la contrainte `PRIMARY KEY` est totalement équivalente à la contrainte `UNIQUE NOT NULL`.

`REFERENCES table [(colonne)] [ON DELETE CASCADE]` :
: Contrainte d'intégrité référentielle pour l'attribut de la table en cours de définition. Les valeurs prises par cet attribut doivent exister dans l'attribut `colonne` qui possède une contrainte `PRIMARY KEY` ou `UNIQUE` dans la table `table`. En l'absence de précision d'attribut `colonne`, l'attribut retenu est celui correspondant à la clé primaire de la table `table` spécifiée.

`CHECK (condition)` :
: Vérifie lors de l'insertion de n-uplets que l'attribut réalise la condition `condition`.

`DEFAULT valeur` :
: Permet de spécifier la valeur par défaut de l'attribut.

Contraintes de table

Les différentes contraintes de table que l'on peut déclarer sont les suivantes :

`PRIMARY KEY (colonne, …) :`

Désigne la concaténation des attributs cités comme clé primaire de la table. Cette contrainte ne peut apparaître qu'une seule fois dans l'instruction.

`UNIQUE (colonne, …) :`

Désigne la concaténation des attributs cités comme clé secondaire de la table. Dans ce cas, au moins une des colonnes participant à cette clé secondaire doit permettre de distinguer le n-uplet. Cette contrainte peut apparaître plusieurs fois dans l'instruction.

`FOREIGN KEY (colonne, …) REFERENCES table [(colonne, …)]`

`[ON DELETE CASCADE | SET NULL] :`

Contrainte d'intégrité référentielle pour un ensemble d'attributs de la table en cours de définition. Les valeurs prises par ces attributs doivent exister dans l'ensemble d'attributs spécifié et posséder une contrainte `PRIMARY KEY` ou `UNIQUE` dans la table `table`.

`CHECK (condition) :`

Cette contrainte permet d'exprimer une condition qui doit exister entre plusieurs attributs de la ligne.

Les contraintes de tables portent sur plusieurs attributs de la table sur laquelle elles sont définies. Il n'est pas possible de définir une contrainte d'intégrité utilisant des attributs provenant de deux ou plusieurs tables.

Complément sur les contraintes

`ON DELETE CASCADE :`

Demande la suppression des n-uplets dépendants, dans la table en cours de définition, quand le n-uplet contenant la clé primaire référencée est supprimé dans la table maître.

`ON DELETE SET NULL :`

Demande la mise à `NULL` des attributs constituant la clé étrangère qui font référence au n-uplet supprimé dans la table maître.

La suppression d'un n-uplet dans la table maître pourra être impossible s'il existe des n-uplets dans d'autres tables référençant cette valeur de clé primaire et ne spécifiant pas l'une de ces deux options.

3.3.4 Supprimer une table : `DROP TABLE`

Supprimer une table revient à éliminer sa structure et toutes les données qu'elle contient. Les *index* associés sont également supprimés.

La syntaxe est la suivante :

```
DROP TABLE nom_table
```

3.3.5 Modifier une table : `ALTER TABLE`

Ajout ou modification de colonne

```
ALTER TABLE nom_table {ADD/MODIFY} ([nom_colonne type [contrainte], ...])
```

Ajout d'une contrainte de table

```
ALTER TABLE nom_table ADD [CONSTRAINT nom_contrainte] contrainte
```

La syntaxe de déclaration de contrainte est identique à celle vue lors de la création de table.

Si des données sont déjà présentes dans la table au moment où la contrainte d'intégrité est ajoutée, toutes les lignes doivent vérifier la contrainte. Dans le cas contraire, la contrainte n'est pas posée sur la table.

Renommer une colonne

```
ALTER TABLE nom_table RENAME COLUMN ancien_nom TO nouveau_nom
```

Renommer une table

```
ALTER TABLE nom_table RENAME TO nouveau_nom
```

3.4 Modifier une base – Langage de manipulation de données (LMD)

3.4.1 Insertion de n-uplets : `INSERT INTO`

La commande `INSERT` permet d'insérer une ligne dans une table en spécifiant les valeurs à insérer. La syntaxe est la suivante :

```
INSERT INTO nom_table(nom_col_1, nom_col_2, ...)
VALUES (val_1, val_2, ...)
```

La liste des noms de colonne est optionnelle. Si elle est omise, la liste des colonnes sera par défaut la liste de l'ensemble des colonnes de la table dans l'ordre de la création de la table. Si une liste de colonnes est spécifiée, les colonnes ne figurant pas dans la liste auront la valeur `NULL`.

Il est possible d'insérer dans une table des lignes provenant d'une autre table. La syntaxe est la suivante :

```
INSERT INTO nom_table(nom_col1, nom_col2, ...)
SELECT ...
```

Le `SELECT` (cf. section 3.5 et 3.6) peut contenir n'importe quelle clause sauf un `ORDER BY` (cf. section 3.5.6).

3.4.2 Modification de n-uplets : `UPDATE`

La commande `UPDATE` permet de modifier les valeurs d'une ou plusieurs colonnes, dans une ou plusieurs lignes existantes d'une table. La syntaxe est la suivante :

```
UPDATE nom_table
SET nom_col_1 = {expression_1 | ( SELECT ...) },
    nom_col_2 = {expression_2 | ( SELECT ...) },
    ...
    nom_col_n = {expression_n | ( SELECT ...) }
WHERE predicat
```

Les valeurs des colonnes `nom_col_1, nom_col_2, ..., nom_col_n` sont modifiées dans toutes les lignes qui satisfont le prédicat `predicat`. En l'absence d'une clause `WHERE`, toutes les lignes sont mises à jour. Les expressions `expression_1, expression_2, ..., expression_n` peuvent faire référence aux anciennes valeurs de la ligne.

3.4.3 Suppression de n-uplets : `DELETE`

La commande `DELETE` permet de supprimer des lignes d'une table.

La syntaxe est la suivante :

```
DELETE FROM nom_table
WHERE predicat
```

Toutes les lignes pour lesquelles `predicat` est évalué à *vrai* sont supprimées. En l'absence de clause `WHERE`, toutes les lignes de la table sont supprimées.

3.5 Interroger une base – Langage de manipulation de données (LMD) : SELECT (1ère partie)

3.5.1 Introduction à la commande SELECT

Introduction

La commande SELECT constitue, à elle seule, le langage permettant d'interroger une base de données. Elle permet de :

- sélectionner certaines colonnes d'une table (projection) ;
- sélectionner certaines lignes d'une table en fonction de leur contenu (sélection) ;
- combiner des informations venant de plusieurs tables (jointure, union, intersection, différence et division) ;
- combiner entre elles ces différentes opérations.

Une requête (*i.e.* une interrogation) est une combinaison d'opérations portant sur des tables (relations) et dont le résultat est lui-même une table dont l'existence est éphémère (le temps de la requête).

Syntaxe simplifiée de la commande SELECT

Une requête se présente généralement sous la forme :

```
SELECT  [ ALL | DISTINCT ] { * | attribut [, ...] }
    FROM nom_table [, ...]
    [ WHERE condition ]
```

- la clause `SELECT` permet de spécifier les attributs que l'on désire voir apparaître dans le résultat de la requête ; le caractère *étoile* (`*`) récupère tous les attributs de la table générée par la clause `FROM` de la requête ;
- la clause `FROM` spécifie les tables sur lesquelles porte la requête ;
- la clause `WHERE`, qui est facultative, énonce une condition que doivent respecter les n-uplets sélectionnés.

Par exemple, pour afficher l'ensemble des n-uplets de la table `film`, vous pouvez utiliser la requête :

```
SELECT * FROM film
```

De manière synthétique, on peut dire que la clause `SELECT` permet de réaliser la *projection*, la clause `FROM` le *produit cartésien* et la clause `WHERE` la *sélection* (cf. section 3.5.2).

Délimiteurs : apostrophes simples et doubles

Pour spécifier littéralement une chaîne de caractères, il faut l'entourer d'apostrophes (*i.e.* guillemets simples). Par exemple, pour sélectionner les films *policiers*, on utilise la requête :

```
SELECT * FROM film WHERE genre='Policier'
```

Les date doivent également être entourée d'apostrophes (ex : `'01/01/2005'`).

Comme l'apostrophe est utilisée pour délimiter les chaînes de caractères, pour la représenter dans une chaîne, il faut la dédoubler (exemple : `'l''arbre'`), ou la faire précéder d'un antislash (exemple : `'l\'arbre'`).

Lorsque le nom d'un élément d'une base de données (un nom de table ou de colonne par exemple) est identique à un mot clef du SQL, il convient de l'entourer d'apostrophes doubles. Par exemple, si la table `achat` possède un attribut `date`, on pourra écrire :

```
SELECT ''date'' FROM achat
```

Bien entendu, les mots réservés du SQL sont déconseillés pour nommer de tels objets. Les apostrophes doubles sont également nécessaires lorsque le nom (d'une colonne ou d'une table) est composé de caractères particuliers tels que les blancs ou autres, ce qui est évidemment déconseillé.

3.5.2 Traduction des opérateurs de projection, sélection, produit cartésien et équi-jointure de l'algèbre relationnelle (1 ère partie)

Traduction de l'opérateur de projection

L'opérateur de projection $\Pi_{(A1, \ldots An)}$(*relation*) se traduit tout simplement en SQL par la requête :

```
SELECT DISTINCT A_1, ..., A_n FROM relation
```

`DISTINCT` permet de ne retenir qu'une occurrence de n-uplet dans le cas où une requête produit plusieurs n-uplets identiques (cf. section 3.5.4).

Traduction de l'opérateur de sélection

L'opérateur de sélection $\sigma_{(prédicat)}$(*relation*) se traduit tout simplement en SQL par la requête :

```
SELECT * FROM relation WHERE prédicat
```

De manière simplifiée, un *prédicat* est une expression logique sur des *comparaisons*. Cf. section 3.5.7 pour une description plus complète.

Traduction de l'opérateur de produit cartésien

L'opérateur de produit cartésien $relation_1 \times relation_2$ se traduit en SQL par la requête :

```
SELECT * FROM relation_1, relation_2
```

Nous reviendrons sur le produit cartésien dans les sections 3.5.5 et 3.6.1.

Traduction de l'opérateur d'équi-jointure

L'opérateur d'équi-jointure $relation_1 \triangleright\!\triangleleft_{A1,A2} relation_2$ se traduit en SQL par la requête :

```
SELECT * FROM relation_1, relation_2 WHERE relation_1.A_1 = relation_2.A_2
```

Nous reviendrons sur les différents types de jointure dans la section 3.6.1.

3.5.3 Syntaxe générale de la commande `SELECT`

Voici la syntaxe générale d'une commande SELECT :

```
SELECT [ ALL | DISTINCT ] { * | expression [ AS nom_affiché ] } [, ...]
    FROM nom_table [ [ AS ] alias ] [, ...]
    [ WHERE prédicat ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ {UNION | INTERSECT | EXCEPT [ALL]} requête ]
    [ ORDER BY expression [ ASC | DESC ] [, ...] ]
```

En fait l'ordre SQL `SELECT` est composé de 7 clauses dont 5 sont optionnelles :

`SELECT` :
Cette clause permet de spécifier les attributs que l'on désire voir apparaître dans le résultat de la requête (cf. section 3.5.4).

`FROM` :
Cette clause spécifie les tables sur lesquelles porte la requête (cf. section 3.5.5 et 3.6.1).

`WHERE` :
Cette clause permet de filtrer les n-uplets en imposant une condition à remplir pour qu'ils soient présents dans le résultat de la requête (cf. section 3.5.7).

`GROUP BY` :
Cette clause permet de définir des groupes (*i.e.* sous-ensemble ; cf. section 3.6.2).

`HAVING` :
Cette clause permet de spécifier un filtre (condition de regroupement des n-uplets) portant sur les résultats (cf. section 3.6.2).

`UNION`, `INTERSECT` et `EXCEPT` :
Cette clause permet d'effectuer des opérations ensemblistes entre plusieurs résultats de requête (*i.e.* entre plusieurs `SELECT`) (cf. section 3.6.3).

`ORDER BY` :
Cette clause permet de trier les n-uplets du résultat (cf. section 3.5.6).

3.5.4 La clause `SELECT`

Introduction

Comme nous l'avons déjà dit, la clause `SELECT` permet de spécifier les attributs que l'on désire voir apparaître dans le résultat de la requête. Pour préciser explicitement les attributs que l'on désire conserver, il faut les lister en les séparant par une virgule. Cela revient en fait à opérer une projection de la table intermédiaire générée par le reste de la requête. Nous verrons dans cette section que la clause `SELECT` permet d'aller plus loin que la simple opération de projection. En effet, cette clause permet également de renommer des colonnes, voire d'en créer de nouvelles à partir des colonnes existantes.

Pour illustrer par des exemples les sections qui suivent, nous utiliserons une table dont le schéma est le suivant :

```
employee(id_employee, surname, name, salary)
```

Cette table contient respectivement l'identifiant, le nom, le prénom et le salaire mensuel des employés d'une compagnie.

L'opérateur étoile *(*)*

Le caractère *étoile* (`*`) permet de récupérer automatiquement tous les attributs de la table générée par la clause `FROM` de la requête.

Pour afficher la table `employee` on peut utiliser la requête :

```
SELECT * FROM employee
```

Les opérateurs `DISTINCT` et `ALL`

Lorsque le SGBD construit la réponse d'une requête, il rapatrie toutes les lignes qui satisfont la requête, généralement dans l'ordre ou il les trouve, même si ces dernières sont en double (comportement `ALL` par défaut). C'est pourquoi il est souvent nécessaire d'utiliser le mot clef `DISTINCT` qui permet d'éliminer les doublons dans la réponse.

Par exemple, pour afficher la liste des prénoms, sans doublon, des employés de la compagnie, il faut utiliser la requête :

```
SELECT DISTINCT name FROM employee
```

Les opérations mathématiques de base

Il est possible d'utiliser les opérateurs mathématiques de base (*i.e.* `+`, `-`, `*` et `/`) pour générer de nouvelles colonnes à partir, en générale, d'une ou plusieurs colonnes existantes.

Pour afficher le nom, le prénom et le salaire annuel des employés, on peut utiliser la requête :

```
SELECT surname, name, salary*12 FROM employee
```

L'opérateur `AS`

Le mot clef `AS` permet de renommer une colonne, ou de nommer une colonne créée dans la requête.

Pour afficher le nom, le prénom et le salaire annuel des employés, on peut utiliser la requête :

```
SELECT surname AS nom, name AS prénom, salary*12 AS salaire FROM employee
```

L'opérateur de concaténation

L'opérateur `||` (double barre verticale) permet de concaténer des champs de type caractères.

Pour afficher le nom et le prénom sur une colonne, puis le salaire annuel des employés, on peut utiliser la requête :

```
SELECT surname || ' ' || name AS nom, salary*12 AS salaire FROM employee
```

3.5.5 La clause `FROM` (1 ère partie)

Comportement

Comme nous l'avons déjà dit, la clause `FROM` spécifie les tables sur lesquelles porte la requête. Plus exactement, cette clause construit la table intermédiaire (*i.e.* virtuelle), à partir d'une ou de plusieurs tables, sur laquelle des modifications seront apportées par les clauses `SELECT`, `WHERE`, `GROUP BY` et `HAVING` pour générer la table finale résultat de la requête. Quand plusieurs tables, séparées par des virgules, sont énumérées dans la clause `FROM`, la table intermédiaire est le résultat du produit cartésien de toutes les tables énumérées.

L'opérateur `AS`

Le mot clef `AS` permet de renommer une table, ou de nommer une table créée dans la requête (c'est à dire une sous-requête) afin de pouvoir ensuite y faire référence. Le renommage du nom d'une table se fait de l'une des deux manières suivantes :

```
FROM nom_de_table AS nouveau_nom
FROM nom_de_table nouveau_nom
```

Une application typique du renommage de table est de simplifier les noms trop long :

```
SELECT * FROM nom_de_table_1 AS t1, nom_de_table_1 AS t2 WHERE t1.A_1 = t2.A_2
```

Attention, le nouveau nom remplace complètement l'ancien nom de la table dans la requête. Ainsi, quand une table a été renommée, il n'est plus possible d'y faire référence en utilisant son ancien nom. La requête suivante n'est donc pas valide :

```
SELECT * FROM nom_table AS t WHERE nom_table.a > 5
```

Sous-requête

Les tables mentionnées dans la clause `FROM` peuvent très bien correspondre à des tables résultant d'une requête, spécifiée entre parenthèses, plutôt qu'à des tables existantes dans la base de données. Il faut toujours nommer les tables correspondant à des sous-requêtes en utilisant l'opérateur `AS`.

Par exemple, les deux requêtes suivantes sont équivalentes :

```
SELECT * FROM table_1, table_2
SELECT * FROM (SELECT * FROM table_1) AS t1, table_2
```

Les jointures

Nous traiterons cet aspect de la clause `FROM` dans la section 3.6.1.

3.5.6 La clause `ORDER BY`

Comme nous l'avons déjà dit, la clause `ORDER BY` permet de trier les n-uplets du résultat et sa syntaxe est la suivante :

```
ORDER BY expression [ ASC | DESC ] [, ...]
```

`expression` désigne soit une colonne, soit une opération mathématique de base (nous avons abordé ce type d'opérations dans la section 3.5.4 sur « La clause `SELECT` ») sur les colonnes.

`ASC` spécifie l'ordre ascendant et `DESC` l'ordre descendant du tri. En l'absence de précision `ASC` ou `DESC`, c'est l'ordre ascendant qui est utilisé par défaut.

Quand plusieurs expressions, ou colonnes sont mentionnées, le tri se fait d'abord selon les premières, puis suivant les suivantes pour les n-uplet qui sont égaux selon les premières.

Le tri est un tri interne sur le résultat final de la requête, il ne faut donc placer dans cette clause que les noms des colonnes mentionnés dans la clause `SELECT`.

La clause `ORDER BY` permet de trier le résultat final de la requête, elle est donc la dernière clause de tout ordre SQL et ne doit figurer qu'une seule fois dans le `SELECT`, même s'il existe des requêtes imbriquées ou un jeu de requêtes ensemblistes (cf. section 3.6.3).

En l'absence de clause `ORDER BY`, l'ordre des n-uplet est aléatoire et non garanti. Souvent, le fait de placer le mot clef `DISTINCT` suffit à établir un tri puisque le SGBD doit se livrer à une comparaison des lignes, mais ce mécanisme n'est pas garanti car ce tri s'effectue dans un ordre non contrôlable qui peut varier d'un serveur à l'autre.

3.5.7 La clause `WHERE`

Comportement

Comme nous l'avons déjà dit, la clause `WHERE` permet de filtrer les n-uplets en imposant une condition à remplir pour qu'ils soient présents dans le résultat de la requête ; sa syntaxe est la suivante :

```
WHERE prédicat
```

Concrètement, après que la table intermédiaire (*i.e.* virtuelle) de la clause `FROM` a été construite, chaque ligne de la table est confrontée au prédicat `prédicat` afin de vérifier si la ligne satisfait (*i.e.* le prédicat est *vrai* pour cette ligne) ou ne satisfait pas (*i.e.* le prédicat est *faux* ou `NULL` pour cette ligne) le prédicat. Les lignes qui ne satisfont pas le prédicat sont supprimées de la table intermédiaire.

Le prédicat n'est rien d'autre qu'une expression logique. En principe, celle-ci fait intervenir une ou plusieurs lignes de la table générée par la clause `FROM`, cela n'est pas impératif mais, dans le cas contraire, l'utilité de la clause `WHERE` serait nulle.

Expression simple

Une expression simple peut être une variable désignée par un nom de colonne ou une constante. Si la variable désigne un nom de colonne, la valeur de la variable sera la valeur située dans la table à l'intersection de la colonne et de la ligne dont le SGBD cherche à vérifier si elle satisfait le prédicat de la clause `WHERE`.

Les expressions simples peuvent être de trois types : numérique, chaîne de caractères ou date.

Une expression simple peut également être le résultat d'une sous-requête, spécifiée entre parenthèses, qui retourne une table ne contenant qu'une seule ligne et qu'une seule colonne (*i.e.* une sous-requête retournant une valeur unique).

Prédicat simple

Un prédicat simple peut être le résultat de la comparaison de deux *expressions simples* au moyen de l'un des opérateurs suivants :

`=`	égal
`!=`	différent
`<`	strictement inférieur
`<=`	inférieur ou égal
`>`	strictement supérieur
`>=`	supérieur ou égal

Dans ce cas, les trois types d'expressions (numérique, chaîne de caractères et date) peuvent être comparés. Pour les types date, la relation d'ordre est l'ordre chronologique. Pour les caractères, la relation d'ordre est l'ordre lexicographique.

Un prédicat simple peut également correspondre à un test de description d'une chaîne de caractères par une expression régulière :

`~`	décrit par l'expression régulière
`~*`	comme `LIKE` mais sans tenir compte de la casse
`!~`	non décrit par l'expression régulière
`!~*`	comme `NOT LIKE` mais sans tenir compte de la casse

Dans ce cas, la chaîne de caractères faisant l'objet du test est à gauche et correspond à une *expression simple* du type chaîne de caractères, il s'agit généralement d'un nom de colonne. L'expression régulière, qui s'écrit entre apostrophe simple, comme une chaîne de caractères, est située à droite de l'opérateur. La section 3.5.8 donne une description détaillée du formalisme des expressions régulières.

Un prédicat simple peut enfin correspondre à l'un des tests suivants :

`expr IS NULL`	test sur l'indétermination de `expr`
`expr IN (expr_1 [, ...])`	comparaison de `expr` à une liste de valeurs
`expr NOT IN (expr_1 [, ...])`	test d'absence d'une liste de valeurs
`expr IN (requête)`	même chose, mais la liste de valeurs est le résultat d'une
`expr NOT IN (requête)`	sous-requête qui doit impérativement retourner une table
	ne contenant qu'une colonne
`EXIST (requête)`	*vraie* si la sous-requête retourne au moins un n-uplet
	vraie si au moins un n-uplet de la sous-requête vérifie la
`expr operateur ANY (requête)`	comparaison « `expr opérateur` n-uplet » ; la sous-requête
	doit impérativement retourner une table ne contenant
	qu'une colonne ; `IN` est équivalent à `= ANY`
	vraie si tous les n-uplets de la sous-requête vérifient la
`expr operateur ALL (requête)`	comparaison « `expr opérateur` n-uplet » ; la sous-requête
	doit impérativement retourner une table ne contenant
	qu'une colonne

Dans ce tableau, `expr` désigne une *expression simple* et `requête` une sous-requête.

Prédicat composé

Les prédicats simples peuvent être combinés au sein d'expression logiques en utilisant les opérateurs logiques `AND` (*et* logique), `OR` (*ou* logique) et `NOT` (*négation* logique).

3.5.8 Les expressions régulières

Introduction

Le terme *expression régulière* est issu de la théorie informatique et fait référence à un ensemble de règles permettant de définir un ensemble de chaînes de caractères.

Une expression régulière constitue donc une manière compacte de définir un ensemble de chaînes de caractères. Nous dirons qu'une chaîne de caractères est décrite par une expression régulière si cette chaîne est un élément de l'ensemble de chaînes de caractères défini par l'expression régulière.

PostgreSQL dispose de trois opérateurs de description par une expression régulière :

1. `LIKE` ou `~~`
2. `~`
3. `SIMILAR TO`

La syntaxe et le pouvoir expressif des expressions régulières diffèrent pour ces trois opérateurs. Nous ne décrirons ici que la syntaxe du formalisme le plus standard et le plus puissant, celui que

l'on retrouve sous *Unix* avec les commandes `egrep`, `sed` et `awk`. Ce formalisme est celui associé à l'opérateur ~.

Avec PostgreSQL, le test d'égalité avec une chaîne de caractères s'écrit :

```
expression='chaine'
```

De manière équivalente, le test de description par une expression régulière s'écrit :

```
expression~'expression_régulière'
```

L'opérateur de description ~ est sensible à la casse, l'opérateur de description insensible à la casse est `~*`. L'opérateur de non description sensible à la casse est `!~`, son équivalent insensible à la casse se note `!~*`.

Formalisme

Comme nous allons le voir, dans une expression régulière, certains symboles ont une signification spéciale. Dans ce qui suit, `expreg`, `expreg_1`, `expreg_2` désignent des expressions régulières, `caractère` un caractère quelconque et `liste_de_caractères` une liste de caractères quelconque.

`caractère` :

> un caractère est une expression régulière qui désigne le caractère lui-même, excepté pour les caractères `.`, `?`, `+`, `*`, `{`, `|`, `(`, `)`, `^`, `$`, `\`, `[`, `]`. Ces derniers sont des méta-caractères et ont une signification spéciale. Pour désigner ces méta-caractères, il faut les faire précéder d'un antislash (`\.`, `\?`, `\+`, `\*`, `\{`, `\|`, `\(`, `\)`, `\^`, `\$`, `\\`, `\[`, `\]`).

`[liste_de_caractères]` :

> est une expression régulière qui décrit l'un des caractères de la liste de caractères, par exemple `[abcdf]` décrit le caractère `a`, le `b`, le `c`, le `d` ou le `f` ; le caractère `-` permet de décrire des ensembles de caractères consécutifs, par exemple `[a-df]` est équivalent à `[abcdf]` ; la plupart des méta-caractères perdent leur signification spéciale dans une liste, pour insérer un `]` dans une liste, il faut le mettre en tête de liste, pour inclure un `^`, il faut le mettre n'importe où sauf en tête de liste, enfin un `-` se place à la fin de la liste.

`[^liste_de_caractères]` :

> est une expression régulière qui décrit les caractères qui ne sont pas dans la liste de caractères.

`[:alnum:]` :

> à l'intérieur d'une liste, décrit un caractère alpha-numérique (`[[:alnum:]]` est équivalent à `[0-9A-Za-z]`) ; sur le même principe, on a également `[:alpha:]`, `[:cntrl:]`, `[:digit:]`, `[:graph:]`, `[:lower:]`, `[:print:]`, `[:punct:]`, `[:space:]`, `[:upper:]` et `[:xdigit:]`.

`.` :

> est une expression régulière et un méta-caractère qui désigne n'importe quel caractère.

`^` :

> est une expression régulière et un méta-caractère qui désigne le début d'une chaîne de caractères.

`$` :

> est une expression régulière et un méta-caractère qui désigne la fin d'une chaîne de caractères.

`expreg?` :

est une expression régulière qui décrit zéro ou une fois `expreg`.

`expreg*` :

est une expression régulière qui décrit `expreg` un nombre quelconque de fois, zéro compris.

`expreg+` :

est une expression régulière qui décrit `expreg` au moins une fois.

`expreg{n}` :

est une expression régulière qui décrit `expreg` `n` fois.

`expreg{n,}` :

est une expression régulière qui décrit `expreg` au moins `n` fois.

`expreg{n,m}` :

décrit `expreg` au moins `n` fois et au plus `m` fois.

`expreg_1expreg_2` :

est une expression régulière qui décrit une chaîne constituée de la concaténation de deux sous-chaînes respectivement décrites par `expreg_1` et `expreg_2`.

`expreg_1|expreg_2` :

est une expression régulière qui décrit toute chaîne décrite par `expreg_1` ou par `expreg_2`.

`(expreg)` :

est une expression régulière qui décrit ce que décrit `expreg`.

`\n` :

où `n` est un chiffre, est une expression régulière qui décrit la sous-chaîne décrite par la `n` ème sous-expression parenthèsée de l'expression régulière.

Remarque :

la concaténation de deux expressions régulières (`expreg_1expreg_2`) est une opération prioritaire sur l'union (`expreg_1|expreg_2`).

Exemples

Un caractère, qui n'est pas un méta-caractère, se décrit lui-même. Ce qui signifie que si vous cherchez une chaîne qui contient « voiture », vous devez utiliser l'expression régulière `'voiture'`.

Si vous ne cherchez que les motifs situés en début de ligne, utilisez le symbole `^`. Pour chercher toutes les chaînes qui commencent par « voiture », utilisez `'^voiture'`.

Le signe `$` (dollar) indique que vous souhaitez trouver les motifs en fin de ligne. Ainsi : `'voiture$'` permet de trouver toutes les chaînes finissant par « voiture ».

Le symbole . (point) remplace n'importe quel caractère. Pour trouver toutes les occurrences du motif composé des lettres `vo`, de trois lettres quelconques, et de la lettre `e`, utilisez : `'vo...e'`. Cette commande permet de trouver des chaînes comme : `voyagent`, `voyage`, `voyager`, `voyageur`, `vous_e`.

Vous pouvez aussi définir un ensemble de lettres en les insérant entre crochets `[ ]`. Pour chercher toutes les chaînes qui contiennent les lettres `P` ou `p` suivies de `rince`, utilisez :`'[Pp]rince'`.

Si vous voulez spécifier un intervalle de caractères, servez-vous d'un trait d'union pour délimiter le début et la fin de l'intervalle. Vous pouvez aussi définir plusieurs intervalles simultanément. Par exemple `[A-Za-z]` désigne toutes les lettres de l'alphabet, hormis les caractères accentués, quelque soit la casse. Notez bien qu'un intervalle ne correspond qu'à un caractère dans le texte.

Le symbole `*` est utilisé pour définir zéro ou plusieurs occurrences du motif précédent. Par exemple, l'expression régulière `'^Pa(pa)*$'` décrit les chaînes : `Pa`, `Papa`, `Papapa`, `Papapapapapapa`, …

Si vous souhaitez qu'un symbole soit interprété littéralement, il faut le préfixer par un `\`. Pour trouver toutes les lignes qui contiennent le symbole `$`, utilisez : `\$`.

3.6 Interroger une base – Langage de manipulation de données (LMD) : SELECT (2 ème partie)

3.6.1 La clause `FROM` (2 ème partie) : les jointures

Recommandation

Dans la mesure du possible, et contrairement à ce que nous avons fait jusqu'à présent, il est préférable d'utiliser un opérateur de jointure normalisé SQL2 (mot-clef `JOIN`) pour effectuer une jointure. En effet, les jointures faites dans la clause `WHERE` (ancienne syntaxe datant de 1986) ne permettent pas de faire la distinction, de prime abord, entre ce qui relève de la sélection et ce qui relève de la jointure puisque tout est regroupé dans une seule clause (la clause `WHERE`). La lisibilité des requêtes est plus grande en utilisant la syntaxe de l'opérateur `JOIN` qui permet d'isoler les conditions de sélections (clause `WHERE`) de celles de jointures (clauses `JOIN`), et qui permet également de cloisonner les conditions de jointures entre chaque couple de table. De plus, l'optimisation d'exécution de la requête est souvent plus pointue lorsque l'on utilise l'opérateur `JOIN`. Enfin, lorsque l'on utilise l'ancienne syntaxe, la suppression de la clause `WHERE` à des fins de tests pose évidemment des problèmes.

Le produit cartésien

Prenons une opération de jointure entre deux tables R_1 et R_2 selon une expression logique E. En algèbre relationnelle, cette opération se note :

$R_1 \triangleright\triangleleft_E R_2$

Dans la section 2.3.8, nous avons vu que la jointure n'est rien d'autre qu'un produit cartésien suivi d'une sélection :

$R_1 \triangleright\triangleleft_E R_2 = \sigma_E (R_1 \times R_2)$

On peut également dire que le produit cartésien n'est rien d'autre qu'une jointure dans laquelle l'expression logique E est toujours vraie :

$R_1 \times R_2 = R_1 \triangleright\triangleleft_{true} R_2$

Nous avons vu section 3.5.5 que le produit cartésien entre deux tables `table_1` et `table_2` peut s'écrire en SQL :

```
SELECT * FROM table_1, table_2
```

Il peut également s'écrire en utilisant le mot-clé `JOIN` dédié aux jointures de la manière suivante :

```
SELECT * FROM table_1 CROSS JOIN table_2
```

En fait, sous PostgreSQL, les quatre écritures suivantes sont équivalentes :

```
SELECT * FROM table_1, table_2
SELECT * FROM table_1 CROSS JOIN table_2
SELECT * FROM table_1 JOIN table_2 ON TRUE
```

```
SELECT * FROM table_1 INNER JOIN table_2 ON TRUE
```

Les deux dernières écritures prendront un sens dans les sections qui suivent.

Syntaxe générale des jointures

Sans compter l'opérateur `CROSS JOIN`, voici les trois syntaxes possibles de l'expression d'une jointure dans la clause `FROM` en SQL :

```
table_1          [ INNER | { { LEFT | RIGHT | FULL } [OUTER] } ] JOIN table_2 ON
predicat [...]
table_1           [ INNER | { { LEFT | RIGHT | FULL } [OUTER] } ] JOIN table_2
USING (colonnes) [...]
table_1 NATURAL [ INNER | { { LEFT | RIGHT | FULL } [OUTER] } ] JOIN table_2
[...]
```

Ces trois syntaxes diffèrent par la condition de jointure spécifiée par les clause `ON` ou `USING`, ou implicite dans le cas d'une jointure naturelle introduite par le mot-clé `NATURAL`.

`ON` :

La clause `ON` correspond à la condition de jointure la plus générale. Le prédicat `predicat` est une expression logique de la même nature que celle de la clause `WHERE` décrite dans la section 3.5.7.

`USING` :

La clause `USING` est une notation correspondant à un cas particulier de la clause `ON` qui, de plus supprime les colonnes superflues. Les deux tables, sur lesquelles portent la jointure, doivent posséder toutes les colonnes qui sont mentionnées, en les séparant par des virgules, dans la liste spécifiée entre parenthèses juste après le mot-clé `USING`. La condition de jointure sera l'égalité des colonnes au sein de chacune des paires de colonnes. De plus, les paires de colonnes seront fusionnées en une colonne unique dans la table résultat de la jointure. Par rapport à une jointure classique, la table résultat comportera autant de colonnes de moins que de colonnes spécifiées dans la liste de la clause `USING`.

`NATURAL` :

Il s'agit d'une notation abrégée de la clause `USING` dans laquelle la liste de colonnes est implicite et correspond à la liste des colonnes communes aux deux tables participant à la jointure. Tout comme dans le cas de la clause `USING`, les colonnes communes n'apparaissent qu'une fois dans la table résultat.

`INNER` et `OUTER` :

Les mots-clé `INNER` et `OUTER` permettent de préciser s'il s'agit d'une jointure *interne* ou *externe*. `INNER` et `OUTER` sont toujours optionnels. En effet, le comportement par défaut est celui de la jointure interne (`INNER`) et les mots clefs `LEFT`, `RIGHT` et `FULL` impliquent forcément une jointure externe (`OUTER`).

`INNER JOIN` :

La table résultat est constituée de toutes les juxtapositions possibles d'une ligne de la table `table_1` avec une ligne de la table `table_2` qui satisfont la condition de jointure.

`LEFT OUTER JOIN` :

Dans un premier temps, une jointure interne (*i.e.* de type `INNER JOIN`) est effectuée. Ensuite, chacune des lignes de la table `table_1` qui ne satisfait la condition de jointure avec aucune des lignes de la table `table_2` (*i.e.* les lignes de `table_1` qui n'apparaissent pas dans la table résultat de la jointure interne) est ajoutée à la table résultats. Les attributs

correspondant à la table `table_2`, pour cette ligne, sont affectés de la valeur `NULL`. Ainsi, la table résultat contient au moins autant de lignes que la table `table_1`.

`RIGHT OUTER JOIN :`

Même scénario que pour l'opération de jointure de type `LEFT OUTER JOIN`, mais en inversant les rôles des tables `table_1` et `table_2`.

`FULL OUTER JOIN :`

La jointure externe bilatérale est la combinaison des deux opérations précédentes (`LEFT OUTER JOIN` et `RIGHT OUTER JOIN`) afin que la table résultat contienne au moins une occurrence de chacune des lignes des deux tables impliquées dans l'opération de jointure.

La jointure externe droite peut être obtenue par une jointure externe gauche dans laquelle on inverse l'ordre des tables (et vice-versa). La jointure externe bilatérale peut être obtenue par la combinaison de deux jointures externes unilatérales avec l'opérateur ensembliste `UNION` que nous verrons dans la section 3.6.3.

Des jointures de n'importe quel type peuvent être chaînées les unes derrières les autres. Les jointures peuvent également être imbriquées étant donné que les tables `table_1` et `table_2` peuvent très bien être elles-mêmes le résultat de jointures de n'importe quel type. Les opérations de jointures peuvent être parenthésées afin de préciser l'ordre dans lequel elles sont effectuées. En l'absence de parenthèses, les jointures s'effectuent de gauche à droite.

Définition de deux tables pour les exemples qui suivent

Afin d'illustrer les opérations de jointure, considérons les tables `realisateur` et `film` définies de la manière suivante :

```
 create table realisateur (
    id_real integer primary key,
    nom varchar(16),
    prenom varchar(16)
);
create table film (
    num_film integer primary key,
    id_real integer,
    titre varchar(32)
);
```

On notera que dans la table `film`, l'attribut `id_real` correspond à une clef étrangère et aurait dû être défini de la manière suivante : `id_real integer references realisateur`. Nous ne l'avons pas fait dans le but d'introduire des films dont le réalisateur n'existe pas dans la table `realisateur` afin d'illustrer les différentes facettes des opérations de jointure.

La table `realisateur` contient les lignes suivantes :

```
 id_real |    nom    | prenom
---------+-----------+---------
       1 | von Trier | Lars
       4 | Tarantino | Quentin
       3 | Eastwood  | Clint
       2 | Parker    | Alan
```

La table `film` contient les lignes suivantes :

```
 id_film | id_real |           titre
---------+---------+---------------------------
       1 |       1 | Dogville
       2 |       1 | Breaking the waves
       3 |       5 | Faux-Semblants
       4 |       5 | Crash
       5 |       3 | Chasseur blanc, coeur noir
```

Exemples de jointures internes

La **jointure naturelle** entre les tables `film` et `réalisateur` peut s'écrire indifféremment de l'une des manières suivante :

```
SELECT * FROM film NATURAL JOIN realisateur
SELECT * FROM film NATURAL INNER JOIN realisateur;
SELECT * FROM film JOIN realisateur USING (id_real);
SELECT * FROM film INNER JOIN realisateur USING (id_real);
```

pour produire le résultat suivant :

```
 id_real | id_film |           titre            |    nom    | prenom
---------+---------+----------------------------+-----------+--------
       1 |       1 | Dogville                   | von Trier | Lars
       1 |       2 | Breaking the waves         | von Trier | Lars
       3 |       5 | Chasseur blanc, coeur noir | Eastwood  | Clint
```

Nous aurions également pu effectuer une **équi-jointure** en écrivant :

```
SELECT * FROM film, realisateur WHERE film.id_real = realisateur.id_real;
SELECT * FROM film JOIN realisateur ON film.id_real = realisateur.id_real;
SELECT * FROM film INNER JOIN realisateur ON film.id_real = realisateur.id_real;
```

Mais la colonne `id_real` aurait été dupliquée :

```
 id_film | id_real |           titre            | id_real |    nom    | prenom
---------+---------+----------------------------+---------+-----------+--------
       1 |       1 | Dogville                   |       1 | von Trier | Lars
       2 |       1 | Breaking the waves         |       1 | von Trier | Lars
       5 |       3 | Chasseur blanc, coeur noir |       3 | Eastwood  | Clint
```

Exemples de jointures externes gauches

La jointure externe gauche entre les tables `film` et `réalisateur` permet de conserver, dans la table résultat, une trace des films dont le réalisateur n'apparaît pas dans la table `realisateur`. Une telle jointure peut s'écrire indifféremment comme suit :

```
SELECT * FROM film NATURAL LEFT JOIN realisateur;
SELECT * FROM film NATURAL LEFT OUTER JOIN realisateur;
SELECT * FROM film LEFT JOIN realisateur USING (id_real);
SELECT * FROM film LEFT OUTER JOIN realisateur USING (id_real);
```

Elle produit le résultat suivant :

```
 id_real | id_film |           titre            |    nom    | prenom
---------+---------+----------------------------+-----------+--------
       1 |       1 | Dogville                   | von Trier | Lars
       1 |       2 | Breaking the waves         | von Trier | Lars
       5 |       3 | Faux-Semblants             |           |
       5 |       4 | Crash                      |           |
       3 |       5 | Chasseur blanc, coeur noir | Eastwood  | Clint
```

Naturellement, en écrivant :

```
SELECT * FROM film LEFT JOIN realisateur ON film.id_real = realisateur.id_real;
SELECT * FROM film LEFT OUTER JOIN realisateur ON film.id_real =
realisateur.id_real;
```

la colonne `id_real` serait dupliquée :

```
 id_film | id_real |           titre            | id_real |    nom    | prenom
---------+---------+----------------------------+---------+-----------+--------
       1 |       1 | Dogville                   |       1 | von Trier | Lars
       2 |       1 | Breaking the waves         |       1 | von Trier | Lars
       3 |       5 | Faux-Semblants             |         |           |
       4 |       5 | Crash                      |         |           |
       5 |       3 | Chasseur blanc, coeur noir |       3 | Eastwood  | Clint
```

Exemples de jointures externes droites

La jointure externe droite entre les tables `film` et `réalisateur` permet de conserver, dans la table résultat, une trace des réalisateurs dont aucun film n'apparaît dans la table `film`. Une telle jointure peut s'écrire indifféremment comme suit :

```
SELECT * FROM film NATURAL RIGHT JOIN realisateur;
SELECT * FROM film NATURAL RIGHT OUTER JOIN realisateur;
SELECT * FROM film RIGHT JOIN realisateur USING (id_real);
SELECT * FROM film RIGHT OUTER JOIN realisateur USING (id_real);
```

Elle produit le résultat suivant :

```
 id_real | id_film |           titre            |    nom    | prenom
---------+---------+----------------------------+-----------+---------
       1 |       1 | Dogville                   | von Trier | Lars
       1 |       2 | Breaking the waves         | von Trier | Lars
       2 |         |                            | Parker    | Alan
       3 |       5 | Chasseur blanc, coeur noir | Eastwood  | Clint
       4 |         |                            | Tarantino | Quentin
```

Exemples de jointures externes bilatérales

La jointure externe bilatérale entre les tables `film` et `réalisateur` permet de conserver, dans la table résultat, une trace de tous les réalisateurs et de tous les films. Une telle jointure peut indifféremment s'écrire :

```
SELECT * FROM film NATURAL FULL JOIN realisateur;
SELECT * FROM film NATURAL FULL OUTER JOIN realisateur;
SELECT * FROM film FULL JOIN realisateur USING (id_real);
SELECT * FROM film FULL OUTER JOIN realisateur USING (id_real);
```

Elle produit le résultat suivant :

```
 id_real | id_film |           titre            |    nom    | prenom
---------+---------+----------------------------+-----------+---------
       1 |       1 | Dogville                   | von Trier | Lars
       1 |       2 | Breaking the waves         | von Trier | Lars
       2 |         |                            | Parker    | Alan
       3 |       5 | Chasseur blanc, coeur noir | Eastwood  | Clint
       4 |         |                            | Tarantino | Quentin
       5 |       3 | Faux-Semblants             |           |
       5 |       4 | Crash                      |           |
```

3.6.2 Les clauses `GROUP BY` et `HAVING` et les fonctions d'agrégation

Notion de groupe

Un groupe est un sous-ensemble des lignes d'une table ayant la même valeur pour un attribut. Par exemple, on peut grouper les films en fonction de leur réalisateur. Un groupe est déterminé par la clause `GROUP BY` suivie du nom du ou des attributs sur lesquels s'effectuent le regroupement.

Syntaxe

La syntaxe d'une requête faisant éventuellement intervenir des fonctions d'agrégation, une clause `GROUP BY` et une clause `HAVING` est la suivante :

```
SELECT expression_1, [...,] expression_N [, fonction_agrégation [, ...] ]
  FROM nom_table [ [ AS ] alias ] [, ...]
  [ WHERE prédicat ]
  [ GROUP BY expression_1, [...,] expression_N ]
  [ HAVING condition_regroupement ]
```

La clause `GROUP BY`

La commande `GROUP BY` permet de définir des regroupements (*i.e.* des agrégats) qui sont projetés dans la table résultat (un regroupement correspond à une ligne) et d'effectuer des calculs statistiques, définis par les expressions `fonction_agrégation [, ...]`, pour chacun des regroupements. La liste d'expressions `expression_1, [...,] expression_N` correspond généralement à une liste de colonnes `colonne_1, [...,] colonne_N`. La liste de colonnes spécifiée derrière la commande `SELECT` (`expression_1, [...,] expression_N`) doit être identique à la liste de colonnes de regroupement spécifiée derrière la commande `GROUP BY` (`expression_1, [...,] expression_N`). A la place des noms de colonne il est possible de spécifier des opérations mathématiques de base sur les colonnes (comme définies dans la section 3.5.4). Dans ce cas, les regroupements doivent porter sur les mêmes expressions.

Si les regroupements sont effectués selon une expression unique, les groupes sont définis par les ensembles de lignes pour lesquelles cette expression prend la même valeur. Si plusieurs expressions sont spécifiées (`expression_1`, `expression_2`, …) les groupes sont définis de la façon suivante : parmi toutes les lignes pour lesquelles `expression_1` prend la même valeur, on regroupe celles ayant `expression_2` identique, etc.

Un `SELECT` avec une clause `GROUP BY` produit une table résultat comportant une ligne pour chaque groupe.

Les fonctions d'agrégation

`AVG( [ DISTINCT | ALL ] expression ):`

Calcule la moyenne des valeurs de l'expression `expression`.

`COUNT( * | [DISTINCT | ALL] expression ):`

Dénombre le nombre de lignes du groupe. Si `expression` est présent, on ne compte que les lignes pour lesquelles cette expression n'est pas `NULL`.

`MAX( [ DISTINCT | ALL ] expression ):`
Retourne la plus petite des valeurs de l'expression `expression`.

`MIN([ DISTINCT | ALL ] expression ):`
Retourne la plus grande des valeurs de l'expression `expression`.

`STDDEV([ DISTINCT | ALL ] expression):`
Calcule l'écart-type des valeurs de l'expression `expression`.

`SUM([ DISTINCT | ALL ] expression):`
Calcule la somme des valeurs de l'expression `expression`.

`VARIANCE([ DISTINCT | ALL ] expression):`
Calcule la variance des valeurs de l'expression `expression`.

`DISTINCT` indique à la fonction de groupe de ne prendre en compte que des valeurs distinctes. `ALL` indique à la fonction de groupe de prendre en compte toutes les valeurs, c'est la valeur par défaut.

Aucune des fonctions de groupe ne tient compte des valeurs `NULL` à l'exception de `COUNT(*)`. Ainsi, `SUM(col)` est la somme des valeurs non `NULL` de la colonne `col`. De même `AVG` est la somme des valeurs non `NULL` divisée par le nombre de valeurs non `NULL`.

Il est tout à fait possible d'utiliser des fonctions d'agrégation sans clause `GROUP BY`. Dans ce cas, la clause `SELECT` ne doit comporter que des fonctions d'agrégation et aucun nom de colonne. Le résultat d'une telle requête ne contient qu'une ligne.

Exemples

Soit la base de données dont le schéma relationnel est :

- film (num_film, num_realisateur, titre, genre, annee)
- cinema (num_cinema, nom, adresse)
- individu (num_individu, nom prenom)
- jouer (num_acteur, num_film, role)
- projection (num_cinema, num_film, jour)

Pour connaître le nombre de fois que chacun des films a été projeté on utilise la requête :

```
SELECT num_film, titre, COUNT(*)
  FROM film NATURAL JOIN projection
  GROUP BY num_film, titre;
```

Si l'on veut également connaître la date de la première et de la dernière projection, on utilise :

```
SELECT num_film, titre, COUNT(*), MIN(jour), MAX(jour)
  FROM film NATURAL JOIN projection
  GROUP BY num_film, titre;
```

Pour connaître le nombre total de films projetés au cinéma `Le Fontenelle`, ainsi que la date de la première et de la dernière projection dans ce cinéma, la requête ne contient pas de clause `GROUP BY` mais elle contient des fonctions d'agrégation :

```
SELECT COUNT(*), MIN(jour), MAX(jour)
  FROM film NATURAL JOIN projection NATURAL JOIN cinema
  WHERE cinema.nom = 'Le Fontenelle';
```

La clause `HAVING`

De la même façon qu'il est possible de sélectionner certaines lignes au moyen de la clause `WHERE`, il est possible, dans un `SELECT` comportant une fonction de groupe, de sélectionner certains groupes par la clause `HAVING`. Celle-ci se place après la clause `GROUP BY`.

Le prédicat dans la clause `HAVING` suit les mêmes règles de syntaxe qu'un prédicat figurant dans une clause `WHERE`. Cependant, il ne peut porter que sur des caractéristiques du groupe : fonction d'agrégation ou expression figurant dans la clause `GROUP BY`.

Une requête de groupe (*i.e.* comportant une clause `GROUP BY`) peut contenir à la fois une clause `WHERE` et une clause `HAVING`. La clause `WHERE` sera d'abord appliquée pour sélectionner les lignes, puis les groupes seront constitués à partir des lignes sélectionnées, les fonctions de groupe seront ensuite évaluées et la clause `HAVING` sera enfin appliquée pour sélectionner les groupes.

Exemples

Pour connaître le nombre de fois que chacun des films a été projeté en ne s'intéressant qu'aux films projetés plus de 2 fois, on utilise la requête :

```
SELECT num_film, titre, COUNT(*)
  FROM film NATURAL JOIN projection
  GROUP BY num_film, titre
  HAVING COUNT(*)>2;
```

Si en plus, on ne s'intéresse qu'aux films projetés au cinéma `Le Fontenelle`, il faut ajouter une clause `WHERE` :

```
SELECT num_film, titre, COUNT(*)
  FROM film NATURAL JOIN projection NATURAL JOIN cinema
  WHERE cinema.nom = 'Le Fontenelle'
  GROUP BY num_film, titre
  HAVING COUNT(*)>2;
```

3.6.3 Opérateurs ensemblistes : `UNION`, `INTERSECT` et `EXCEPT`

Les résultats de deux requêtes peuvent être combinés en utilisant les opérateurs ensemblistes d'*union* (`UNION`), d'*intersection* (`INTERSECT`) et de *différence* (`EXCEPT`). La syntaxe d'une telle requête est la suivante :

```
requête_1 { UNION | INTERSECT | EXCEPT } [ALL] requête_2 [...]
```

Pour que l'opération ensembliste soit possible, il faut que `requête_1` et `requête_2` aient le même schéma, c'est à dire le même nombre de colonnes respectivement du même type. Les noms de colonnes (titres) sont ceux de la première requête (`requête_1`).

Il est tout à fait possible de chaîner plusieurs opérations ensemblistes. Dans ce cas, l'expression est évaluée de gauche à droite, mais on peut modifier l'ordre d'évaluation en utilisant des parenthèses.

Dans une requête on ne peut trouver qu'une seule instruction `ORDER BY`. Si elle est présente, elle doit être placée dans la dernière requête (cf. section 3.5.6). La clause `ORDER BY` ne peut faire référence qu'aux numéros des colonnes (la première portant le numéro 1), et non pas à leurs noms, car les noms peuvent être différents dans chacune des requêtes sur lesquelles porte le ou les opérateurs ensemblistes.

Les opérateurs `UNION` et `INTERSECT` sont commutatifs.

Contrairement à la commande `SELECT`, le comportement par défaut des opérateurs ensemblistes élimine les doublons. Pour les conserver, il faut utiliser le mot-clef `ALL`.

Attention, il s'agit bien d'opérateurs portant sur des tables générées par des requêtes. On ne peut pas faire directement l'union de deux tables de la base de données.

3.6.4 Traduction des opérateurs d'union, d'intersection, de différence et de division de l'algèbre relationnelle (2 ème partie)

Traduction de l'opérateur d'union

L'opérateur d'union $relation_1 \cup relation_2$ se traduit tout simplement en SQL par la requête :

```
SELECT * FROM relation_1 UNION SELECT * FROM relation_2
```

Traduction de l'opérateur d'intersection

L'opérateur d'intersection $R_1 \cap R_2$ se traduit tout simplement en SQL par la requête :

```
SELECT * FROM relation_1 INTERSECT SELECT * FROM relation_2
```

Traduction de l'opérateur de différence

L'opérateur de différence $R_1 - R_2$ se traduit tout simplement en SQL par la requête :

```
SELECT * FROM relation_1 EXCEPT SELECT * FROM relation_2
```

Traduction de l'opérateur de division

Il n'existe pas de commande SQL permettant de réaliser directement une division. Prenons la requête :

Quels sont les acteurs qui ont joué dans tous les films de Lars von Trier ?

Cela peut se reformuler par :

Quels sont les acteurs qui vérifient : quel que soit un film de Lars von Trier, l'acteur a joué dans ce film.

Malheureusement, le quantificateur universel (∀) n'existe pas en SQL. Par contre, le quantificateur existentiel (∃) existe : `EXISTS`. Or, la logique des prédicats nous donne l'équivalence suivante :

$\forall\, x\, P(x) = \neg\, \exists\, x\, \neg\, P(x)$

On peut donc reformuler le problème de la manière suivante :

Quels sont les acteurs qui vérifient : il est faux qu'il existe un film de Lars von Trier dans lequel l'acteur n'a pas joué.

Ce qui correspond à la requête SQL :

```
SELECT DISTINCT nom, prenom FROM individu AS acteur_tous_lars
WHERE NOT EXISTS (
  SELECT * FROM ( film JOIN individu ON num_realisateur = num_individu
                  AND nom = 'von Trier' AND prenom = 'Lars' ) AS film_lars
  WHERE NOT EXISTS (
    SELECT * FROM individu JOIN jouer ON num_individu = num_acteur
          AND num_individu = acteur_tous_lars.num_individu
          AND num_film = film_lars.num_film
  )
);
```

En prenant le problème d'un autre point de vue, on peut le reformuler de la manière suivante :

Quels sont les acteurs qui vérifient : le nombre de films réalisés par Lars von Trier dans lequel l'acteur à joué est égal au nombre de films réalisés par Lars von Trier.

Ce qui peut se traduire en SQL indifféremment par l'une des deux requêtes suivantes :

```
SELECT acteur.nom, acteur.prenom
FROM individu AS acteur JOIN jouer ON acteur.num_individu = jouer.num_acteur
    JOIN film ON jouer.num_film = film.num_film
    JOIN    individu    AS    realisateur    ON    film.num_realisateur    =
realisateur.num_individu
WHERE realisateur.nom = 'von Trier' AND realisateur.prenom = 'Lars'
GROUP BY acteur.nom, acteur.prenom
HAVING COUNT (DISTINCT film.num_film) = (
  SELECT DISTINCT COUNT(*)
  FROM film JOIN individu ON num_realisateur = num_individu
  WHERE nom = 'von Trier' AND prenom = 'Lars'
);
```

```
SELECT DISTINCT acteur_tous_lars.nom, acteur_tous_lars.prenom
FROM individu AS acteur_tous_lars
WHERE (
  SELECT DISTINCT COUNT(*)
  FROM jouer JOIN film ON jouer.num_film = film.num_film
    JOIN individu ON num_realisateur = num_individu
  WHERE nom = 'von Trier' AND prenom = 'Lars'
    AND jouer.num_acteur = acteur_tous_lars.num_individu
) = (
  SELECT DISTINCT COUNT(*)
  FROM film JOIN individu ON num_realisateur = num_individu
  WHERE nom = 'von Trier' AND prenom = 'Lars'
);
```