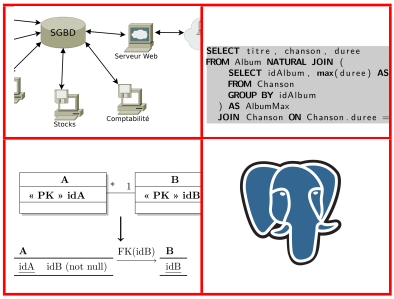
# INTRODUCTION AUX BASES DE DONNÉES

DUT Informatique – M1104
Promotion 2015–2017

Julien Vion – julien.vion@univ-valenciennes.fr




UVHC
UNIVERSITÉ DE VALENCIENNES ET DU HAINAUT - CAMBRESIS

IUT département informatique

# Introduction

Ce cours se veut être un *manuel de référence* que vous devrez avoir **systématiquement et obligatoirement** sur vous lors de toutes les séances d'apprentissage et d'évaluation. Vous irez y piocher les informations dont vous aurez besoin pour résoudre les problèmes auxquels vous serez confrontés. Contrairement à un cours habituel, *ne vous attendez pas* à une présentation systématique du contenu de ce livret lors des séances de cours. L'ensemble des notions expliquées ici devront être progressivement mises en œuvre lors de la résolution de chaque problème.

Ce livret n'est pas exhaustif. N'hésitez pas à consulter les documents listés dans la section bibliographique si vous avez besoin de plus d'informations.

## Programme

Ce cours correspond au module *Introduction aux bases de données* (M1104) du Programme Pédagogique National du DUT Informatique [5]. L'objectif est de vous former aux compétences suivantes :

### Objectif du module :

- acquérir les connaissances nécessaires pour la manipulation d'une base de données.

### Compétences visées :

- conception technique d'une solution informatique ;
- réalisation d'une solution informatique ;
- tests de validation d'une solution informatique.

## Contenus :

- le modèle relationnel (concepts, contraintes d'intégrité, dépendances fonctionnelles) ;
- algèbre relationnelle ;
- *SQL* (*Structured Query Language*) : langage de manipulation des données, langage de définition des données ;
- approche de la conception des bases de données : modèle conceptuel de données et traduction vers le modèle relationnel ;
- éléments sur les tests de requêtes.

## Modalités de mise en œuvre :

Pour la mise en œuvre pratique de ce module, nous nous appuierons sur le logiciel *PostgreSQL* [6].

# Organisation des séances

La formation est organisée sous forme de *problèmes* sur lesquels vous travaillerez lors des séances de TD et TP. L'ensemble des sujets des problèmes que vous aurez à résoudre au cours de la formation se trouve en annexe de ce document. Généralement, la résolution d'un problème suivra le schéma suivant :

1. Séance de TD (1 heure 30) : étude du sujet *en groupe de 3 ou 4*. Préparation d'une solution.
2. Séance de TP (3 heures) : réalisation *individuelle* de la solution élaborée lors de la séance de groupe.
3. Séance de TD (1 heure 30) :
   - mise en commun des réalisations individuelles et préparation d'un compte-rendu *collectif par groupe*, rédaction de questions à destination de l'enseignant ;
   - présentation de la solution du groupe aux autres groupes présents lors de la séance.
4. Séance de cours (1 heure 30) :
   - restructuration et réponse aux questions ;
   - éventuellement une évaluation *individuelle*.

Soit 7 heures 30 en séance par problème. Entre chaque séance : travail autonome (non encadré), individuel et/ou en groupe, pour compléter les préparations et travaux non terminés (les évaluations sont faites en considérant que les problèmes sont entièrement résolus) et acquérir les connaissances nécessaires à la résolution des problèmes (normalement d'après ce livret).

Le module comporte 60 heures de travail encadré séquencées comme suit :

- Un cours introductif (1h30) ;
- 7 séquences de problème (52h30 au total) ;
- un TP contrôle (3h) ;
- un DS final (1h30).

Notez que le module *Programmation et administration des bases de données* (M2106), d'une durée de 45 heures, complète cette formation au deuxième semestre.

L'annexe A de ce livret donne des indications sur les méthodes de travail. Consultez-la avant le début du premier problème.

# Évaluation

L'évaluation est répartie de cette manière :

**Évaluation pratique (TP contrôle)** (coefficient 2)

**Évaluation théorique** (coefficient 3), subdivisée comme suit :

**évaluations individuelles** : coefficient **1** ;

**devoir surveillé** : coefficient **2**.

# Liste des Acronymes

*ANSI* ..... *American National Standards Institute*

*API* ..... *Application Programming Interface*

APP ..... Apprentissage par Problèmes

BD ..... Base de Données

BDR ..... Base de Données Relationnelle

*DB* ..... *Database*

*DBMS* ..... *Database Management System*

E/A ..... Entité/Association

IHM ..... Interface Homme-Machine

*ISO* ..... *International Organisation for Standardization*

*JDBC* ..... *Java Database Connectivity*

MCD ..... Modèle Conceptuel de Données

*OCL* ..... *Object Constraint Language*

*ODBC* ..... *Open Database Connectivity*

*PHP* ..... *PHP : Hypertext Preprocessor*

*RDB* ..... *Relational Database*

*RDBMS* ..... *Relational Database Management System*

SGBD ..... Système de Gestion de Bases de Données

SGBDR..... Système de Gestion de Bases de Données Relationnelles

*SQL*..... *Structured Query Language*

UF..... Unité de Formation

*UML*..... *Unified Modelling Language*

# Table des matières

**Introduction** **iii**

**Liste des Acronymes** **viii**

**1 Principes et administration des SGBD** **1**
1.1 Qu'est-ce qu'une Base de Données ? . . . . . 2
1.2 Le langage SQL . . . . . 2
1.3 Qu'est-ce qu'un SGBD ? . . . . . 4
1.4 Objectifs des SGBD . . . . . 5
1.5 Fonctions des SGBD . . . . . 7
1.6 *PostgreSQL* 9.4 : prise en main . . . . . 8

**2 Modèle de données relationnel** **21**
2.1 Le modèle relationnel . . . . . 21
2.2 La clé primaire . . . . . 22
2.3 Les clés étrangères . . . . . 23
2.4 Définition des données en SQL . . . . . 23
2.5 Suppression d'une table . . . . . 29
2.6 Faire un script de création de base de données . . . . . 30

**3 Manipulation des données en SQL** **31**
3.1 Modifier des données . . . . . 31
3.2 Obtenir des données . . . . . 33

**4 SQL : l'essentiel** **41**
4.1 Compétences mises en œuvre . . . . . 42
4.2 Introduction . . . . . 42
4.3 Types de données . . . . . 43
4.4 Définition des données . . . . . 44
4.5 Manipulation des données . . . . . 52
4.6 Expressions et opérateurs . . . . . 64

**5 Conception** **69**
5.1 Introduction . . . . . 70
5.2 Pourquoi une méthode de conception ? . . . . . 70
5.3 Modèle conceptuel . . . . . 73
5.4 Modèle physique : le modèle Relationnel . . . . . 81
5.5 Rétro-ingéniérie . . . . . 87

**Bibliographie** **89**

**A Consignes de travail** **91**
A.1 Travailler en groupe . . . . . 91
A.2 Travail individuel . . . . . 92
A.3 Évaluation du travail en groupe . . . . . 93

**B Problème 1** **95**
B.1 Le problème . . . . . 95
B.2 Travail demandé . . . . . 95

**C Problème 2** **99**
C.1 Le problème . . . . . 99
C.2 Travail demandé . . . . . 99

**D Problème 3** **101**
D.1 Le problème . . . . . 101
D.2 Travail demandé . . . . . 101

**E Problème 4** **105**
E.1 Le problème . . . . . 105
E.2 Travail demandé . . . . . 107
E.3 Préparation . . . . . 108
E.4 Réalisation . . . . . 108
E.5 Présentation . . . . . 108

**F Problème 5** **109**
F.1 Le problème . . . . . 109
F.2 Travail demandé . . . . . 109

**G Problème 6** **113**
G.1 Le problème . . . . . 113
G.2 Travail demandé . . . . . 116

**H Problème 7** **117**
H.1 Analyse de l'existant . . . . . 117
H.2 Extension de la base . . . . . 117
H.3 Migration des données . . . . . 119

**Glossaire** **122**

**Index** **123**

Chapitre 1

# Principes et notions d'administration des Systèmes de Gestion de Bases de Données

## Sommaire


**1.1 Qu'est-ce qu'une Base de Données ? . . . . . 2**

**1.2 Le langage SQL . . . . . 2**

**1.3 Qu'est-ce qu'un SGBD ? . . . . . 4**

**1.4 Objectifs des SGBD . . . . . 5**

1.4.1 Administration cohérente des données . . . 6

1.4.2 Indépendance physique des données . . . . . 6

1.4.3 Indépendance logique des données . . . . . 6

1.4.4 Contrôle de la redondance des données . . . 6

1.4.5 Manipulation des données par des non-informaticiens . . . . . 7

**1.5 Fonctions des SGBD . . . . . 7**

1.5.1 Optimisation de l'accès aux données . . . . 7

1.5.2 Contrôle de l'intégrité des données . . . . . 8

1.5.3 Partage simultané des données (contrôle de la concurrence) . . . . . 8

1.5.4 Sécurité des données (confidentialité) . . . . 8

1.5.5 Sureté des données (résistance aux pannes) 8
**1.6 *PostgreSQL* 9.4 : prise en main . . . . . 8**
1.6.1 Démarrage . . . . . 8
1.6.2 Fonctionnement du serveur . . . . . 11
1.6.3 Bases, utilisateurs, tables et droits d'accès . 11
1.6.4 Utilisation de psql . . . . . 12
1.6.5 Créer une table . . . . . 13
1.6.6 Droits d'accès . . . . . 14
1.6.7 Utilisation de pgAdmin . . . . . 14

## 1.1 Qu'est-ce qu'une Base de Données ?

Une BD (Base de Données) – *DB* (*Database*) en anglais – est un ensemble d'informations reliées entre elles de manière cohérente, et stockées de manière permanente dans un système informatique. Par exemple, dans le cadre du système d'information d'un supermarché, on stockera la liste des produits et leurs caractéristiques (code barre, rayon), leur provenance (traçabilité), l'état des stocks, la liste des clients (cartes de fidélité, service après-vente), des employés (paye), etc. Le tableau 1.1 page ci-contre donne un exemple d'une petite BD qui pourrait être utilisée dans le cadre d'un logiciel de *media center* ou pour un site web communautaire de critiques musicales.

La plupart des bases de données modernes se basent sur l'Algèbre Relationnelle, une modélisation mathématique des bases de données inventée en 1970 par Edgar F. Codd [4], récompensé en 1981 par le prix Turing (250 000 $, soient 650 000 $ d'aujourd'hui compte tenu de l'inflation). *Oracle Database* est le premier SGBD (Système de Gestion de Bases de Données) à s'être inspiré des travaux de Codd, et a donné alors naissance à l'ensemble des technologies modernes des bases de données. Ce type de BD est appelé BDR (Base de Données Relationnelle) – *RDB* (*Relational Database*) en anglais. Le principe des bases de données relationnelles est de représenter les données sous formes de *tables* indépendantes, reliées entre elles à l'aide de *clés* (cf chapitre 2). C'est ce type de BD qui sera étudié dans le cadre de ce cours. Pour information, on peut citer l'existence de BD hiérarchiques, BD réseau (deux familles de BD antérieures aux BDR, moins souples et expressives mais parfois plus rapides), BD déductives, BD clés/valeurs et BD orientées objet. Ici, remarquez l'utilisation de *clés* (idArtiste, idAlbum, piste) pour associer les différentes informations. Celles-ci sont au centre des mécanismes des BDR.

## 1.2 Le langage SQL

Le *SQL* est un langage de programmation déclaratif permettant notamment de définir, manipuler et contrôler les données. Il s'agit d'un langage normalisé utilisé par la plupart des SGBD relationnels.

**Artiste**

| idArtiste | nom | image |
|---|---|---|
| | ... | |
| 67 | Noir Désir | pas_dimage.png |
| 68 | Moby | moby_jump.jpg |
| 69 | The Dandy Warhols | banana.jpg |
| 70 | Ghinzu | glive.jpg |
| | ... | |

**Album**

| idAlbum | titre | année | idArtiste* |
|---|---|---|---|
| | ... | | |
| 154 | 666.667 Club | 1996 | 67 |
| 155 | Tostaky | 1992 | 67 |
| 156 | Welcome to the Monkey House | 2003 | 69 |
| | ... | | |

**Chanson**

| piste | idAlbum* | chanson | durée |
|---|---|---|---|
| | | ... | |
| 3 | 156 | Plan A | 241 |
| 4 | 156 | The Dope (Wonderful You) | 277 |
| 3 | 154 | Un jour en France | 193 |
| | | ... | |

TAB. 1.1 : Exemple de base de données

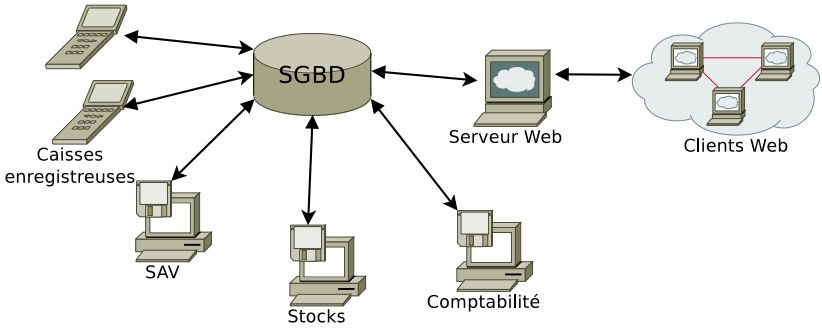



FIG. 1.1 : Le SGBD au sein du système d'information

## 1.3 Qu'est-ce qu'un Système de Gestion de Bases de Données ?

Un SGBD – *DBMS* (*Database Management System*) en anglais – est généralement un « serveur », logiciel fonctionnant en permanence en arrière-plan (« démon » sous Unix/Linux, « service » sous Windows). À partir d'une certaine importance (gestion d'une entreprise, site web commercial…), on peut lui dédier une ou plusieurs machines physiques. Sa fonction est de gérer le stockage et l'accès à de grandes quantités de données de manière :

- centralisée,
- rationalisée,
- unifiée,
- indépendante des logiciels applicatifs.

Ainsi, différents logiciels nécessitant l'accès à ces données (site web, application de gestion, caisse enregistreuse…) pourront *interroger* le serveur à tout moment, récupérer les données dont ils ont besoin, et éventuellement mettre à jour les données stockées. La figure 1.1 illustre un exemple d'organisation d'un système d'information autour du SGBD.

Il existe de nombreux logiciels de SGBD, certains libres et gratuits (*Firebird*, *MySQL*, *PostgreSQL* …), d'autres commerciaux (*IBM DB2*, *Microsoft SQL Server*, *Oracle Database*…). La plupart des SGBD sont basés sur le modèle relationnel, on parle alors de SGBDR (Système de Gestion de Bases de Données Relationnelles) – *RDBMS* (*Relational Database Management System*) en anglais. Les solutions commerciales sont souvent très onéreuses (pas moins de 20 000 € par processeur pour *DB2*, *Oracle* ou *SQL Server Enterprise*, par exemple). Les éditeurs de logiciels proposent généralement des versions bridées, au rapport fonctionnalités/prix plus adapté aux besoins des

entreprises plus modestes, voire même gratuites. Certains SGBD, comme *Microsoft Access* ou *Filemaker*, intègrent un environnement de développement permettant la réalisation rapide d'applications simples. Ils sont particulièrement adaptés aux non-informaticiens ou au prototypage d'applications. Finalement, il existe des moteurs de SGBD destinés à être directement intégrés (embarqués) dans des applications, sans nécessiter l'installation et la configuration d'un serveur (*Apache Derby*, *SQLite*...). De tels systèmes peuvent être utilisés dans toute application nécessitant une persistance des données (du logiciel de gestion au *media center*).

Bien que le langage SQL, utilisé pour dialoguer avec les SGBDR, soit normalisé, chaque logiciel peut utiliser une syntaxe légèrement différente (par exemple, pour générer une clé automatiquement, *MySQL* propose l'option **auto_increment** là où *PostgreSQL* utilise le type-macro **serial** ). Les fonctionnalités supportées, les performances (en fonction de la quantité de données à gérer), les méthodes d'installation et d'administration... sont également très variables.

Le choix d'un SGBD est une tâche difficile, qui dépend de nombreux paramètres (budget, fonctionnalités requises, perspectives de montée en charge, services et garanties du fournisseur, etc.). Il existe relativement peu de comparatifs complets, et encore moins de comparatifs objectifs. On peut toutefois citer la page de F. Celaia sur le site `developpez.com` [3]. Bien qu'il ne s'agisse pas nécessairement du SGBD le plus populaire[1], nous utiliserons le SGBDR *PostgreSQL* version 9.4 [6] comme référence dans le cadre de ce cours : il s'agit à l'heure actuelle du SGBD libre et gratuit supportant le plus grand nombre de fonctionnalités – il n'est toutefois pas pour autant le plus adapté dans toutes les situations. La version 9.4 est la version distribuée avec le système d'exploitation GNU/Linux *Debian 8.2 (« Jessie »)*.

## 1.4 Objectifs des Systèmes de Gestion de Bases de Données

Cette section récapitule les différents besoins des utilisateurs auxquels les SGBD apportent une solution.

[1] Le SGBD libre le plus souvent déployé, en particulier pour la réalisation de sites Web à trafic modeste, est sans aucun doute *MySQL*, très simple d'utilisation et assez rapide, mais un peu pauvre au niveau des fonctionnalités. Le SGBD le plus simple d'accès est probablement *MS Access*, mais il n'est pas vraiment conçu pour centraliser l'information (pas de client/serveur, capacités de montée en charge très limitées), ni pour interagir avec d'autres langages de programmation que le *VBA*. Finalement, le SGBDR le plus souvent déployé pour de grosses applications industrielles est *Oracle*, très performant et riche en fonctionnalités, mais onéreux et difficile à administrer (c'est un métier à part entière).

### 1.4.1 Administration cohérente des données

En informatique de gestion, le SGBD permet d'englober la totalité de l'information de l'entreprise. Par une modélisation efficace, cohérente et rigoureuse des données, au plus proche du fonctionnement réel de l'entreprise, il permet un accès simple aux données et une meilleure circulation de l'information. La modélisation peut se faire de manière incrémentale, un service de l'entreprise après l'autre. Une telle approche apporte une compréhension profonde des données de l'entreprise, et permet la mise en place de traitements complexes.

### 1.4.2 Indépendance physique des données

Le SGBD et son langage de requête constituent les uniques points d'entrée aux données. Le stockage physique des données (disques durs, fichiers, index, clusters, sauvegardes…) reste totalement transparent du point de vue de l'utilisateur. Les problématiques purement informatiques (algorithmes de recherche, de tri, manipulation de fichiers de grande taille) sont déléguées au SGBD. Seule une représentation à haut niveau, abstraite, sous forme de classes d'objets, est nécessaire.

Par exemple, suite à la mise à jour matérielle du serveur de SGBD, l'intégralité des données d'une PME peut être stockée en mémoire vive. Le SGBD prend en compte cette nouvelle structure et adapte ses algorithmes de gestion des données en conséquence, augmentant alors considérablement les performances de l'application.

Pour améliorer encore l'indépendance entre le SGBD et l'application, on adopte généralement un modèle de développement à trois couches (MVC pour Modèle, Vue, Contrôleur) : le SGBD au niveau inférieur (couche Modèle), puis une couche dite « métier », contenant les traitements spécifiques à l'application (Vue), et enfin une couche « présentation » – IHM (Interface Homme-Machine) (Contrôleur).

### 1.4.3 Indépendance logique des données

La notion de *Vues* permet à chaque application d'ignorer les besoins des autres applications tout en partageant la même BD. Par exemple, à partir de la base donnée en exemple en page 3, on peut générer la vue présentée en tableau 1.2 page ci-contre, afin de simplifier le développement d'une application ne nécessitant pas de référencer les artistes.

Ce mécanisme peut également permettre la coexistence plus facile d'applications nouvelles et anciennes, et de limiter l'accès aux données suivant les utilisateurs.

### 1.4.4 Contrôle de la redondance des données

Une redondance des données peut être ou non souhaitée suivant les besoins. Minimiser la redondance permet de limiter les tâches de saisie, les

**AlbumArtiste**

| idAlbum | titre | année | artiste |
|---|---|---|---|
| | | ... | |
| 154 | 666.667 Club | 1996 | Noir Désir |
| 155 | Tostaky | 1992 | Noir Désir |
| 156 | Welcome to the... | 2003 | The Dandy Warhols |
| | | ... | |

TAB. 1.2 : Exemple de vue obtenue à partir de la BD présentée en table 1.1.

risques d'erreur, et de simplifier considérablement le maintien de la cohérence des données entre plusieurs applications. Cependant, on peut souhaiter une certaine forme de redondance pour améliorer la fiabilité et les performances des applications. Le SGBD permet de contrôler au mieux la réplication des informations.

### 1.4.5 Manipulation des données par des non-informaticiens

Les non-informaticiens doivent pouvoir manipuler les données à partir de leur connaissance du monde réel et de la modélisation. Les langages utilisés pour interagir avec les SGBD sont généralement des langages déclaratifs (on indique ce que l'on veut, et non pas comment l'obtenir), proches du langage naturel. Il reste cependant possible d'intégrer ce langage dans des applications réalisées avec des outils plus généraux (Java, PHP, .Net...), et même de stocker des procédures en langage impératif directement dans le SGBD.

## 1.5 Fonctions des Systèmes de Gestion de Bases de Données

Cette section récapitule les différentes fonctions des SGBD, utilisées pour répondre aux besoins des utilisateurs.

### 1.5.1 Optimisation de l'accès aux données

Seul le SGBD a accès aux fichiers contenant les données. Il est capable d'optimiser automatiquement les requêtes des utilisateurs pour minimiser les besoins d'accès aux disques (cache en RAM, compilation des requêtes plus fréquentes, etc.) De quoi économiser l'écriture de milliers de lignes de code d'optimisation dans les logiciels applicatifs.

### 1.5.2 Contrôle de l'intégrité des données

Au moment de la création de la base de données, on définit des *contraintes d'intégrité*, utilisées pour contrôler en temps réel la validité des données. Par exemple, on peut définir qu'un nom propre ne doit contenir que des caractères alphabétiques, qu'un salaire doit avoir un minimum et un maximum, qu'un prix de vente doit être supérieur au prix d'achat, etc. Les contraintes d'intégrité sont également utilisées pour contrôler les relations entre les données, on parle alors du *contrôle de l'intégrité référentielle*. Dans le cas de la base présentée page 3, on interdira par exemple de définir un album sans que l'artiste ne soit référencé dans la base de données.

### 1.5.3 Partage simultané des données (contrôle de la concurrence)

Si plusieurs applications fonctionnent en parallèle tout en partageant des données, on peut rencontrer très vite des risques de conflits lors d'accès simultanés à l'information : que faire quand une application modifie des données à l'instant même où une autre application tente d'accéder à celles-ci ? Les SGBD, quand ils fonctionnent en mode « serveur » gèrent naturellement les accès concurrents. Ils disposent généralement de fonctions destinées à simplifier au maximum la gestion des cas de concurrence aux développeurs d'applications.

### 1.5.4 Sécurité des données (confidentialité)

Les SGBD disposent de mécanismes de gestion des droits d'accès aux données, afin de garantir la confidentialité de certaines informations (comptes bancaires, dossiers médicaux, mots de passe, etc.)

### 1.5.5 Sureté des données (résistance aux pannes)

En cas de panne, l'intégrité des données doit être préservée. En général, cela passe par un système de sauvegarde et/ou de redondance physique. D'autre part, si une panne survient au cours d'un traitement, le SGBD doit rétablir les données dans un état cohérent et défini.

## 1.6 *PostgreSQL* 9.4 : prise en main

### 1.6.1 Démarrage

Cette section ne concerne pas les TP réalisés sur les machines fournies par l'IUT. Ici, un serveur unique est accessible depuis le réseau. Son nom d'hôte est `iutinfo-sgbd.univ-valenciennes.fr`. Il est possible de s'y connecter directement avec les clients psql et pgAdmin installés sur les machines de TP

en utilisant les login/mot de passe fournis par l'enseignant lors du premier TP. Vous pouvez passer immédiatement à la section 1.6.2 page 11.

### Installation

*PostgreSQL* est disponible pour les systèmes d'exploitation *MS Windows*, *Sun Solaris*, *Mac OS X*, *FreeBSD* et *GNU/Linux* sur la page http://www.postgresql.org/download/. Sous *Windows* et *Mac OS X*, il existe des installateurs « un-clic ». Il existe également des distributions *WAPP* et *MAPP* (respectivement pour Windows et Mac OS X) qui installent simultanément le serveur web *Apache*, *PostgreSQL* et l'interpréteur du langage *PHP*. Sous Linux, il est préférable de passer par le système de paquetage de la distribution (*apt* sous Ubuntu ou Debian, *rpm* sous RedHat, Fedora ou Mandriva, etc.) Par exemple, pour installer *PostgreSQL* sous Ubuntu, on utilise la commande :

```
sudo apt-get install postgresql
```

*PostgreSQL* est un serveur, c'est-à-dire qu'il fonctionne en tâche de fond (« service » sous Windows, « démon » sous Unix/Linux). Si vous avez installé *PostgreSQL* grâce au système de paquetage de votre distribution Linux, celui-ci est probablement déjà configuré pour être lancé automatiquement au démarrage de votre machine. Vous pouvez vous en assurer en utilisant la commande `ps aux | grep postgres` (notez que *PostgreSQL* lance plusieurs processus).

À partir d'ici, on considèrera que *PostgreSQL* est installé à partir du système de paquetage d'une distribution Linux.

### Connexion administrateur

Sous Linux, `psql` réalise une association entre les utilisateurs de *PostgreSQL* et les utilisateurs Unix. Il va donc falloir vous créer un utilisateur *PostgreSQL* du même nom que votre utilisateur Unix pour vous connecter facilement à la base de données. À l'installation de *PostgreSQL*, un compte d'administration (nommé généralement *postgres*) est créé. Il est actuellement le seul autorisé à se connecter à la base de données. Il s'agit d'un compte Linux *sans mot de passe*, c'est-à-dire qu'il est impossible de se connecter directement en tant que *postgres*. Il faut passer par l'administrateur de la machine Linux (*root*) :

```
sudo su postgres -c psql
```

La commande `sudo` (*Superuser do*) permet de devenir *root* (entrer le mot de passe approprié), et la commande `su` (*Switch User*) permet alors de devenir *postgres* (comme on est alors *root*, le mot de passe n'est pas nécessaire). Enfin, le client `psql` est lancé (cf section 1.6.4 page 12).

`postgres` est une base de donnée créée automatiquement lors de l'installation de *PostgreSQL*, qui sert à se connecter au serveur pour la première fois (*bootstrapping*). Elle est vide et a vocation à le rester.

### Créer un utilisateur

La première étape consiste à créer un utilisateur que vous utiliserez pour vous connecter à la base de données. Par sécurité, il est fortement recommandé de ne *pas* faire de cet utilisateur un autre administrateur (*superuser*). Il est plus pratique de choisir le même login pour l'accès à la base de données que pour l'accès à votre compte Linux (ainsi, vous n'aurez pas besoin de saisir un login et mot de passe à chaque accès à *PostgreSQL*).

Pour ce faire, il faut tout d'abord lancer le client interactif psql en tant qu'administrateur de *PostgreSQL*, c'est-à-dire l'utilisateur *postgres*. Une fois connecté avec le client interactif, on peut créer un utilisateur avec la commande SQL :

```
CREATE USER login [ options ]
```

*options* peut comprendre notamment **CREATEDB** pour autoriser l'utilisateur à créer des bases de données (recommandé pour les développeurs) et PASSWORD '*password*' pour définir un mot de passe (indispensable pour se connecter depuis le réseau). D'autres options sont disponibles : consultez la documentation officielle [7]. Pour les développeurs, il est conseillé de choisir le même login pour *PostgreSQL* que pour le compte Linux : cela simplifie l'authentification (pas besoin de répéter le mot de passe pour une connexion locale par le socket Unix).

Par la suite, il vous sera possible de modifier un utilisateur (augmenter ou réduire ses droits, changer de mot de passe) ou de le supprimer via les commandes **ALTER USER** et **DROP USER**. Par exemple :

```
DROP USER test;
ALTER USER vion PASSWORD 'p455w0rD';
ALTER USER vion CREATEDB;
```

Ces trois commandes, respectivement, supprime l'utilisateur *test*, modifie le mot de passe de l'utilisateur *vion* et enfin autorise *vion* à créer des bases de données.

### Créer une base

Déconnectez-vous du compte *postgres* (Ctrl-D) et reconnectez-vous sous votre propre nom d'utilisateur. Comme vous devez obligatoirement vous connecter à une base, vous pouvez pour le moment utiliser la base *postgres* :

```
psql postgres
```

Vous pouvez alors créer la base de données avec la commande :

```
CREATE DATABASE tuto [ options ]
```

Choisissez un nom de base approprié (*tuto* dans l'exemple). *options* peut notamment permettre de spécifier un propriétaire à la base de données (avec **OWNER** login). Par défaut, c'est l'utilisateur qui crée la base qui en devient le propriétaire.

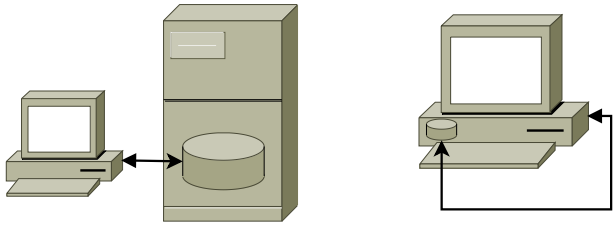


FIG. 1.2 : Connexion au SGBD fonctionnant sur un serveur physique par le réseau (à gauche) ou sur la même machine *via* le *loopback* (à droite)

### 1.6.2 Fonctionnement du serveur

*PostgreSQL* est un *serveur*, il fonctionne en tâche de fond (*daemon* Unix ou *service* Windows). Pour interagir avec lui, on utilise un *client* pour s'y connecter par le réseau. Il est possible de se connecter au serveur à partir d'un client situé sur la même machine physique en passant par l'interface réseau *loopback* (bouclage) ou par un *socket Unix*. La figure 1.2 illustre les deux comportements.

Le client peut être n'importe quelle application, programmée pour se connecter au serveur et dialoguer avec lui dans un langage qu'il comprend. Dans des applications réelles, on passe généralement par des *API* (*Application Programming Interface*) comme *ODBC* ou *JDBC*, ou encore par une extension du langage de programmation (c'est le cas en *PHP*).

*PostgreSQL* est fourni avec un client interactif. Il permet à l'utilisateur de saisir directement ses requêtes, ou de charger des fichiers contenant un ensemble de requêtes, qui sont alors immédiatement exécutées par le serveur. Ce client interactif, exécuté par la commande psql, servira de référence pour la prise en main de *PostgreSQL*.

### 1.6.3 Bases, utilisateurs, tables et droits d'accès

Un serveur *PostgreSQL* est capable de gérer plusieurs BD simultanément. Une BD est un ensemble clos de données, constituée essentiellement de *relations* (ou *tables*). Pour dialoguer avec le serveur, il faut obligatoirement indiquer une BD de travail. Il existe une BD par défaut du nom de *postgres*, qui peut servir de point d'entrée pour créer des utilisateurs et d'autres BD.

*PostgreSQL* gère également la notion d'*utilisateur* et de *groupes d'utilisateurs* (*rôle* en terminologie *PostgreSQL*) et de *droits d'accès*. L'un des utilisateurs est *administrateur* du SGBD, et est autorisé à accéder à et modifier l'ensemble des données et des paramètres des BD gérées. Par défaut, il est le seul autorisé à créer des bases et des utilisateurs. L'administrateur est généralement l'utilisateur *postgres*.

La figure 1.3 page suivante montre un exemple de serveur en fonctionne-

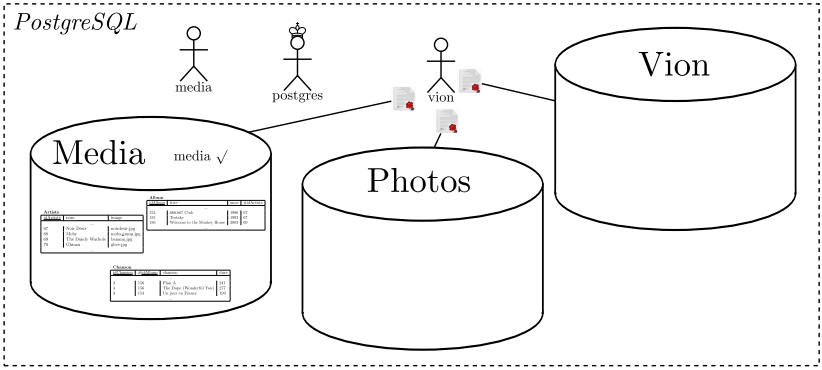



FIG. 1.3 : État d'un serveur *PostgreSQL*

ment. Trois utilisateurs, *media*, *postgres* et *vion*, sont enregistrés. *postgres* est administrateur. Trois bases existent : *Media*, *Photos*, et *Vion*. *Media* contient les tables *Artiste*, *Album* et *Chanson*. *vion* est le *propriétaire* des trois bases, et a donc un accès illimité à celles-ci. *postgres*, en tant qu'administrateur, y a également accès. De plus, l'utilisateur *media* a été autorisé à accéder à la BD *Media*. Seuls les utilisateurs *vion* et *postgres* ont accès aux BD *Photos* et *Vion*.

Il est possible de définir des droits très précisément : droits de créer/modifier des tables, insérer, sélectionner, mettre à jour ou supprimer des données différents pour chaque utilisateur et pour chaque table (par exemple, l'utilisateur *media* pourrait n'avoir le droit que de sélectionner des données dans la table *Artistes*, et le droit de sélectionner, insérer et modifier des informations dans la table *Chansons*.

Toutes les commandes d'administration, comme les commandes d'utilisation de la base, se font en langage SQL, *via* le client interactif psql.

Sur le serveur de l'IUT `iutinfo-sgbd`, un utilisateur ainsi qu'une base à son nom a été créé pour chaque étudiant, l'un d'entre eux vous sera affecté lors du premier TP.

### 1.6.4 Utilisation de psql

psql est un client en ligne de commande : il vous fournit une invite dans le terminal, à laquelle vous pouvez envoyer des requêtes SQL qui sont immédiatement exécutées. psql est installé automatiquement avec *PostgreSQL*. On l'exécute avec la commande :

psql [ *options* ] [ *database* ]

*database* est le nom de la base de données à laquelle vous vous connectez. Si elle est omise, psql se connecte à la base de données portant le nom de l'utilisateur courant. On se connecte généralement à *PostgreSQL* par le réseau

TCP/IP. Il faut indiquer un nom d'hôte (avec l'option `-h`), un utilisateur (avec l'option `-U`) et un mot de passe. Si vous travaillez sur les machines de TP de l'IUT, vous pouvez vous connecter au serveur commun avec la commande :

```
psql -h iutinfo-sgbd -U votre_login
```

Une fois psql lancé, vous obtenez l'invite suivante :

```
votre_login=>
```

Le symbole = signifie qu'aucune commande n'est en cours. Quand vous taperez des commandes, observez bien l'évolution de ce symbole.

Le symbole > signifie que vous êtes connecté avec un compte utilisateur. Si vous êtes connecté en tant qu'administrateur, il sera remplacé par le symbole #. Le compte administrateur ne doit être utilisé que pour créer des utilisateurs et éventuellement des bases de données, jamais en utilisation quotidienne.

Vous pouvez maintenant taper des commandes SQL. psql définit également des *macros*, qui sont converties en commandes SQL particulières, et servent généralement à obtenir des meta-informations, par exemple :

\? permet d'obtenir de l'aide sur psql.

\h (et \h *commande*) permet d'obtenir un aide-mémoire SQL.

\l permet d'obtenir une liste des bases de données.

\d permet d'obtenir la liste des tables et objets de la base de données actuelle.

\d *table* permet d'obtenir le schéma de la table *table*.

\i *fichier.sql* permet de charger le fichier *fichier.sql* et d'exécuter toutes les commandes SQL qui s'y trouvent.

\q (ou CTRL+D) permet de se déconnecter et de quitter psql.

Toutes les commandes SQL (pas les macros) doivent être terminées par un point-virgule « ; ». Vous pouvez insérer des retours à la ligne où vous le souhaitez, mais la commande ne sera finalement exécutée qu'au moment où vous inscrivez un point-virgule.

### 1.6.5 Créer une table

Déconnectez-vous si vous êtes encore connecté à la base *postgres* pour vous reconnecter à la nouvelle base.

Les commandes de création, insertion et requêtes sont détaillées dans le chapitre 4 page 41. Vous pouvez créer une première table avec la commande :

```
CREATE TABLE prems (
  id int PRIMARY KEY,
  nom text
);
```

Remarquez qu'après la première ligne, vous obtenez l'invite tuto(>. La parenthèse signifie qu'une parenthèse a été ouverte. Il faudra la refermer avant de terminer la commande par un point-virgule ;.

Insérez des données dans la table :

```
INSERT INTO prems VALUES (1, 'a'), (2, 'b'), (3, 'c');
```

Et affichez les avec la commande :

```
SELECT * FROM prems ;
```

### 1.6.6 Droits d'accès

Vous pouvez autoriser un autre utilisateur à se connecter à votre base par la commande :

```
GRANT CONNECT ON DATABASE nom_de_la_base TO login ;
```

Ensuite, vous devez définir les droits table par table et type par type :

```
GRANT SELECT ON prems TO login ;
```

Cette commande n'autorise *login* qu'à lire les données de la table *prems*. Vous pouvez donner tous les droits (lecture, insertion, modification, suppression) sur une table :

```
GRANT ALL ON prems TO login ;
```

De même, vous pouvez supprimer les droits par la commande **REVOKE**.

### 1.6.7 Utilisation de pgAdmin

pgAdmin est un logiciel graphique d'administration de bases de données *PostgreSQL*. Très complet, il est assez compliqué à prendre en main. Vous ne pourrez utiliser pgAdmin qu'une fois que votre utilisateur de travail aura été créé. Si vous souhaitez utiliser pgAdmin sur votre propre machine, suivez tout d'abord les instructions de la section 1.6.1 page 8. Installez en plus le *package* pgadmin3. Quand vous lancez le logiciel, vous devez voir apparaitre la fenêtre représentée sur la figure figure 1.4 page ci-contre.

pgAdmin est capable de gérer la connexion à plusieurs serveurs, sur une machine locale ou distante. Ici, nous allons nous connecter à la base de données locale. Cela se fait dans le menu Fichier → Ajouter un serveur... ou l'icône de la barre d'outils. Remplissez la fenêtre de connexion comme sur la figure 1.5 page 16. Vous pouvez indiquer n'importe quel nom. Remplacez le nom d'utilisateur par celui qui vous a été attribué ou que vous avez créé en section 1.6.1 page 10. Dans la case Hôte, laissez le champ vide pour vous

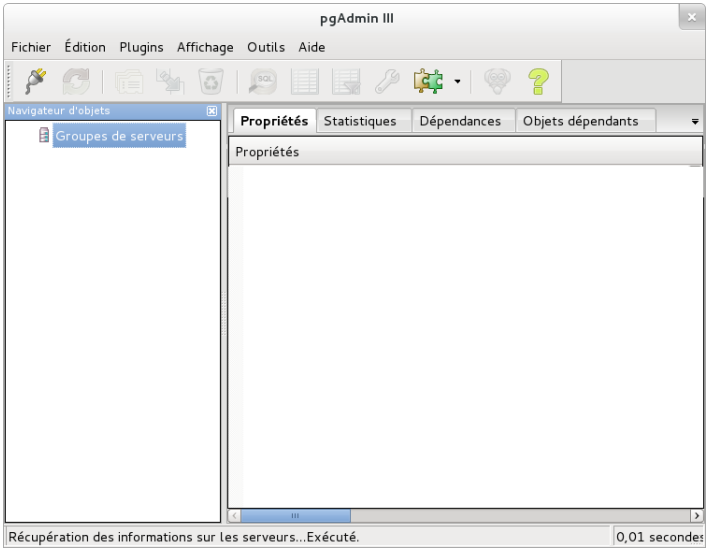



FIG. 1.4 : La fenêtre de pgAdmin au démarrage.

connecter *via* le *socket Unix*, ou indiquez *iutinfo-sgbd* ou *localhost* pour vous connecter *via* TCP/IP. Dans ce cas, vous devrez également indiquer un mot de passe, renseigné à la création de l'utilisateur.

Vous pouvez maintenant vous connecter à la base de données en double-cliquant sur le serveur nouvellement créé dans le navigateur d'objets. Les Bases de données, Tablespaces, Rôles groupe et Rôles de connexion apparaissent. Sur une nouvelle installation de *PostgreSQL*, seule la base de données *postgres* et les deux rôles *postgres* et celui que vous avez créé en section 1.6.1 page 10 doivent apparaitre, comme sur la figure figure 1.6 page 17. Sur *iutinfo-sgbd*, vous verrez apparaitre toutes les bases de données et rôles attribués aux étudiants. Nous allons ignorer les sections Tablespaces et Rôles groupe pour l'instant.

Si vous travaillez sur votre propre serveur, créez une base de données de travail. Faites un clic droit sur Base de données, et cliquez sur Ajouter une base de données…. Remplissez la nouvelle fenêtre comme indiqué sur la figure 1.7 page 17. Remplacez le propriétaire par votre utilisateur de travail. La base de données apparait dans le navigateur d'objets. Cliquez dessus pour l'activer. Si vous travaillez sur *iutinfo-sgbd*, cliquez simplement sur la base de données qui vous est attribuée : elle a le même nom que votre login.

Vous pouvez maintenant ouvrir le menu Outils → Éditeur de requêtes (ou l'icône ). La fenêtre de la figure 1.8 page 18 apparait. Vous pouvez y travailler comme dans psql, avec de plus une coloration syntaxique, la possibilité de naviguer dans le code, charger et enregistrer du code SQL, etc. Exécutez la requête *via* le menu Requêtes → Exécuter, l'icône ▶ ou la touche *F5*.

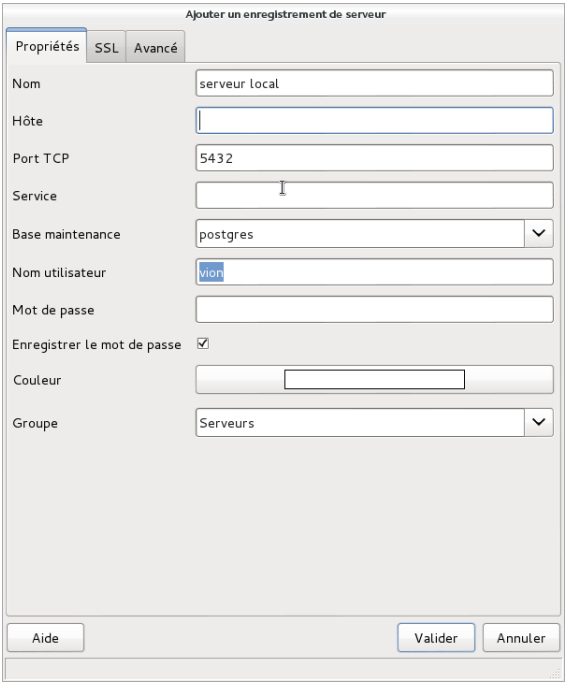



FIG. 1.5 : La fenêtre de connexion à un serveur.

Finalement, pgAdmin permet de naviguer dans les tables de la base de données, et permet la création de tables et de requêtes par formulaire, bien que cette dernière fonctionnalité ne soit pas très pratique. Pour afficher le contenu d'une table, dépliez le navigateur d'objets : Bases de données → test (ou autre base de données) → Schémas → public → Tables comme sur la figure 1.9 page 18. On affiche les données par un clic-droit sur la table, puis Afficher les données → Afficher toutes les lignes. Vous pouvez utiliser cette fenêtre pour manipuler les données (équivalent des commandes SQL **INSERT**, **UPDATE** et **DELETE**), comme sur la figure 1.10 page 19. Après avoir modifié des données ou des tables dans l'éditeur de requêtes, pensez à rafraichir pgAdmin par le menu Affichage → Rafraichir, l'icône ou la touche *F5*.

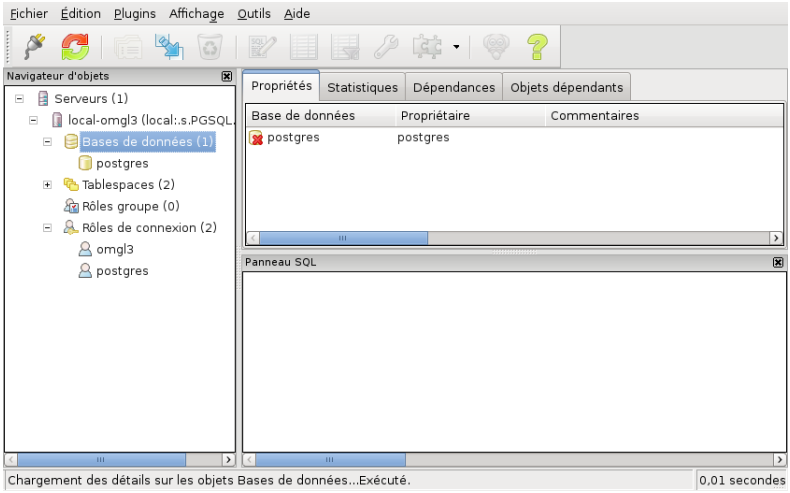



FIG. 1.6 : La fenêtre de *pgAdmin* après la connexion.

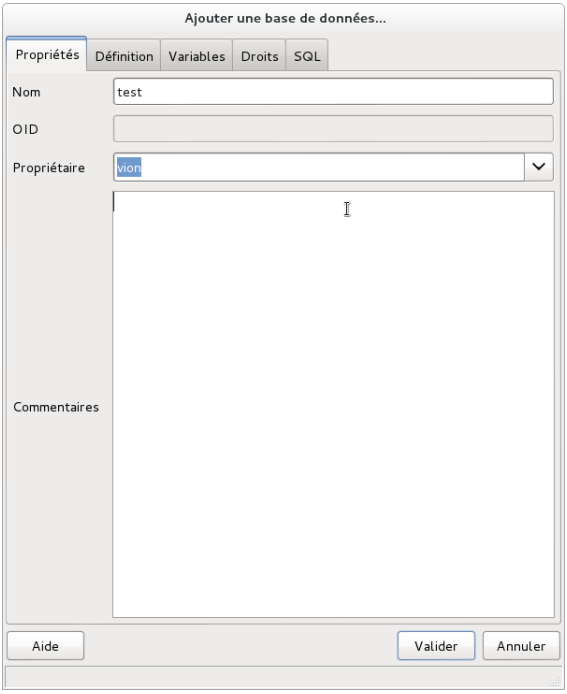



FIG. 1.7 : La fenêtre de création de base de données.

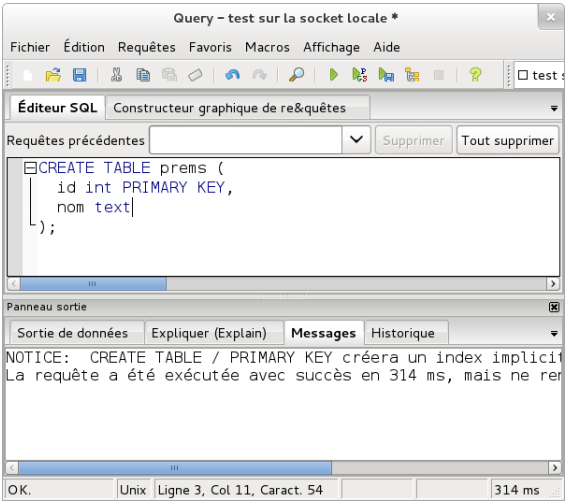



FIG. 1.8 : La fenêtre d'édition de requête SQL.

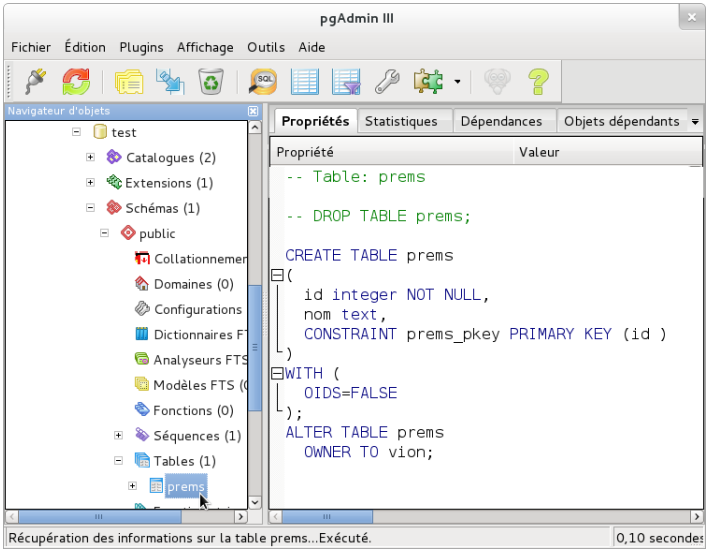



FIG. 1.9 : Administrer une table dans *pgAdmin*.

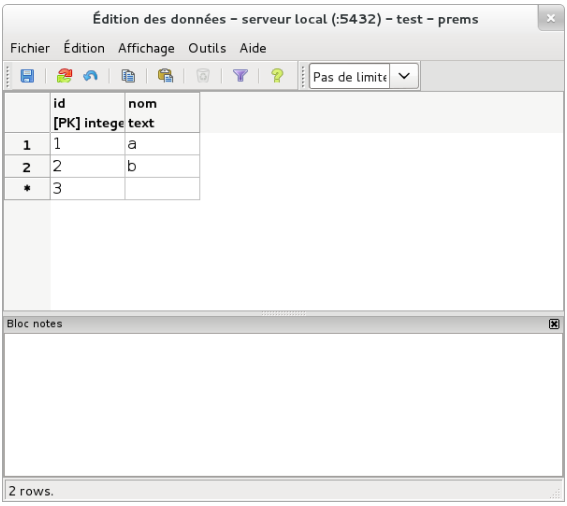



FIG. 1.10 : Manipuler des données dans *pgAdmin*.

Chapitre 2

# Modèle de données relationnel

## Sommaire


**2.1 Le modèle relationnel . . . . . 21**
**2.2 La clé primaire . . . . . 22**
**2.3 Les clés étrangères . . . . . 23**
**2.4 Définition des données en SQL . . . . . 23**
2.4.1 Colonnes et types . . . . . 24
2.4.2 Contraintes de clé primaire . . . . . 24
2.4.3 Contraintes de clé étrangère . . . . . 25
2.4.4 Autres contraintes . . . . . 27
2.4.5 Valeurs par défaut . . . . . 28
**2.5 Suppression d'une table . . . . . 29**
**2.6 Faire un script de création de base de données 30**


## 2.1 Le modèle relationnel

Une base de données relationnelle est constituée de *tables* indépendantes, comme dans l'exemple 1.1 page 3. Dans le modèle mathématique, on représente les données comme des ensembles de $n$-uplets ou des ensembles multidimensionnels discrets appelés *relations*. Quand on réalise un modèle relationnel, on conçoit la structure des tables (la liste des colonnes et leurs types), ainsi que les contraintes, notamment les *clés primaires* et *étrangères*. Le modèle obtenu est tel que celui représenté sur la figure 2.1.

La structure d'une table (son nom et ses colonnes) représente généralement un *type d'entité* ou une *classe d'objets* : la classe des artistes, la classe des albums, la classe des chansons…

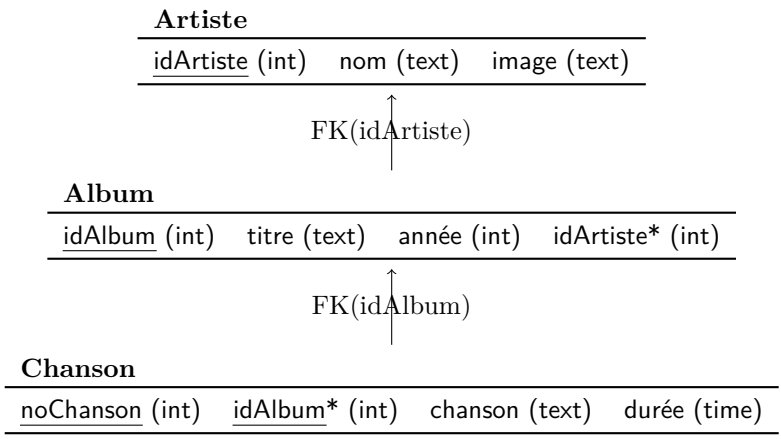



FIG. 2.1 : Modèle relationnel de la base de données de la page 3.

Une ligne de la base de données représente généralement une *entité* ou un *objet* : un artiste, un album, une chanson…donné. L'ordre des lignes dans la table n'est pas défini et peut changer à tout moment !

## 2.2 La clé primaire

Chaque table dispose *obligatoirement* d'*une et une seule clé primaire*. La clé primaire doit permettre d'identifier une ligne de la base de données de manière *unique*. Sur le modèle, la clé primaire est *soulignée*. Dans la table **Artiste**, la clé primaire est le champ idArtiste : à partir d'une valeur d'idArtiste, on a accès à toutes les informations de l'artiste. On aurait pu choisir le champ nom comme clé primaire, mais si plusieurs artistes ont le même nom, on ne peut plus assurer l'identification d'une ligne unique. Les SGBD imposent que chaque ligne ait une clé primaire distincte : l'insertion d'un nouvel artiste d'idArtiste 70 est refusée.

Parfois, la clé primaire est *composite*, c'est-à-dire qu'elle englobe plusieurs champs, comme dans la table Chanson. Cela signifie qu'il faut *tous les champs* de la clé primaire pour identifier une ligne de manière unique. Il ne suffit pas de savoir que l'on veut « la piste 3 », ou « la chanson de l'album 154 », pour savoir de quelle chanson on parle. Il faut la *combinaison* des deux informations : « piste 3 de l'album 154 ». La contrainte d'unicité est également imposée sur la combinaison des champs : on peut avoir plusieurs pistes 3, on peut avoir plusieurs chansons sur l'album 154, mais une seule piste 3 sur l'album 154.

Attention, même en cas de clé primaire composite, on considère toujours qu'il n'y a qu'*une seule clé primaire*. Deux clés primaires n'auraient pas la

même signification : cela signifierait qu'un seul des deux champs permettrait l'identification d'une ligne ! Or, il faut bien la *combinaison des deux* pour y parvenir.

## 2.3 Les clés étrangères

On utilise les clés étrangères pour *référencer* les lignes de la table à partir d'autres tables de la base de données. Par exemple, dans la table **Album**, on indique l'idArtiste pour indiquer l'auteur de l'album, au lieu de recopier toutes les informations sur l'artiste. Le champ idArtiste de la table **Album** est appelé *clé étrangère*.

On représente la clé étrangère par une flèche partant de la table *référençant* vers la table *référencée*. Une clé étrangère « pointe » toujours vers *toute la clé primaire* d'une table. Ainsi, il faudra une *clé étrangère composite* pour référencer une clé primaire composite. Pour éviter toute ambigüité, on indique quel(s) champ(s) ser(ven)t de référence sur la flèche de la clé primaire. De plus, le symbole « * » est utilisé pour repérer plus facilement les clés étrangères dans les tables. Par exemple, dans la table **Chanson**, le champ idAlbum fait référence à la clé primaire idAlbum de la table **Album**.

Ce mécanisme, central en base de données relationnelles, a de nombreux avantages : il évite les redondances d'informations (le nom de l'artiste n°70 n'apparait qu'une seule fois dans toute la base de données) et les éventuelles incohérences qui pourraient en découler (nom mal orthographié ou avec une casse différente), simplifie les modifications d'informations (changer l'orthographe du nom d'un artiste ne nécessite de modifier qu'une seule ligne de données) et dissocie les informations (on peut supprimer tous les albums d'un artiste sans perdre les informations sur l'artiste). Tout ceci sera développé dans le chapitre 5 page 69 de ce cours.

Les SGBD peuvent contrôler l'*intégrité* référentielle : quand on fait référence à l'artiste n°70, le SGBD contrôle que l'artiste n°70 existe.

## 2.4 Définition des données en SQL

La partie du langage permettant la définition des données est appelée *Langage de Définition des Données*. Ce langage permet de créer et définir la structure des tables, ainsi que certaines contraintes d'intégrité. Toutes les commandes SQL de définition des données commencent par **CREATE TABLE**, **ALTER TABLE** ou **DROP TABLE**. La documentation complète de ces commandes est donnée dans la section 4.4, à partir de la page 44. Il est également possible de créer les tables à partir d'une interface graphique telle que *pgAdmin*, bien que ce soit beaucoup plus long et laborieux.

### 2.4.1 Colonnes et types

La principale commande permettant la gestion des tables est la commande **CREATE TABLE**, qui, comme son nom l'indique, permet de créer une table. La commande est suffisamment complète pour créer la table, toutes ses colonnes avec leurs types, et définir les contraintes d'intégrité.

Commençons par créer une première table. La syntaxe de base de la commande est la suivante :

```
CREATE TABLE nom_table (
  colonne1 type1,
  colonne2 type2,
  ...);
```

Les types disponibles les plus utiles sont décrits dans la section 4.3 page 43. Par exemple, pour créer la table **Artiste** du modèle de la figure 2.1 page 22 :

```
CREATE TABLE Artiste (
  idArtiste int,
  nom text,
  image text);
```

La commande **ALTER TABLE** permet de renommer une table, renommer, ajouter ou supprimer une colonne. Par exemple, pour ajouter une colonne nationalite à la table **Artiste**, on peut utiliser la commande :

```
ALTER TABLE Artiste ADD nationalite text;
```

Pour supprimer la colonne image :

```
ALTER TABLE Artiste DROP image;
```

### 2.4.2 Contraintes de clé primaire

Il y a deux façons de définir une contrainte :

**Contrainte de colonne :** uniquement si la contrainte ne concerne qu'une seule colonne. Dans ce cas, on indique simplement la contrainte (ici **PRIMARY KEY**) à la suite de la définition de la colonne.

**Contrainte de table :** si la contrainte concerne une ou plusieurs colonnes, on définit la contrainte sur une ligne à part, comme s'il s'agissait d'une colonne. La liste des colonnes concernées est indiquée entre parenthèses.

Par exemple, pour créer la table **Album** avec sa clé primaire, on peut utiliser l'une des deux commandes suivantes :

**Contrainte de colonne :**

```
CREATE TABLE Album (
  idAlbum int PRIMARY KEY,
  titre text,
  annee int,
  idArtiste int);
```

**Contrainte de table :**

```
CREATE TABLE Album (
  idAlbum int,
  PRIMARY KEY (idAlbum),
  titre text,
  annee int,
  idArtiste int);
```

Dans le cas de la table Chanson, comme la clé primaire concerne deux champs, seule la syntaxe de la contrainte de table peut être utilisée :

```
CREATE TABLE Chanson (
  piste int,
  idAlbum int,
  PRIMARY KEY (piste, idAlbum),
  chanson text,
  duree time);
```

Pour ajouter une contrainte à une table (par exemple la clé primaire de la table **Artiste**), on utilise la commande **ALTER TABLE** avec la syntaxe de la contrainte de table :

```
ALTER TABLE Artiste ADD PRIMARY KEY (idArtiste);
```

La contrainte de clé primaire garantit l'unicité des clés, et définit automatiquement une *indexation* de la clé pour des recherches plus rapides sur la clé.

### 2.4.3 Contraintes de clé étrangère

Comme les clés primaires, les contraintes de clé étrangère peuvent être définies sous forme de contraintes de colonne ou de contrainte de table :

**Contrainte de colonne :** si la colonne référençante est simple, on ajoute la contrainte **REFERENCES** table_referencee (colonne_referencee) après la colonne concernée.

**Contrainte de table :** dans ce cas, on ajoute une ligne dans la définition de la table :

```
FOREIGN KEY (colonne11, colonne12...)
  REFERENCES table_referencee (colonne21, colonne22...)
```

Par exemple, pour la table Album :

### Contrainte de colonne :

```
CREATE TABLE Album (
  idAlbum int PRIMARY KEY,
  titre text,
  annee int,
  idArtiste int REFERENCES Artiste (idArtiste));
```

### Contrainte de table :

```
CREATE TABLE Album (
  idAlbum int PRIMARY KEY,
  titre text,
  annee int,
  idArtiste int,
  FOREIGN KEY idArtiste
    REFERENCES Artiste (idArtiste));
```

Si la colonne référençante et la colonne référencée ont le même nom, on peut omettre le nom de la colonne référencée :

```
CREATE TABLE Album (
  idAlbum int PRIMARY KEY,
  idArtiste int REFERENCES Artiste);
```

Rien n'empêche de définir plusieurs contraintes sur la même colonne :

```
CREATE TABLE Chanson (
  piste int,
  idAlbum int REFERENCES Album,
  PRIMARY KEY (piste, idAlbum));
```

Lorsque l'on définit un script SQL créant plusieurs tables à la suite, il faut bien sûr créer les tables référencées avant les tables référençantes. S'il y a un cycle, on peut reporter la création des clés étrangères et faire un **ALTER TABLE** une fois toutes les tables du cycle créées.

Les contraintes de clé étrangère interdisent de faire référence à une clé inexistante. Par exemple, on ne peut pas créer d'album d'idArtiste 80 s'il n'y a pas d'artiste n°80. Elles empêchent aussi la suppression d'éléments référencés. Par exemple, il est interdit de supprimer l'artiste n°70 s'il y a un album de cet artiste dans la base de données. Cependant, dans ce dernier cas, il est possible de paramétrer la contrainte pour supprimer automatiquement toutes les lignes qui font référence à la ligne supprimée (e.g., supprimer tous les albums de l'artiste n°70 si celui-ci est supprimé). Il faut pour cela ajouter le paramètre **ON DELETE CASCADE** derrière la clé étrangères. Des variantes de ce paramétrage sont disponibles (cf section 4.4.1 page 46). Il faut bien évidemment utiliser ce genre de fonctionnalités avec précaution. Par

exemple, dans une base de données de facturation, propager la suppression d'un article du stock pourrait altérer toutes les factures référençant cet article ! En tout état de cause, on évite généralement de modifier ou supprimer des informations d'une base de données.

### 2.4.4 Autres contraintes

Il y a trois autres types de contraintes que l'on peut définir sur une table :

- **NOT NULL** : uniquement sous forme de contrainte de colonne, elle interdit les valeurs **null** sur la colonne concernée. **null** signifie « valeur inconnue ». Il est préférable d'éviter les valeurs **null** dans une base de données, les champs seront donc presque toujours **NOT NULL**.
- **UNIQUE** : impose l'unicité d'une colonne ou d'une combinaison de colonnes. Notez que la contrainte **PRIMARY KEY** impose implicitement les contraintes **UNIQUE** et **NOT NULL**.
- **CHECK** : permet de définir une contrainte sous forme de prédicat (par exemple, duree $> 0$). Voir la section 4.6 page 64 pour un détail des opérateurs de prédicat disponibles.

On fait apparaitre ces contraintes sur le modèle relationnel avant de les implémenter.

Voici un exemple complet comportant des contraintes **NOT NULL** et **CHECK** :

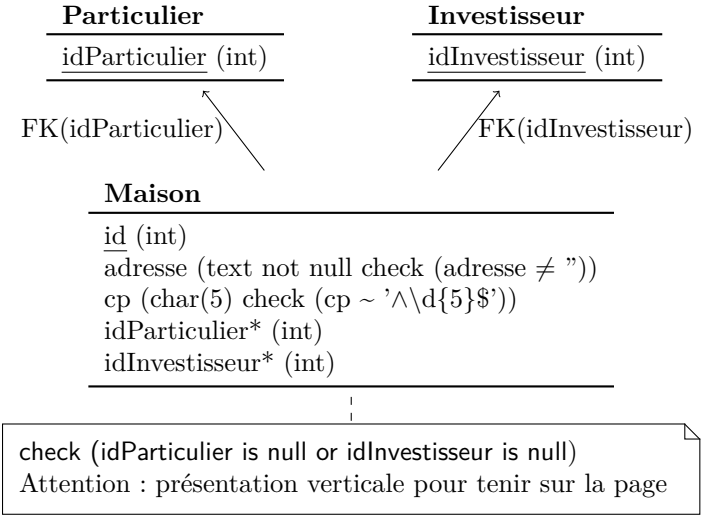



FIG. 2.2 : Modèle relationnel avec contraintes

```
CREATE TABLE Particulier(
  idParticulier int PRIMARY KEY);

CREATE TABLE Investisseur(
  idInvestisseur int PRIMARY KEY);

CREATE TABLE Maison(
  id int PRIMARY KEY,
  adresse text NOT NULL CHECK (adresse != ''),
  code_postal varchar(5) NOT NULL
    CHECK (code_postal ~ '^\\d{5}$'),
  idParticulier int REFERENCES Particulier,
  idInvestisseur int REFERENCES Investisseur,
  CHECK (idParticulier IS NULL
    OR idInvestisseur IS NULL));
```

### 2.4.5 Valeurs par défaut

Il est courant de vouloir définir des valeurs par défaut, c'est-à-dire des valeurs générées automatiquement, lorsque l'on définit une table de base de données. Par exemple, on peut vouloir insérer automatiquement la date et l'heure à laquelle la ligne a été créée, mettre une valeur **null** quand une valeur est inconnue, générer automatiquement une clé primaire numérique…

Cela se fait simplement en indiquant le mot-clé **DEFAULT** suivi d'une fonction calculant la valeur par défaut après la ligne de base de données. La fonction peut-être une valeur simple. La valeur par défaut peut s'ajouter aux éventuelles contraintes.

Par exemple, dans la table **Artiste** pour indiquer de mettre l'image pas_dimage.png en image par défaut, on peut indiquer :

```
CREATE TABLE Artiste (
  idArtiste int PRIMARY KEY,
  nom text NOT NULL,
  image text NOT NULL DEFAULT 'pas_dimage.png');
```

La génération automatique de clé primaire fonctionne différemment d'un SGBD à l'autre. Par exemple, avec *MySQL*, il suffit d'ajouter le mot-clé **AUTO_INCREMENT** après la déclaration de la clé primaire. Dans *PostgreSQL* ou *Oracle*, il faut créer une *séquence* et utiliser la fonction nextval pour obtenir un numéro unique à partir de la séquence. Là encore, la syntaxe diffère entre *PostgreSQL* et *Oracle*.

Pour *PostgreSQL* :

```
CREATE SEQUENCE Seq_Artiste;
CREATE TABLE Artiste (
  idArtiste int PRIMARY KEY
    DEFAULT nextval('Seq_Artiste'),
```

```
  nom text NOT NULL,
  image text NOT NULL DEFAULT 'pas_dimage.png');
```

Pour *Oracle* :

```
CREATE SEQUENCE Seq_Artiste;
CREATE TABLE Artiste (
  idArtiste int PRIMARY KEY
    DEFAULT Seq_Artiste.nextval,
  nom text NOT NULL,
  image text NOT NULL DEFAULT 'pas_dimage.png');
```

Pour *MySQL* :

```
CREATE TABLE Artiste (
  idArtiste int PRIMARY KEY AUTO_INCREMENT,
  nom text NOT NULL,
  image text NOT NULL DEFAULT 'pas_dimage.png');
```

Finalement, pour *PostgreSQL*, on peut remplacer le type **int** par **serial** pour créer automatiquement la séquence et la valeur par défaut. Le résultat obtenu est strictement équivalent à l'exemple précédent :

```
CREATE TABLE Artiste (
  idArtiste serial PRIMARY KEY,
  nom text NOT NULL,
  image text NOT NULL DEFAULT 'pas_dimage.png');
```

Notez que dans ce cas, les clés étrangères faisant référence à idArtiste seront de type **int** et non pas **serial** : les clés étrangères ne sont pas générées automatiquement !

## 2.5 Suppression d'une table

On supprime une table simplement par la commande :

```
DROP TABLE nom_table;
```

Par exemple :

```
DROP TABLE Chanson;
```

Toutes les données de la table seront supprimées sans aucune confirmation ; il faut donc manier cette commande avec précaution.

S'il existe une référence vers cette table (notamment une contrainte de clé étrangère), la suppression de la table sera refusée. On peut forcer malgré tout la suppression, en supprimant automatiquement les contraintes, en ajoutant le mot **CASCADE** derrière la commande.

## 2.6 Faire un script de création de base de données

Dans le cadre de vos développements, vous serez amenés à créer des bases de données de plusieurs tables. Comme il est très rare de créer toute une base sans faire de faute, on gagne un temps précieux en écrivant un *script de création*, c'est-à-dire un fichier texte, d'extension `.sql`, contenant toutes les commandes **CREATE TABLE** et éventuellement **ALTER TABLE**. Le script commence généralement par des commandes supprimant d'éventuelles anciennes tables (l'option **IF EXISTS** permet de ne pas interrompre le script si la table n'existe pas). Il est ensuite très facile de charger le script dans la base de données, et de le modifier en cas d'erreur.

Voici un script complet (le plus concis possible) de développement de la base de données d'exemple :

```
DROP TABLE IF EXISTS Chanson;
DROP TABLE IF EXISTS Album;
DROP TABLE IF EXISTS Artiste;

CREATE TABLE Artiste (
  idArtiste serial PRIMARY KEY,
  nom text NOT NULL,
  image text NOT NULL DEFAULT 'pas_dimage.png');

CREATE TABLE Album (
  idAlbum serial PRIMARY KEY,
  titre text NOT NULL,
  annee int,
  idArtiste int NOT NULL REFERENCES Artiste);

CREATE TABLE Chanson (
  piste int,
  idAlbum int REFERENCES Album,
  PRIMARY KEY (piste, idAlbum),
  chanson text NOT NULL,
  duree time NOT NULL);
```

Chapitre 3

# Manipulation des données en SQL

## Sommaire


**3.1 Modifier des données . . . . . 31**
3.1.1 Insérer des données . . . . . 31
3.1.2 Supprimer des données . . . . . 32
3.1.3 Mettre à jour des données . . . . . 33
**3.2 Obtenir des données . . . . . 33**
3.2.1 Restriction (clause **WHERE** ou **HAVING**) . 35
3.2.2 Projection et liste **SELECT** . . . . . 36
3.2.3 Produit cartésien et jointure : . . . . . 36
3.2.4 Groupements ou agrégats . . . . . 38
3.2.5 Jointures externes . . . . . 40


## 3.1 Modifier des données

La modification des données en SQL concerne l'ajout, la suppression et la mise à jour des lignes dans table.

### 3.1.1 Insérer des données

L'insertion de données dans une table se fait avec la commande **INSERT**. La documentation de la commande se trouve dans la section 4.5.1 page 52 de ce cours.

On ne peut insérer des données que dans une seule table à la fois. Les chaines de caractères (champs **text**) doivent être indiqués entre *simple quotes* (apostrophes droites). Les champs spéciaux (dates, heures, timestamps) sont généralement indiqués sous forme de chaine de caractères automatiquement converties par le SGBD. On insère une ligne complète, en indiquant les valeurs de chaque colonne dans l'ordre où elles ont été créées. La syntaxe de base est :

```
INSERT INTO nom_table VALUES (val1, val2, ...);
```

Par exemple, pour insérer une chanson dans la table **Chanson** :

```
INSERT INTO Chanson VALUES (3, 156, 'Plan A', 241);
```

On peut insérer plusieurs lignes à la fois :

```
INSERT INTO Chanson VALUES
  (4, 156, 'The Dope (Wonderful You)', 277),
  (3, 154, 'Un jour en France', 193);
```

Dans ce cas, si une ligne ne peut pas être insérée (violation de contrainte ou type incorrect), aucune donnée ne sera enregistrée.

Pour générer les valeurs par défaut, on remplace les valeurs correspondantes par le mot-clé **DEFAULT** :

```
INSERT INTO Artiste
VALUES (DEFAULT, 'Hooverphonic', DEFAULT);
```

Il est également possible de redéfinir l'ordre ou de n'indiquer qu'une partie des colonnes en listant les colonnes souhaitées après le nom de la table (dans ce cas, les colonnes omises se verront attribuer leur valeur par défaut). Ainsi, la dernière requête est équivalente à :

```
INSERT INTO Artiste (nom) VALUES ('Hooverphonic');
```

Et la précédente à :

```
INSERT INTO Chanson
  (noPiste, idAlbum, idArtiste, titre)
VALUES
  (4, 156, 277, 'The Dope (Wonderful You)'),
  (3, 154, 193, 'Un jour en France');
```

Finalement, il est possible de combiner une commande **INSERT INTO** avec une commande **SELECT** (détaillée dans la section 3.2 page suivante) pour copier des informations d'une table à une autre ou migrer des données. Dans ce cas, chaque ligne renvoyée par la commande **SELECT** sera insérée dans la table.

### 3.1.2 Supprimer des données

La commande **DELETE** (documentée en section 4.5.2 page 54) permet de supprimer des lignes d'une table. La syntaxe est :

```
DELETE FROM nom_table
WHERE prédicat ;
```

Le prédicat est libre et peut contenir n'importe quelle expression telle que définie dans la section 4.6 page 64. On supprime le plus souvent une ligne à la fois en filtrant sur une clé primaire. Par exemple :

```
DELETE FROM Chanson
WHERE (piste, idAlbum) = (3, 154);
```

Attention, si on oublie la clause **WHERE**, toutes les données de la table sont supprimées. Attention aussi à ne pas se tromper dans le prédicat pour ne pas supprimer des lignes imprévues. Plus généralement, on supprime rarement des lignes d'une base de données.

### 3.1.3 Mettre à jour des données

La commande **UPDATE** (documentée en section 4.5.3 page 55) permet de mettre à jour les lignes d'une base de données. Comme la commande **DELETE**, on utilise une clause **WHERE** pour indiquer les lignes à modifier. De plus, une clause **SET** permet d'indiquer la nature d'une mise à jour : *champ* = nouvelle_valeur. La nouvelle valeur peut être calculée avec n'importe quel opérateur ou fonction détaillé dans la section 4.6 page 64.

La syntaxe :

```
UPDATE table
SET column1 = expression1,
    column2 = expression2,
    ...
WHERE condition
```

Pour modifier le titre d'une chanson :

```
UPDATE Chanson
SET titre = 'The Dope (...)'
WHERE (piste, idAlbum) = (4, 156);
```

Pour diviser par 60 les durées de toutes les chansons :

```
UPDATE Chanson
SET duree = duree / 60
```

Comme pour **DELETE**, oublier la clause **WHERE** revient à modifier toutes les lignes de la table. Il faut donc manier la commande avec précaution, voire l'éviter complètement si possible.

## 3.2 Obtenir des données

La commande **SELECT** (documentée en section 4.5.4 page 56) est la commande la plus compliquée du langage SQL. Ses possibilités sont multiples :

recherche d'informations, croisement de données, calcul de statistiques… Sa maitrise demande de la pratique et une compréhension profonde de la nature des données dans une base de données relationnelles. L'*algèbre relationnelle*, théorisée par Edgar F. Codd en 1970 [4] fournit les outils mathématiques nécessaires à la manipulation des tables de bases de données et le croisement d'informations entre plusieurs tables. Basée sur la théorie des ensembles, elle définit et exploite les opérations qui seront détaillées dans la suite de cette section.

La commande **SELECT** se compose de huit clauses optionnelles, à indiquer *en respectant l'ordre indiqué* :

1. **SELECT** : indique les colonnes et opérations sur celles-ci à renvoyer. On peut effectuer des calculs avec les expressions et opérateurs décrits en section 4.6. Il est possible de supprimer d'éventuels doublons avec le mot-clé **DISTINCT** (projection). Le symbole * permet de renvoyer toutes les colonnes de toutes les tables concernées ;

2. **FROM** : indique les tables d'où obtenir les informations, et éventuellement la manière de croiser celles-ci (jointures) ;

3. **WHERE** : filtre les lignes obtenues suivant un prédicat (restriction), le prédicat peut être complexe et même comporter des sous-requêtes (cf section 4.6) ;

4. **GROUP BY** : regroupe certaines lignes présentant des points communs (généralement un champ identique) en une seule, permet de faire des statistiques sur les lignes regroupées (comptage, sommes, moyennes…) ;

5. **HAVING** : effectue une seconde restriction *après* le regroupement (permet notamment de faire une restriction sur le résultat d'une statistique) ;

6. **ORDER BY** : indique comment trier les lignes (suivant quelle(s) colonne(s)) ;

7. **LIMIT** et

8. **OFFSET** : permettent de n'afficher qu'une partie des lignes renvoyées par la requête.

La requête **SELECT** la plus simple consiste simplement à afficher le contenu d'une seule table :

```
SELECT * FROM Artiste ;
```

Même la clause **FROM** est optionnelle : **SELECT** 2 + 2 est une requête valide (elle renvoie le résultat de l'opération). **SELECT** now() renvoie la date et l'heure du serveur.

### 3.2.1 Restriction (clause WHERE ou HAVING)

La restriction permet de filtrer les données d'une table de la base de données. Il s'agit de l'opération la plus simple et naturelle : elle s'apparente à une recherche. On fournit le prédicat, et le SGBD renvoie les lignes validant celui-ci. Le prédicat peut être compliqué, comporter de nombreux expressions et opérateurs (cf section 4.6).

Voici quelques exemples simples :

**Album n°156 :**

```
SELECT * FROM Album
WHERE idAlbum = 156
```

**Albums parus en 2000 :**

```
SELECT * FROM Albums
WHERE annee = 2000
```

**Albums parus entre 2001 et 2009 (inclus) :**

```
SELECT * FROM Albums
WHERE 2001 <= annee AND annee <= 2009
```

ou :

```
SELECT * FROM Albums
WHERE annee BETWEEN 2001 AND 2009
```

**Albums dont l'année de parution est inconnue :**

```
SELECT * FROM Albums
WHERE annee IS NULL
```

**Albums dont le titre est « Welcome to the Monkey House » (sans tenir compte des majuscules) :**

```
SELECT * FROM Album
WHERE titre ILIKE 'welcome to the monkey house'
```

**Albums de Noir Désir :**

En connaissant l'identifiant de Noir Désir :

```
SELECT * FROM Album
WHERE idArtiste = 67
```

Sans connaitre l'identifiant (en imbriquant des requêtes) :

```
SELECT * FROM Album
WHERE idArtiste = (
  SELECT idArtiste FROM Artiste
  WHERE nom = 'Noir Désir')
```

### 3.2.2 Projection et liste SELECT

Bien qu'il soit usuel d'indiquer simplement la liste des colonnes que l'on souhaite voir retourner par la commande **SELECT**, il est également possible d'effectuer des calculs, statistiques et supprimer les doublons en indiquant des formules derrière la commande **SELECT**.

Par exemple, pour n'indiquer que le nom et l'année de sortie des albums, on peut utiliser la requête :

```
SELECT titre, annee FROM Album
```

Quelques exemples incluant des calculs d'expressions :

**Durée en minutes des chansons :**

```
SELECT piste, idAlbum, chanson, duree / 60 FROM Chanson
```

**Titre des albums en majuscules :**

```
SELECT upper(titre) FROM Album
```

La suppression des doublons (qui correspond à la projection d'un ensemble multidimensionnel – chaque colonne de la table correspond à une dimension de l'ensemble – sur une partie de ses dimensions) est une commande importante permettant d'obtenir de manière simple la réponse à des questions apparemment complexes.

**Années où au moins un album est paru :**

```
SELECT DISTINCT annee FROM Album
```

**Identifiant des artistes dont au moins un album est référencé :**

```
SELECT DISTINCT idArtiste FROM Album
```

### 3.2.3 Produit cartésien et jointure :

Le produit cartésien permet de générer une seule table à partir de deux tables. Chaque ligne de la première table est associée à chaque ligne de la deuxième table. Si la première table contient $n$ lignes et la deuxième $m$ lignes,

| idArtiste | nom | idAlbum | titre | idArtiste |
|---|---|---|---|---|
| | | ... | | |
| 67 | Noir Désir | 154 | 666.667 Club | 67 |
| 67 | Noir Désir | 155 | Tostaky | 67 |
| 67 | Noir Désir | 156 | Welcome to... | 69 |
| 68 | Moby | 154 | 666.667 Club | 67 |
| 68 | Moby | 155 | Tostaky | 67 |
| 68 | Moby | 156 | Welcome to... | 69 |
| 69 | The Dandy... | 154 | 666.667 Club | 67 |
| 69 | The Dandy... | 155 | Tostaky | 67 |
| 69 | The Dandy... | 156 | Welcome to... | 69 |
| 70 | Ghinzu | 154 | 666.667 Club | 67 |
| 70 | Ghinzu | 155 | Tostaky | 67 |
| 70 | Ghinzu | 156 | Welcome to... | 69 |
| | | ... | | |

TAB. 3.1 : Produit cartésien des tables **Artiste** et **Album**. Les lignes grisées n'ont aucune signification !

la table obtenue fera $n \times m$ lignes (ainsi, effectuer le produit cartésien de deux tables de 1 000 lignes chacune renvoie une table d'un million de lignes). On trouve rarement le produit cartésien utilisé seul : il n'a généralement aucun sens en bases de données relationnelles.

On effectue le produit cartésien de deux tables en utilisant le mot-clé **CROSS JOIN** ou une virgule dans la clause **FROM**.

```
SELECT Artiste.idArtiste, nom, idAlbum, titre,
  Album.idArtiste
FROM Artiste CROSS JOIN Album
```

Le tableau 3.1 indique le résultat de cette requête d'après les données de la page 3.

Le principal intérêt du produit cartésien est qu'il permet, en combinaison avec la restriction, de *croiser* les données de plusieurs tables. Le résultat du produit cartésien de la table 3.1 n'a pas d'intérêt en soi, mais si on se limite aux lignes où l'idArtiste de la table **Artiste** correspond à l'idArtiste de la table **Album**, on parvient à associer les informations de l'artiste et de ses albums.

Dans un premier temps, on peut ajouter explicitement la restriction dans la requête. Ainsi, seules les lignes noires de la table précédente sont conservées :

```
SELECT * FROM Artiste CROSS JOIN Album
WHERE Artiste.idArtiste = Album.idArtiste
```

Cependant, il est bien plus intéressant d'indiquer directement cette *condition de jointure* à l'intérieur de la clause **FROM**, à l'aide des mots-clés **JOIN** et **ON**. Cette syntaxe permet de dissocier bien mieux les conditions de jointure des véritables restrictions :

```
SELECT *
FROM Artiste JOIN Album
  ON Artiste.idArtiste = Album.idArtiste
```

On peut ainsi obtenir tous les albums de Noir Désir sans connaitre son identifiant, à l'aide d'une « simple » jointure :

```
SELECT Album.*
FROM Artiste JOIN Album
  ON Artiste.idArtiste = Album.idArtiste
WHERE nom = 'Noir Désir'
```

Bien que la condition de jointure soit libre (n'importe quel prédicat peut être utilisé), on fait généralement des jointures en associant une clé étrangère à la clé primaire correspondante. *Si la clé étrangère a le même nom que la clé primaire*, on peut utiliser une syntaxe abrégée à l'aide du mot-clé **USING** :

```
SELECT * FROM Artiste JOIN Album USING (idArtiste)
```

Enfin, *PostgreSQL* est capable de détecter automatiquement les *champs de même nom* et d'effectuer la jointure sur ces champs à l'aide du mot-clé **NATURAL** :

```
SELECT * FROM Artiste NATURAL JOIN Album
```

Attention cependant à ce que les champs servant à faire la jointure soient les seuls champs de même nom.

Il est finalement possible d'enchainer les jointures : il suffit de considérer le résultat d'une jointure comme une « table virtuelle » que l'on joint avec une troisième table :

```
SELECT *
FROM Chanson NATURAL JOIN Album NATURAL JOIN Artiste
```

### 3.2.4 Groupements ou agrégats

Beaucoup de requêtes visent à obtenir des statistiques sur un ensemble de données. Comment répondre aux questions « quelle est la durée totale d'un album », « combien d'albums a réalisé chaque artiste », etc. ?

On réalise assez facilement ce genre de requêtes en SQL grâce aux *fonctions d'agrégat* : comptage, somme, moyenne, etc. Ainsi, la requête précédente peut s'écrire en SQL :

```
SELECT sum(durée) FROM Chanson WHERE idAlbum=3
```

Les fonctions d'agrégat calculent une valeur unique à partir d'un ensemble (regroupement) de valeurs. La simple requête **SELECT** duree **FROM** Chanson renvoie la durée de chaque chanson, et la fonction **sum** calcule alors la somme de ces durées. Différentes fonctions d'agrégats utiles sont présentées dans la section 4.6.7 page 66. Les fonctions les plus courantes sont **count**(*) qui compte le nombre de lignes regroupées (**count**(champ) compte le nombre de lignes où champ n'est pas **null**) et **sum**(champ) qui compte la somme des valeurs du champ des lignes regroupées.

Il est finalement possible de *regrouper* les lignes ayant certains champs identiques grâce à la clause **GROUP BY**. Le résultat de la fonction d'agrégat est alors calculé *pour chaque regroupement*. L'exemple suivant calcule le nombre d'albums de chaque artiste :

```
SELECT idArtiste, nom, count(*)
FROM Artiste NATURAL JOIN Album
GROUP BY idArtiste, nom
```

Consultez la section 4.5.4 page 60 pour le détail du fonctionnement de la clause **GROUP BY**.

La clause SQL **HAVING** permet d'effectuer une restriction *après* que l'agrégat ait été réalisé. La clause **HAVING** se place donc après le **GROUP BY**. Il faut bien différencier les deux requêtes suivantes (d'après les données de la table 1.1) :

### Durée moyenne des chansons des albums, ne prenant en compte que les chansons de plus de 250 secondes :

```
SELECT idAlbum, titre, avg(durée)
FROM Album NATURAL JOIN Chanson
WHERE durée > 250
GROUP BY idAlbum, titre
```

| idAlbum | titre | avg |
|---|---|---|
| 156 | Welcome to the Monkey House | 277 |

### Albums dont la durée moyenne des chansons est de plus de 250 secondes :

```
SELECT idAlbum, titre, avg(durée)
FROM Album NATURAL JOIN Chanson
GROUP BY idAlbum, titre
HAVING avg(durée) > 250
```

| idAlbum | titre | avg |
|---|---|---|
| 156 | Welcome to the Monkey House | 259 |

Notez qu'il n'est possible de filtrer une requête sur le résultat d'une fonction d'agrégat que dans la clause **HAVING**.

### 3.2.5 Jointures externes

Dans l'exemple précédent, si on a un artiste dans la base de données qui n'a aucun album référencé, celui-ci n'apparaitra pas dans le résultat d'une jointure entre Album et Artiste, puisque la condition de jointure n'est jamais vérifiée. Ce comportement peut être gênant quand on veut par exemple connaitre le nombre d'album de chaque artiste : on souhaiterait que l'artiste apparaisse avec « 0 albums ».

Il est possible de répondre à cette question en utilisant une *jointure externe* : si un artiste n'a aucun album, il apparait quand même dans le résultat de la requête, les champs de la table Album étant alors mis à **NULL**. On obtient une jointure externe en utilisant le mot-clé **LEFT JOIN** (ou **RIGHT JOIN** suivant le sens de l'opération) :

```
SELECT nom, titre
FROM Artiste LEFT JOIN Album USING (idArtiste)
```

| nom | titre |
|---|---|
| Noir Désir | 666.667 Club |
| Noir Désir | Tostaky |
| Moby | *NULL* |
| The Dandy Warhols | Welcome to the Monkey House |
| Ghinzu | *NULL* |

Cette construction permet de trouver facilement les artistes n'ayant pas d'albums :

```
SELECT nom
FROM Artiste LEFT JOIN Album USING (idArtiste)
WHERE idAlbum IS NULL
```

Ou encore pour compter le nombre d'albums par artiste, en faisant apparaitre la valeur « 0 » en face des artistes n'ayant aucun album (notez que la fonction **count** ne compte pas les lignes pour lesquelles le champ indiqué est à **NULL**) :

```
SELECT nom, count(idAlbum)
FROM Artiste LEFT JOIN Album USING (idArtiste)
```

Chapitre 4

# SQL : l'essentiel

## Sommaire


**4.1 Compétences mises en œuvre . . . . . 42**
**4.2 Introduction . . . . . 42**
**4.3 Types de données . . . . . 43**
4.3.1 Types de données standard . . . . . 43
4.3.2 Types énumérés . . . . . 44
**4.4 Définition des données . . . . . 44**
4.4.1 **CREATE TABLE** – Définir une nouvelle table 44
4.4.2 **ALTER TABLE** – Modifier une définition de table . . . . . 48
4.4.3 **DROP TABLE** – Supprimer une table . . . 51
**4.5 Manipulation des données . . . . . 52**
4.5.1 **INSERT** – Insérer de nouvelles lignes dans une table . . . . . 52
4.5.2 **DELETE** – Supprimer des lignes d'une table 54
4.5.3 **UPDATE** – Modifier des lignes dans une table 55
4.5.4 **SELECT** – Récupèrer des lignes d'une table ou d'une vue . . . . . 56
**4.6 Expressions et opérateurs . . . . . 64**
4.6.1 Opérateurs logiques . . . . . 64
4.6.2 Opérateurs de comparaison . . . . . 64
4.6.3 Opérateurs mathématiques . . . . . 64
4.6.4 Opérateurs de chaines . . . . . 65
4.6.5 Correspondance de motifs . . . . . 65

4.6.6 Opérateurs sur date/heure . . . . . 66
4.6.7 Fonctions d'agrégat . . . . . 66
4.6.8 Expressions de sous-requêtes . . . . . 67

## 4.1 Compétences mises en œuvre

Ce chapitre traite des compétences suivantes :

- Maitriser le langage SQL,
- Langages prédicatifs,
- Définition d'un schéma relationnel en SQL, gestion des contraintes d'intégrité, notions d'index.

## 4.2 Introduction

Le *SQL* est un langage déclaratif, créé en 1975. Il a été normalisé pour la première fois en 1986 par l'*ANSI* (*American National Standards Institute*), puis en 1987 par l'*ISO* (*International Organisation for Standardization*). La norme a été révisée régulièrement (jusque récemment en 2008), mais c'est généralement la norme de 1992 (SQL-92) qui est prise en compte par les SGBDR. À titre informatif, la norme *SQL:2008* représente près de 3 800 pages de texte. Tous les SGBDR n'implémentent pas toute la norme, et peuvent au contraire supporter des fonctions qui ne sont pas normalisées. *PostgreSQL* supporte la plus grande partie de la norme SQL-92.

Le SQL se décompose en 5 parties :

**Définition des données.** Permet de modifier la structure de la BD, par exemple **CREATE**/**ALTER**/**DROP TABLE**/**VIEW**.

**Manipulation des données.** Permet de consulter et modifier le contenu de la BD (**INSERT**, **UPDATE**, **DELETE**, **SELECT**).

**Contrôle des données.** Permet de gérer les privilèges, (**CREATE**/**ALTER**/**DROP ROLE**, **GRANT**, **REVOKE**). Cf chapitre 1.6, sera revu plus en détail au module M2106.

**Contrôle des transactions.** Permet de gérer les transactions (regrouper un ensemble d'ordres en une seule opération atomique avec **BEGIN**, **COMMIT**, **ROLLBACK**, **SET TRANSACTION ISOLATION LEVEL**). (Sera étudié au module M2106.)

**SQL procédural.** Ensemble d'outils pour développer des *procédures stockées* et des *déclencheurs*, permettant de programmer directement au niveau du SGBDR : **CREATE FUNCTION**/**TRIGGER**. (Sera étudié au module M2106.)

Ce chapitre est une version abrégée de la documentation officielle de *PostgreSQL* [7]. Tous les paramètres définis dans ce chapitre pourront être utilisés lors des problèmes.

**Lire un synopsis :** Les sections suivantes décrivent les *synopsis* (simplifiés) des principales commandes SQL. Les MAJUSCULES D'IMPRIMERIE indiquent un mot-clé SQL. Un terme en *italiques* indique un paramètre défini par le programmeur.

Une section [ entre crochets ] indique un paramètre *optionnel*. Enfin, la barre verticale | dans une section { entre accolades } ou [ entre crochets ] indique une alternative : { a | b | c } signifie « soit a, soit b, soit c ». L'alternative entre crochets signifie que l'ensemble de l'alternative est optionnelle (entre accolades, au moins une des alternatives doit obligatoirement être sélectionnée).

# 4.3 Types de données

## 4.3.1 Types de données standard

bigint : entier relatif sur 8 octets ($-2^{63}$ à $2^{63}-1 \approx 9{,}2 \cdot 10^{18}$).

bigserial : cf type serial. Entier sur 8 octets à incrémentation automatique.

boolean : booléen (**TRUE** ou **FALSE**).

date : date du calendrier (année, mois, jour).

double precision : nombre à virgule flottante de double précision (sur 8 octets).

int : entier relatif sur 4 octets ($-2^{31}$ à $2^{31}-1$, soit $-2\,147\,483\,648$ à $2\,147\,483\,647$).

numeric (*p*, *s*) : nombre à virgule fixe de précision paramétrable : $p$ est la précision (nombre de chiffres significatifs), $s$ est l'échelle (nombre de chiffres après la virgule). 23,514 12 a une précision de 7 et une échelle de 5. Les entiers ont une échelle de 0. Le type numeric est utile pour stocker des nombres de manière exacte (des valeurs monétaires, par exemple), mais est beaucoup plus lent que les types int ou flottants.

real : nombre à virgule flottante de simple précision (4 octets).

serial : entier sur 4 octets à incrémentation automatique (spécifique à *PostgreSQL*). Notez qu'il ne s'agit pas vraiment d'un type, mais d'une macro : *PostgreSQL* crée une *Sequence* (un nombre entier stocké en mémoire), qui est incrémenté et affecté par défaut au champ lors d'une insertion de données.

**text :** chaine de caractères de longueur variable (sans maximum). Dans la plupart des SGBD, on réserve ce type aux longs paragraphes de texte sur lesquels on évite d'effectuer des restrictions, peu performantes. En *PostgreSQL*, il n'y a aucune différence d'implantation entre le type text et le type varchar, donc pourra donc utiliser le type text dans tous les cas.

**time :** heure du jour.

**timestamp :** date et heure.

**varchar (*n*) :** chaine de caractères de longueur variable (maximum *n*). Utile pour stocker des mots ou expressions.

### 4.3.2 Types énumérés

Les types énumérés (*enum*) sont des types de données qui comprennent un ensemble statique, prédéfini de valeurs dans un ordre spécifique. Ils sont équivalents aux types *enum* dans de nombreux langages de programmation. Les jours de la semaine ou un ensemble de valeurs de statut pour un type de données sont de bons exemples de type enum.

Les types enum sont créés en utilisant la commande **CREATE TYPE**. Par exemple :

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Une fois créé, le type enum peut être utilisé dans des définitions de table et de fonction, comme tous les autres types :

```
CREATE TYPE humeur AS ENUM ('triste', 'ok', 'heureux');
CREATE TABLE personne (
    nom TEXT,
    humeur_actuelle humeur);
INSERT INTO personne VALUES ('Moe', 'heureux');

SELECT * FROM personne
WHERE humeur_actuelle = 'heureux';
```

| name | humeur_actuelle |
|---|---|
| Moe | heureux |

## 4.4 Définition des données

### 4.4.1 CREATE TABLE – Définir une nouvelle table

**Synopsis**

```
CREATE TABLE table_name ( [
  { column_name data_type [ DEFAULT default_expr ]
          [ column_constraint [ ... ] ]
    | table_constraint } [ , ... ] ] )
```

où *column_constraint* peut être :

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  UNIQUE |
  PRIMARY KEY |
  CHECK ( expression ) |
  REFERENCES reftable [ ( refcolumn ) ]
    [ ON DELETE action ] [ ON UPDATE action ] }
```

et *table_constraint* :

```
[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [ , ... ] ) |
  PRIMARY KEY ( column_name [ , ... ] ) |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [ , ... ] )
    REFERENCES reftable [ ( refcolumn [ , ... ] ) ]
    [ ON DELETE action ] [ ON UPDATE action ] }
```

### Description

**CREATE TABLE** crée une nouvelle table initialement vide dans la base de données courante. Les clauses de contrainte optionnelles spécifient les contraintes (ou tests) que les nouvelles lignes ou les lignes mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Une contrainte est un objet SQL qui aide à définir l'ensemble des valeurs valides de différentes façons.

Il existe deux façons de définir des contraintes : celles de table et celles de colonnes. Une contrainte de colonne fait partie de la définition de la colonne. Une définition de contrainte de tables n'est pas liée à une colonne particulière et peut englober plusieurs colonnes. Chaque contrainte de colonne peut être écrite comme une contrainte de table ; une contrainte de colonne n'est qu'un outil de notation utilisé lorsque la contrainte n'affecte qu'une colonne.

### Paramètres

*table_name* **:** Le nom de la table à créer.

*column_name* **:** Le nom d'une colonne de la nouvelle table.

*data_type* **:** Le type de données de la colonne (cf section 4.3).

DEFAULT *default_expr* **:** La clause DEFAULT, apparaissant dans la définition d'une colonne, permet de lui affecter une valeur par défaut. La valeur

est une expression libre de variable (les sous-requêtes et références croisées aux autres colonnes de la table courante ne sont pas autorisées). Le type de données de l'expression par défaut doit correspondre au type de données de la colonne.

L'expression par défaut est utilisée dans les opérations d'insertion qui ne spécifient pas de valeur pour la colonne. S'il n'y a pas de valeur par défaut pour une colonne, elle est NULL.

CONSTRAINT *constraint_name* **:** Le nom optionnel d'une contrainte de colonne ou de table. Si la contrainte est violée, le nom de la contrainte est présente dans les messages d'erreur. Donc les noms de contraintes comme « col doit être positive » peut être utilisés pour communiquer des informations utiles aux applications clientes. (Des doubles guillemets sont nécessaires pour indiquer les noms des contraintes qui contiennent des espaces.) Si un nom de contrainte n'est pas donné, le système en génère un.

NOT NULL **:** Interdiction des valeurs NULL dans la colonne. Il s'agit de la seule contrainte qui ne peut être exprimée sous forme de contrainte de table.

UNIQUE (contrainte colonne), ou

UNIQUE (*column_name* **:** [, … ]) (contrainte table) : La contrainte UNIQUE spécifie qu'un groupe de colonnes d'une table ne peut contenir que des valeurs uniques. Le comportement de la contrainte de table est le même que celui des contraintes de colonnes avec la possibilité supplémentaire de grouper des colonnes.

PRIMARY KEY (contrainte colonne), ou

PRIMARY KEY (*column_name* [, … ]) (contrainte table) : Techniquement, PRIMARY KEY n'est qu'une combinaison de UNIQUE et NOT NULL. Toutefois, identifier un ensemble de colonnes comme clé primaire fournit des informations sur les schémas. En effet, une clé primaire implique que d'autres tables puissent utiliser cet ensemble de colonnes comme identifiant unique de ligne.

Une seule clé primaire peut être spécifiée par table, qu'il s'agisse d'une contrainte de colonne ou de table.

CHECK ( *expression* ) **:** La clause CHECK spécifie une expression de résultat booléen que les nouvelles lignes ou celles mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Si une des lignes de l'opération d'insertion ou de mise à jour produit un résultat **FALSE**, une exception est levée et la base de données n'est pas modifiée. Une contrainte de vérification sur une colonne ne fait référence qu'à la valeur de la colonne tandis qu'une contrainte sur la table fait référence à plusieurs colonnes (par exemple, col1 $<$ col2).

REFERENCES *reftable* [ ( *refcolumn* ) ] [ ON DELETE *action* ] [ ON UPDATE *action* ]] (contrainte de colonne), ou

FOREIGN KEY ( *column* [, ... ] ) REFERENCES *reftable* [ ( *refcolumn* [, ... ] ) ] [ ON DELETE *action* ] [ ON UPDATE *action* ]] (contrainte de table) : Ces clauses spécifient une contrainte de clé étrangère. Cela signifie qu'un groupe de colonnes de la nouvelle table ne peut contenir que des valeurs correspondant à celles des colonnes de référence de la table de référence. Si refcolumn est omis, la clé primaire de la reftable est utilisée. Les colonnes référencées doivent être celles d'une contrainte d'unicité ou de clé primaire dans la table référencée.

Lorsque les données des colonnes référencées sont modifiées, des actions sont réalisées sur les données de la table référençant. La clause ON DELETE spécifie l'action à réaliser lorsqu'une ligne référencée de la table de référence est supprimée. De la même façon, la clause ON UPDATE spécifie l'action à réaliser lorsqu'une colonne référencée est mise à jour. Les actions suivantes sont possibles pour chaque clause :

NO ACTION : Une erreur est produite pour indiquer que la suppression ou la mise à jour entraine une violation de la contrainte de clé étrangère. C'est le comportement par défaut.

CASCADE : La mise à jour ou la suppression de la ligne de référence est propagée à l'ensemble des lignes qui la référencent, qui sont, respectivement, mises à jour ou supprimées.

SET NULL : La valeur de la colonne qui référence est positionnée à NULL.

SET DEFAULT : La valeur de la colonne qui référence est positionnée à celle par défaut.

### Exemples

Créer une table films et une table distributeurs :

```
CREATE TABLE films (
    code        varchar(5) CONSTRAINT premierecle
                    PRIMARY KEY,
    titre       varchar(40) NOT NULL,
    did         int NOT NULL,
    date_prod   date,
    genre       varchar(10),
    duree       int);
```

```
CREATE TABLE distributeurs (
     did     serial PRIMARY KEY,
     nom     varchar(40) NOT NULL CHECK (nom != ''));
```

Une autre manière de définir la table distributeurs, en utilisant des contraintes de type table :

```
CREATE TABLE distributeurs (
    did      serial,
    nom      varchar(40) NOT NULL,
    PRIMARY KEY(did),
    CONSTRAINT nom_non_vide CHECK (nom != ''));
```

Définir une contrainte de clé primaire sur la table films :

```
CREATE TABLE films (
    code        varchar(5),
    titre       varchar(40),
    did         int,
    date_prod   date,
    genre       varchar(10),
    duree       int,
    CONSTRAINT code_titre PRIMARY KEY(code, titre));
```

Affecter une valeur par défaut à la colonne nom, et une valeur par défaut à la colonne modtime, égale à la date et l'heure où la ligne est insérée. Notez que le type serial revient à définir une valeur par défaut égale au dernier numéro de série généré +1 :

```
CREATE TABLE distributeurs (
    name      varchar(40) DEFAULT 'Luso Films',
    did       serial,
    modtime   timestamp DEFAULT current_timestamp);
```

Définir deux contraintes de colonne NOT NULL sur la table distributeurs, dont l'une est explicitement nommée :

```
CREATE TABLE distributeurs (
    did      integer CONSTRAINT no_null NOT NULL,
    nom      varchar(40) NOT NULL);
```

Définir une contrainte d'unicité sur la colonne nom :

```
CREATE TABLE distributeurs (
    did      int,
    nom      varchar(40) UNIQUE);
```

### 4.4.2 ALTER TABLE – Modifier une définition de table

#### Synopsis

```
ALTER TABLE name action [, ... ]
```

```
ALTER TABLE name RENAME column TO new_column
```

**ALTER TABLE** *name* **RENAME TO** *new_name*

où *action* peut être :

**ADD** *column type* [ *column_constraint* [ ... ] ]
**DROP** *column* [ **CASCADE** ]
**ALTER** *column* **TYPE** *type* [ **USING** *expression* ]
**ALTER** *column* **SET DEFAULT** *expression*
**ALTER** *column* **DROP DEFAULT**
**ALTER** *column* { **SET** | **DROP** } **NOT NULL**
**ADD** *table_constraint*
**DROP CONSTRAINT** *constraint_name* [ **CASCADE** ]

## Description

ALTER TABLE modifie la définition d'une table existante. Il existe plusieurs variantes :

ADD COLUMN **:** Ajoute une nouvelle colonne à la table en utilisant une syntaxe identique à celle de CREATE TABLE.

DROP COLUMN **:** Supprime une colonne de la table. Les index et les contraintes de table référençant cette colonne sont automatiquement supprimés. L'option CASCADE doit être utilisée lorsque des objets en dehors de la table dépendent de cette colonne, comme par exemple des références de clés étrangères ou des vues.

SET DATA TYPE **:** Change le type d'une colonne de la table. Les index et les contraintes simples de table qui impliquent la colonne sont automatiquement convertis pour utiliser le nouveau type de la colonne en réanalysant l'expression d'origine. La clause optionnelle USING précise comment calculer la nouvelle valeur de la colonne à partir de l'ancienne ; en cas d'omission, la conversion par défaut est identique à une affectation de transtypage de l'ancien type vers le nouveau. Une clause USING doit être fournie s'il n'existe pas de conversion implicite ou d'assignement entre les deux types.

SET/DROP DEFAULT **:** Ajoute ou supprime les valeurs par défaut d'une colonne. Les valeurs par défaut ne s'appliqueront qu'aux commandes INSERT ultérieures. Elles ne modifient pas les lignes déjà présentes dans la table. Des valeurs par défaut peuvent aussi être créées pour les vues. Dans ce cas, elles sont ajoutées aux commandes **INSERT** de la vue avant que la règle **ON INSERT** de la vue ne soit appliquée.

SET/DROP NOT NULL **:** Modifie l'autorisation de valeurs NULL. SET NOT NULL ne peut être utilisé que si la colonne ne contient pas de valeurs NULL.

ADD *table_constraint* **:** Ajoute une nouvelle contrainte à une table en utilisant une syntaxe identique à CREATE TABLE.

DROP CONSTRAINT **:** Supprime la contrainte de table précisée.

### Paramètres

*name* : Le nom de la table à modifier.

*column* : Le nom d'une colonne, existante ou nouvelle.

*new_column* : Le nouveau nom d'une colonne existante.

*new_name* : Le nouveau nom de la table.

*type* : Le type de données de la nouvelle colonne, ou le nouveau type de données d'une colonne existante.

*table_constraint* : Une nouvelle contrainte de table pour la table.

*constraint_name* : Le nom d'une contrainte existante à supprimer.

CASCADE : Les objets qui dépendent de la colonne ou de la contrainte supprimée sont automatiquement supprimés (par exemple, les vues référençant la colonne).

### Exemples

Ajouter une colonne de type text à une table :

```
ALTER TABLE distributeurs ADD COLUMN adresse text;
```

Supprimer une colonne de table :

```
ALTER TABLE distributeurs DROP COLUMN adresse ;
```

Renommer une colonne existante :

```
ALTER TABLE distributeurs
    RENAME COLUMN adresse TO city;
```

Renommer une table existante :

```
ALTER TABLE distributeurs RENAME TO fournisseurs;
```

Ajouter une contrainte NOT NULL à une colonne :

```
ALTER TABLE distributeurs
    ALTER COLUMN rue SET NOT NULL;
```

Supprimer la contrainte NOT NULL d'une colonne :

```
ALTER TABLE distributeurs
    ALTER COLUMN rue DROP NOT NULL;
```

Ajouter une contrainte de vérification sur une table :

```
ALTER TABLE distributeurs
    ADD CONSTRAINT verif_cp
    CHECK (char_length(code_postal) = 5);
```

Supprimer une contrainte de vérification d'une table :

```
ALTER TABLE distributeurs DROP CONSTRAINT verif_cp;
```

Ajouter une contrainte de clé étrangère à une table :

```
ALTER TABLE distributeurs
    ADD CONSTRAINT dist_fk FOREIGN KEY (adresse)
    REFERENCES adresses (adresse);
```

Ajouter une contrainte unique (multicolonnes) à une table :

```
ALTER TABLE distributeurs
    ADD CONSTRAINT dist_id_codepostal_key
    UNIQUE (dist_id, code_postal);
```

Ajouter une clé primaire nommée automatiquement à une table. Une table ne peut avoir qu'une seule clé primaire.

```
ALTER TABLE distributeurs ADD PRIMARY KEY (dist_id);
```

### 4.4.3 DROP TABLE – Supprimer une table

#### Synopsis

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE ]
```

#### Description

DROP TABLE supprime des tables de la base de données. Seul son propriétaire peut détruire une table. Utilisez DELETE pour supprimer les lignes d'une table sans détruire la table.

DROP TABLE supprime tout index, règle, déclencheur ou contrainte qui existe sur la table cible. Néanmoins, pour supprimer une table référencée par une contrainte de clé étrangère d'une autre table, CASCADE doit être ajouté. (CASCADE ne supprime que la contrainte, pas l'autre table.)

#### Paramètres

IF EXISTS **:** Ne pas renvoyer une erreur si la table n'existe pas. Un message d'avertissement est affiché dans ce cas.

*name* **:** Le nom de la table à supprimer.

CASCADE **:** Les objets qui dépendent de la table sont automatiquement supprimés.

#### Exemples

Supprimer les deux tables films et distributeurs :

```
DROP TABLE films, distributeurs;
```

## 4.5 Manipulation des données

### 4.5.1 INSERT – Insérer de nouvelles lignes dans une table

#### Synopsis

```
INSERT INTO table [ ( column [, ...] ) ]
    { DEFAULT VALUES |
      VALUES ( { expression | DEFAULT } [, ...] )
          [, ...] |
      query }
    [ RETURNING * |
      output_expression [ [ AS ] output_name ] [, ...] ]
```

#### Description

INSERT insère de nouvelles lignes dans une table. Vous pouvez insérer une ou plusieurs lignes spécifiées par les expressions de valeur, ou zéro ou plusieurs lignes provenant d'une requête.

L'ordre des noms des colonnes n'a pas d'importance. Si aucune liste de noms de colonnes n'est donnée, toutes les colonnes de la table sont utilisées dans l'ordre de leur déclaration (les $n$ premiers noms de colonnes si seules $n$ valeurs de colonnes sont fournies dans la clause VALUES ou dans la requête). Les valeurs fournies par la clause VALUES ou par la requête sont associées à la liste explicite ou implicite des colonnes de gauche à droite.

Chaque colonne absente de la liste, implicite ou explicite, des colonnes se voit attribuer sa valeur par défaut, s'il y en a une, ou NULL dans le cas contraire.

Un transtypage automatique est entrepris lorsque l'expression d'une colonne ne correspond pas au type de donnée déclaré.

La clause RETURNING optionnelle fait que INSERT calcule et renvoie le(s) valeur(s) basée(s) sur chaque ligne en cours d'insertion. C'est principalement utile pour obtenir les valeurs qui ont été fournies par défaut, comme un numéro de séquence. Néanmoins, toute expression utilisant les colonnes de la table est autorisée. La syntaxe de la liste RETURNING est identique à celle de la commande SELECT.

#### Paramètres

*table* : Le nom de la table.

*column* : Le nom d'une colonne de table.

DEFAULT VALUES : Toutes les colonnes se voient attribuer leur valeur par défaut.

*expression* : Une expression ou une valeur à affecter à la colonne correspondante.

DEFAULT : La colonne correspondante se voit attribuer sa valeur par défaut.

*query* : Une requête (instruction SELECT) dont le résultat fournit les lignes à insérer. La syntaxe complète de la commande est décrite dans la documentation de l'instruction SELECT.

*output_expression* : Une expression à calculer et renvoyée par la commande INSERT après chaque insertion de ligne. L'expression peut utiliser tout nom de colonne de la table. Indiquez * pour que toutes les colonnes soient renvoyées.

*output_name* : Un nom à utiliser pour une colonne renvoyée.

Si la commande INSERT contient une clause RETURNING, la commande renvoie un résultat similaire à celui d'une instruction SELECT contenant les colonnes et les valeurs définies dans la liste RETURNING, à partir de la liste des lignes insérées par la commande.

### Exemples

Insérer une ligne dans la table films :

```
INSERT INTO films
VALUES ('UA502', 'Bananas', 105, '1971-07-13',
    'Comédie', 82);
```

Dans l'exemple suivant, la colonne longueur est omise et prend donc sa valeur par défaut :

```
INSERT INTO films (code, titre, did, date_prod, genre)
VALUES ('T_601', 'Yojimbo', 106, '1961-06-16',
    'Drame');
```

L'exemple suivant utilise la clause DEFAULT pour les colonnes date plutôt qu'une valeur précise :

```
INSERT INTO films
VALUES ('UA502', 'Bananas', 105, DEFAULT, 'Comédie',
    82);

INSERT INTO films (code, titre, did, date_prod, genre)
VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drame');
```

Insérer une ligne constituée uniquement de valeurs par défaut :

```
INSERT INTO films DEFAULT VALUES;
```

Pour insérer plusieurs lignes en utilisant la syntaxe multi-lignes VALUES :

```
INSERT INTO films (code, titre, did, date_prod, genre)
VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT,
        'Comedy');
```

Insérer dans la table films des lignes extraites de la table tmp_films (la disposition des colonnes est identique dans les deux tables) :

```
INSERT INTO films
SELECT * FROM tmp_films
WHERE date_prod < '2004-05-07';
```

Insérer une ligne simple dans la table distributers, en renvoyant le numéro de séquence généré par la clause DEFAULT :

```
INSERT INTO distributers (did, dname)
VALUES (DEFAULT, 'XYZ Widgets')
RETURNING did;
```

### 4.5.2 DELETE – Supprimer des lignes d'une table

#### Synopsis

```
DELETE FROM table [ WHERE condition ]
```

#### Description

**DELETE** supprime de la table spécifiée les lignes qui satisfont la clause **WHERE**. Si la clause **WHERE** est absente, toutes les lignes de la table sont supprimées.

#### Paramètres

*table* **:** Le nom de la table.

*condition* **:** Une expression retournant une valeur de type boolean. Seules les lignes pour lesquelles cette expression renvoie **TRUE** seront supprimées.

#### Exemples

Supprimer tous les films qui ne sont pas des films musicaux :

```
DELETE FROM films WHERE genre != 'Comédie musicale';
```

Effacer toutes les lignes de la table films :

```
DELETE FROM films;
```

### 4.5.3 UPDATE – Modifier des lignes dans une table

#### Synopsis

```
UPDATE table
SET { column = { expression | DEFAULT } |
        ( column [ , ... ] )
          = ( { expression | DEFAULT } [ , ... ] ) }
    [ , ... ]
[ WHERE condition ]
```

#### Description

**UPDATE** modifie les valeurs des colonnes spécifiées pour toutes les lignes qui satisfont la condition. Seules les colonnes à modifier doivent être mentionnées dans la clause **SET** ; les autres colonnes conservent leur valeur.

#### Paramètres

*table* **:** Le nom de la table à mettre à jour.

*colonne* **:** Le nom d'une colonne dans table. Il ne faut pas inclure le nom de la table dans la spécification d'une colonne cible – par exemple, **UPDATE** tab **SET** tab.col = 1 est invalide.

*expression* **:** Une expression à affecter à la colonne. L'expression peut utiliser les anciennes valeurs de cette colonne et d'autres colonnes de la table.

DEFAULT **:** Réinitialise la colonne à sa valeur par défaut (qui vaut **NULL** si aucune expression par défaut ne lui a été affectée).

*condition* **:** Une expression qui renvoie une valeur de type boolean. Seules les lignes pour lesquelles cette expression renvoie true sont mises à jour.

#### Exemples

Changer le mot « Drame » en « Dramatique » dans la colonne genre de la table films :

```
UPDATE films SET genre = 'Dramatique'
WHERE genre = 'Drame';
```

Ajuster les entrées de température et réinitialiser la précipitation à sa valeur par défaut dans une ligne de la table temps :

```
UPDATE temps
SET temp_basse = temp_basse+1,
    temp_haute = temp_basse+15,
    prcp = DEFAULT
WHERE ville = 'San Francisco' AND date = '2005-07-03';
```

Utiliser une autre syntaxe pour faire la même mise à jour :

```
UPDATE temps
SET (temp_basse, temp_haute, prcp)
    = (temp_basse+1, temp_basse+15, DEFAULT)
WHERE ville = 'San Francisco' AND date = '2005-07-03';
```

Incrémenter le total des ventes de la personne qui gère le compte d'« *Acme Corporation* », en utilisant une sous-requête dans la clause **WHERE** :

```
UPDATE employes
SET total_ventes = total_ventes + 1
WHERE id = (
  SELECT vendeur FROM comptes
  WHERE nom = 'Acme Corporation');
```

### 4.5.4 SELECT – Récupèrer des lignes d'une table ou d'une vue

#### Synopsis

```
SELECT [ DISTINCT ]
    * | expression [ AS output_name ] [, ...]
    [ FROM from_item ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ DESC ]
      [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT count ]
    [ OFFSET start ]
```

où *from_item* peut être :

```
    table_name [ AS alias ]
    ( select ) AS alias
    from_item [ NATURAL ] join_type from_item
      [ ON join_condition |
        USING ( join_column [, ...] )]
```

#### Description

**SELECT** récupère des lignes de zéro ou plusieurs tables. traitement général de **SELECT** est le suivant :

1. Tous les éléments de la liste **FROM** sont calculés. (Chaque élément dans la liste **FROM** est une table réelle ou virtuelle ; voir Clause **FROM** ci-dessous.)

2. Si la clause **WHERE** est spécifiée, toutes les lignes qui ne satisfont pas les conditions sont éliminées de l'affichage. (Voir Clause **WHERE** ci-dessous.)

3. Si la clause **GROUP BY** est spécifiée, l'affichage est divisé en groupes de lignes qui correspondent à une ou plusieurs valeurs. Si la clause **HAVING** est présente, elle élimine les groupes qui ne satisfont pas la condition donnée. (Voir Clause **GROUP BY** et Clause **HAVING** ci-dessous.)

4. Les lignes retournées sont traitées en utilisant les expressions de sortie de **SELECT** pour chaque ligne sélectionnée. (Voir Liste **SELECT** ci-dessous.)

5. En utilisant les opérateurs **UNION**, **INTERSECT** et **EXCEPT**, l'affichage de plusieurs instructions **SELECT** peut être combiné pour former un ensemble unique de résultats. L'opérateur **UNION** renvoie toutes les lignes qui appartiennent, au moins, à l'un des ensembles de résultats. L'opérateur **INTERSECT** renvoie toutes les lignes qui sont dans tous les ensembles de résultats. L'opérateur **EXCEPT** renvoie les lignes qui sont présentes dans le premier ensemble de résultats mais pas dans le deuxième. Dans les trois cas, les lignes dupliquées sont éliminées sauf si **ALL** est spécifié. (Voir Clause **UNION**, Clause **INTERSECT** et Clause **EXCEPT** ci-dessous.)

6. Si la clause **ORDER BY** est spécifiée, les lignes renvoyées sont triées dans l'ordre spécifié. Si **ORDER BY** n'est pas indiqué, les lignes sont retournées dans l'ordre qui permet la réponse la plus rapide du système. (Voir Clause **ORDER BY** ci-dessous.)

7. Si **DISTINCT** est spécifié, les lignes dupliquées du résultat sont supprimées. (Voir Clause **DISTINCT** ci-dessous.)

8. Si les clauses **LIMIT** ou **OFFSET** sont spécifiées, l'instruction **SELECT** ne renvoie qu'un sous-ensemble de lignes de résultats. (Voir clause **LIMIT** ci-dessous.)

### Paramètres

**Liste SELECT.** La liste **SELECT** (entre les mots clés **SELECT** et **FROM**) spécifie les expressions qui forment les lignes en sortie de l'instruction **SELECT**. Les expressions font généralement référence aux colonnes traitées dans la clause **FROM** (cf section 4.6 pour plus d'informations sur les expressions).

Comme pour une table, chaque colonne de sortie d'un **SELECT** a un nom. Dans un **SELECT** simple, ce nom est juste utilisé pour donner un titre à la colonne pour l'affichage, mais quand le **SELECT** est une sous-requête d'une requête plus grande, le nom est vu par la grande requête comme le nom de colonne de la table virtuelle produite par la sous-requête. Pour indiquer le

nom à utiliser pour une colonne de sortie, écrivez **AS** *output_name* après l'expression de la colonne. Si vous n'indiquez pas de nom de colonne, un nom est choisi automatiquement par *PostgreSQL*.

Un nom de colonne de sortie peut être utilisé pour se référer à la valeur de la colonne dans les clauses **ORDER BY** et **GROUP BY**, mais pas dans la clause **WHERE** ou **HAVING** ; à cet endroit, vous devez écrire l'expression.

***** peut être utilisé, à la place d'une expression, dans la liste de sortie comme raccourci pour toutes les colonnes des lignes sélectionnées. De plus, *table_name*.***** peut être écrit comme raccourci pour toutes les colonnes de cette table (dans ces cas, il est impossible de spécifier de nouveaux noms avec **AS**).

La liste **SELECT** (en particulier en combinaison avec la clause **DISTINCT**) correspond à la *projection* en algèbre relationnelle.

**Clause DISTINCT.** Si **DISTINCT** est spécifié, toutes les lignes dupliquées sont supprimées de l'ensemble de résultats (une ligne est conservée pour chaque groupe de lignes dupliquées).

**Clause FROM.** La clause **FROM** spécifie une ou plusieurs tables source pour le **SELECT**. La clause **FROM** peut contenir les éléments suivants :

*table_name* **:** Le nom d'une table existante ou d'une vue.

*alias* **:** Un nom de substitution pour l'élément **FROM** contenant l'alias. Un alias est utilisé par brièveté ou pour lever toute ambigüité lors d'auto-jointures (la même table est parcourue plusieurs fois). Quand un alias est fourni, il cache complètement le nom réel de la table ou fonction ; par exemple, avec **FROM** foo **AS** f, le reste du **SELECT** doit faire référence à cet élément de **FROM** par f et non pas par foo.

*select* **:** Un sous-**SELECT** peut apparaitre dans la clause **FROM**. Il agit comme si sa sortie était transformée en table temporaire pour la durée de cette seule commande **SELECT**. Le sous-**SELECT** doit être entouré de parenthèses et un alias doit lui être fourni. Une commande **VALUES** peut aussi être utilisée ici.

*join_type* **:** **JOIN**, **LEFT JOIN**, **RIGHT JOIN** ou **FULL JOIN**. La jointure permet de mettre plusieurs tables en relation à l'aide d'un produit cartésien et d'une condition de jointure. Cf section 3.2.3 pour plus de détails. Une condition de jointure doit être spécifiée, à choisir parmi :

- **ON** *join_condition*,
- **NATURAL**,
- **USING** (*join_column* [, ...]).

Une clause **JOIN** combine deux éléments **FROM**. Les parenthèses peuvent être utilisées pour déterminer l'ordre d'imbrication. En l'absence de parenthèses, les **JOIN** sont imbriqués de gauche à droite. **JOIN** réalise un produit cartésien, restreint par la condition de jointure.

**LEFT JOIN** renvoie toutes les lignes du produit cartésien qualifié (c'est-à-dire toutes les lignes combinées qui satisfont la condition de jointure), plus une copie de chaque ligne de la table de gauche pour laquelle il n'y a pas de ligne à droite qui satisfasse la condition de jointure. La ligne de gauche est étendue à la largeur complète de la table jointe par insertion de valeurs **NULL** pour les colonnes de droite. Seule la condition de la clause **JOIN** est utilisée pour décider des lignes qui correspondent. Les conditions externes sont appliquées après coup.

À l'inverse, **RIGHT JOIN** renvoie toutes les lignes jointes plus une ligne pour chaque ligne de droite sans correspondance (complétée par des **NULL** pour le côté gauche). C'est une simple aide à la notation car il est aisément convertible en **LEFT** en inversant les entrées gauche et droite.

**FULL JOIN** renvoie toutes les lignes jointes, plus chaque ligne gauche sans correspondance (étendue par des **NULL** à droite), plus chaque ligne droite sans correspondance (étendue par des **NULL** à gauche).

ON *join_condition* **:** *join_condition* est une expression qui retourne une valeur de type boolean (comme une clause **WHERE**) qui spécifie les lignes d'une jointure devant correspondre.

USING (*join_column* [, ...]) : **USING** (a, b, ...) n'est qu'un raccourci pour **ON** t_gauche.a = t_droite.a **AND** t_gauche.b = t_droite.b ...

De plus, **USING** implique l'affichage d'une seule paire des colonnes correspondantes dans la sortie de la jointure.

NATURAL **:** **NATURAL** est un raccourci pour une liste **USING** qui mentionne toutes les colonnes de même nom dans les deux tables.

**Clause WHERE.** La clause **WHERE** optionnelle a la forme générale

**WHERE** *condition*

où *condition* est une expression dont le résultat est de type boolean (cf section 4.6 pour plus d'informations sur les expressions). Toute ligne qui ne satisfait pas cette condition est éliminée de la sortie. Une ligne satisfait la condition si elle retourne vrai quand les valeurs réelles de la ligne sont substituées à toute référence de variable.

La clause **WHERE** correspond à la *restriction* en algèbre relationnelle.

**Clause GROUP BY.** La clause **GROUP BY** optionnelle a la forme générale

**GROUP BY** *expression* [ , . . . ]

**GROUP BY** condense en une seule ligne toutes les lignes sélectionnées qui partagent les mêmes valeurs pour les expressions regroupées. *expression* peut être le nom d'une colonne ou une expression quelconque formée de valeurs de colonnes.

Les fonctions d'agrégat, si utilisées, sont calculées pour toutes les lignes composant un groupe, produisant une valeur séparée pour chaque groupe (alors que sans **GROUP BY**, un agrégat produit une valeur unique calculée pour toutes les lignes sélectionnées). Quand **GROUP BY** est présent, les expressions du **SELECT** ne peuvent faire référence qu'à des colonnes groupées, sauf à l'intérieur de fonctions d'agrégat, la valeur de retour d'une colonne non-groupée n'étant pas unique. Voir sections 3.2.4 (page 38) et 4.6.7 (page 66) pour plus d'informations sur les fonctions d'agrégat.

**Clause HAVING.** La clause optionnelle **HAVING** a la forme générale

**HAVING** *condition*

où condition est identique à celle spécifiée pour la clause **WHERE**.

**HAVING** élimine les lignes groupées qui ne satisfont pas à la condition. **HAVING** est différent de **WHERE** : **WHERE** filtre les lignes individuelles avant l'application de **GROUP BY** alors que **HAVING** filtre les lignes groupées créées par **GROUP BY**. Chaque colonne référencée dans condition doit faire référence sans ambigüité à une colonne groupée, sauf si la référence apparait dans une fonction d'agrégat.

Même en l'absence de clause **GROUP BY**, la présence de **HAVING** transforme une requête en requête groupée. Cela correspond au comportement d'une requête contenant des fonctions d'agrégats sans clause **GROUP BY**. Les lignes sélectionnées ne forment qu'un groupe, la liste du **SELECT** et la clause **HAVING** ne peuvent donc faire référence qu'à des colonnes à l'intérieur de fonctions d'agrégats. Une telle requête ne produira qu'une seule ligne si la condition **HAVING** est réalisée, aucune dans le cas contraire.

**Clauses UNION, INTERSECT et EXCEPT.** Ces clauses ont la forme générale :

*select_statement*
{ **UNION** | **INTERSECT** | **EXCEPT** } [ **ALL** ]
*select_statement*

*select_statement* est une instruction **SELECT** sans clause **ORDER BY** ou **LIMIT**. Les opérateurs **UNION**, **INTERSECT** et **EXCEPT** calculent respectivement l'union, l'intersection et la différence ensemblistes. Les deux instructions **SELECT** qui représentent les opérandes directes de l'opérateur doivent

produire le même nombre de colonnes et les colonnes correspondantes doivent être d'un type de données compatible.

Sauf lorsque l'option **ALL** est spécifiée, les doublons sont supprimés. **ALL** empêche l'élimination des lignes dupliquées. Cela accélère significativement les calculs, et sera utilisé si possible.

**Clause ORDER BY.** La clause optionnelle **ORDER BY** a la forme générale :

```
ORDER BY expression [ DESC ]
  [ NULLS { FIRST | LAST } ] [, ...]
```

La clause **ORDER BY** impose le tri des lignes de résultat suivant les expressions spécifiées. Si deux lignes sont identiques suivant l'expression la plus à gauche, elles sont comparées avec l'expression suivante et ainsi de suite. Chaque expression peut être le nom d'une colonne (élément de la liste **SELECT**). Il est aussi possible d'utiliser des expressions quelconques dans la clause **ORDER BY**, ce qui inclut des colonnes qui n'apparaissent pas dans la liste résultat du **SELECT**. Ainsi, l'instruction suivante est valide :

```
SELECT nom FROM distributeurs ORDER BY code;
```

Un mot clé **DESC** (descendant) peut être ajouté après toute expression de la clause **ORDER BY**. Par défaut, les **NULL** sont considérés comme plus grands que les non **NULL**. On peut modifier ce comportement avec les clauses **NULLS LAST** ou **NULLS FIRST**.

Notez que les options de tri s'appliquent seulement à l'expression qu'elles suivent. Par exemple, **ORDER BY** x, y **DESC** ne signifie pas la même chose que **ORDER BY** x **DESC**, y **DESC**.

**Clause LIMIT/OFFSET.** La clause **LIMIT** est constituée de deux parties indépendantes :

```
LIMIT count
OFFSET start
```

*count* spécifie le nombre maximum de lignes à renvoyer alors que *start* spécifie le nombre de lignes à passer avant de commencer à renvoyer des lignes. Lorsque les deux clauses sont spécifiées, *start* lignes sont passées avant de commencer à compter les nombre lignes à renvoyer.

### Exemples

Joindre la table films avec la table distributeurs :

```
SELECT f.titre, f.did, d.nom, f.date_prod, f.genre
FROM distributeurs d JOIN films f USING (did)
```

| titre | did | nom | date_prod | genre |
|---|---|---|---|---|
| The Third Man | 101 | British Lion | 1949-12-23 | Drame |
| The African… | 101 | British Lion | 1951-08-11 | Romantique |
| | | … | | |

Additionner la colonne longueur de tous les films, grouper les résultats par genre :

```
SELECT genre, sum(longueur) AS total FROM films
GROUP BY genre;
```

| genre | total |
|---|---|
| Action | 07:34 |
| Comédie | 02:58 |
| Drame | 14:28 |
| Musical | 06:42 |
| Romantique | 04:38 |

Additionner la colonne longueur de tous les films, grouper les résultats par genre et afficher les groupes dont les totaux font moins de cinq heures :

```
SELECT genre, sum(longueur) AS total
FROM films
GROUP BY genre
HAVING sum(longueur) < '5:00';
```

| genre | total |
|---|---|
| Comedie | 02:58 |
| Romantique | 04:38 |

L'exemple suivant montre comment trier les résultats individuels en fonction du contenu de la colonne *nom* :

```
SELECT * FROM distributeurs ORDER BY nom;
```

| did | nom |
|---|---|
| 109 | 20th Century Fox |
| 110 | Bavaria Atelier |
| 101 | British Lion |
| 107 | Columbia |
| 102 | Jean Luc Godard |
| 113 | Luso films |
| 104 | Mosfilm |
| 103 | Paramount |
| 106 | Toho |
| 105 | United Artists |
| 111 | Walt Disney |
| 112 | Warner Bros. |
| 108 | Westward |

L'exemple suivant présente l'union des tables distributeurs et acteurs, restreignant les résultats à ceux de chaque table dont la première lettre est un W. Le mot clé **ALL** est omis, ce qui permet de n'afficher que les lignes distinctes.

**Distributeurs**

| did | nom |
|---|---|
| 108 | Westward |
| 111 | Walt Disney |
| 112 | Warner Bros. |
| | … |

**Acteurs**

| id | nom |
|---|---|
| 1 | Woody Allen |
| 2 | Warren Beatty |
| 3 | Walter Matthau |
| | … |

```
SELECT distributeurs.nom
    FROM distributeurs
    WHERE distributeurs.nom LIKE 'W%'
UNION
SELECT actors.nom
    FROM acteurs
    WHERE acteurs.nom LIKE 'W%';
```

| nom |
|---|
| Walt Disney |
| Walter Matthau |
| Warner Bros. |
| Warren Beatty |
| Westward |
| Woody Allen |

| Prédicat | Résultat |
|---|---|
| NULL = NULL | NULL |
| NULL IS NULL | TRUE |
| NULL IS NOT NULL | FALSE |
| NULL AND TRUE | NULL |
| NULL AND FALSE | FALSE |
| NULL OR FALSE | NULL |
| NULL OR TRUE | TRUE |

TAB. 4.1 : Table de vérité de la valeur NULL

## 4.6 Expressions et opérateurs

*PostgreSQL* supporte un grand nombre de fonctions et d'opérateurs utilisables sur les types de données de base. Il est même possible de définir ses propres fonctions et opérateurs. Attention, tous ces opérateurs ne sont pas inclus dans le standard SQL, mais la plupart sont supportés par les SGBD du marché.

### 4.6.1 Opérateurs logiques

On peut utiliser les opérateurs logiques habituels **AND**, **OR** et **NOT**. Ceux-ci sont capables de travailler avec la valeur **NULL** (cf la table de vérité 4.1).

### 4.6.2 Opérateurs de comparaison

Les opérateurs de comparaison habituels sont disponibles (<, >, <=, >=, =, <>/ !=). Ils ne fonctionnent évidemment que pour les types de données pour lesquels cela a du sens.

En plus de ces opérateurs, on trouve la construction spéciale **BETWEEN**. a **BETWEEN** x **AND** y est équivalent à a >= x **AND** a <= y. Le test inverse est effectué par **NOT BETWEEN**.

La plupart du temps, toute comparaison avec un type **NULL** renvoie **NULL** (par exemple, 3 > **NULL** = **NULL**). D'un point de vue logique, le type **NULL** respecte la table de vérité 4.1.

### 4.6.3 Opérateurs mathématiques

On pourra utiliser les opérateurs +, −, *, /, % (modulo). De nombreux autres opérateurs et fonctions sont disponibles (valeur absolue, puissances, racines carrées, logarithmes…) N'hésitez pas à consulter la documentation officielle de *PostgreSQL* [7] pour une liste plus complète.

### 4.6.4 Opérateurs de chaines

|| **:** concaténation. 'Post' || 'greSQL' donne par exemple 'PostgreSQL'.

length(*string*) **:** retourne la longueur de la chaine *string*.

lower(*string*) **:** convertit *string* en minuscules.

D'autres fonctions sont disponibles (extraction de sous-chaines, etc.) Consultez la documentation pour plus d'information.

### 4.6.5 Correspondance de motifs

#### Avec LIKE

```
string LIKE motif
string NOT LIKE motif
```

L'expression **LIKE** renvoie **TRUE** si *string* est contenue dans l'ensemble de chaines représenté par le motif. (L'expression **NOT LIKE** renvoie **FALSE** si **LIKE** renvoie **TRUE** et vice versa.)

Si le motif ne contient ni signe pourcent ni tiret bas, alors il ne représente que la chaine elle-même ; dans ce cas, **LIKE** agit exactement comme l'opérateur d'égalité. Un tiret bas (_) dans motif correspond à un seul caractère, un signe pourcent (%) à toutes les chaines de zéro ou plusieurs caractères.

Quelques exemples :

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

Le modèle **LIKE** correspond toujours à la chaine entière. Du coup, pour faire correspondre une séquence à l'intérieur d'une chaine, le motif doit donc commencer et finir avec un signe pourcent.

Pour faire correspondre un vrai tiret bas ou un vrai signe de pourcentage sans correspondance avec d'autres caractères, le caractère correspondant dans motif doit être précédé du caractère d'échappement (*antislash*).

L'opérateur **ILIKE** est identique à **LIKE** mais ne tient pas compte de la différence entre majuscules et minuscules.

#### Expressions rationnelles

On peut effectuer des comparaisons à l'aide d'expressions rationnelles grâce à l'opérateur ~ et ses variantes ~* (ne différencie pas les majuscules des minuscules), !~ (inverse le test) ou !~*.

Quelques exemples :

```
'abc' ~ 'abc'     true
'abc' ~ '^a'      true
'abc' ~ '(b|d)'   true
'abc' ~ '^(b|c)'  false
```

### Recherche plein texte

**LIKE** et les expressions rationnelles ne suffisent pas pour rechercher des données dans de grands ensembles de textes (implantation d'un moteur de recherche sur des documents complets). *PostgreSQL* dispose de fonctions spécifiques et performantes pour cela, capables de valider des mots dérivés (« satisfait » alors que l'on recherche « satisfaire », par exemple), et de classer les résultats par pertinence ([7], chapitre 12).

## 4.6.6 Opérateurs sur date/heure

Il est possible d'effectuer des opérations +, − entre données de type date/heure (**date**, **time** ou **timestamp**), et * et / entre une donnée de type date/heure et un entier ou un flottant. De plus, on peut utiliser les mots-clés **CURRENT_TIME**, **CURRENT_DATE** et **CURRENT_TIMESTAMP** pour obtenir l'heure et/ou la date courante.

Une fonction utile, **EXTRACT**, permet d'extraire l'année, le mois, le jour, etc. d'un type date/heure :

**EXTRACT** (*champ* **FROM** *source*)

*champ* peut être : *century, day, decade, dow* (jour de la semaine, de dimanche – 0 – à samedi – 6), *doy* (jour de l'année, de 1 à 366), *epoch* (nombre de jours depuis le 1er janvier 1970), *minute, month, quarter* (trimestre, de 1 à 4), *second, week, year*.

## 4.6.7 Fonctions d'agrégat

Les fonctions d'agrégat calculent une valeur unique à partir d'un ensemble de valeurs en entrée (généralement obtenu grâce à la clause **GROUP BY** d'une requête).

avg(*expression*) **:** la moyenne arithmétique de toutes les valeurs en entrée

count(*) **:** nombre de lignes en entrée

count(*expression*) **:** nombre de lignes en entrée pour lesquelles l'expression n'est pas **NULL**.

max(*expression*) **:** valeur maximale de l'expression pour toutes les valeurs en entrée

min(*expression*) **:** valeur minimale de l'expression pour toutes les valeurs en entrée

sum(*expression*) **:** somme de l'expression pour toutes les valeurs en entrée

string_agg(*expression*, *séparateur*) **:** liste des expressions de chaque valeur en entrée, séparées par le séparateur indiqué (généralement une virgule).

### 4.6.8 Expressions de sous-requêtes

Toutes les formes d'expressions documentées dans cette section renvoient des résultats booléens (vrai/faux).

#### Opérateur EXISTS

**EXISTS** ( *subquery* )

L'argument d'**EXISTS** est une instruction **SELECT** arbitraire ou une sous-requête. La sous-requête est évaluée pour déterminer si elle renvoie des lignes. Si elle en renvoie au moins une, le résultat d'**EXISTS** est vrai (**TRUE**) ; si elle n'en renvoie aucune, le résultat d'**EXISTS** est faux (**FALSE**).

La sous-requête peut faire référence à des variables de la requête englobante qui agissent comme des constantes à chaque évaluation de la sous-requête.

Puisque le résultat ne dépend que d'un éventuel retour de lignes, et pas de leur contenu, la liste des champs retournés par la sous-requête n'a normalement aucun intérêt. Une convention de codage habituelle consiste à écrire tous les tests **EXISTS** sous la forme **EXISTS**(**SELECT** 1 **WHERE** ...)

L'exemple suivant, simpliste, ressemble à une jointure sur col2 mais il sort au plus une ligne pour chaque ligne de tab1, même s'il y a plusieurs correspondances dans les lignes de tab2 :

```
SELECT col1 FROM tab1
WHERE EXISTS(
    SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

#### Opérateur IN

*expression* **IN** (*subquery*)

Le côté droit est une sous-expression entre parenthèses qui ne peut retourner qu'une seule colonne. L'expression de gauche est évaluée et comparée à chaque ligne du résultat de la sous-requête. Le résultat de **IN** est vrai (**TRUE**) si une ligne équivalente de la sous-requête est trouvée. Le résultat est faux (**FALSE**) si aucune ligne correspondante n'est trouvée (ce qui inclut le cas spécial de la sous-requête ne retournant aucune ligne).

L'opérateur **NOT IN** donne le résultat inverse à **IN**.

L'exemple suivant est équivalent à l'exemple avec **EXISTS** ci-dessus :

```
SELECT col1 FROM tab1
WHERE col2 IN (SELECT col2 FROM tab2);
```

#### Opérateur ANY

*expression operator* **ANY** (*subquery*)

Le côté droit est une sous-requête entre parenthèses qui ne doit retourner qu'une seule colonne. L'expression du côté gauche est évaluée et comparée à chaque ligne du résultat de la sous-requête à l'aide de l'opérateur indiqué, ce qui doit aboutir à un résultat booléen. Le résultat de **ANY** est vrai (**TRUE**) si l'un des résultats est vrai. Le résultat est faux (**FALSE**) si aucun résultat vrai n'est trouvé (ce qui inclut le cas spécial de la requête ne retournant aucune ligne).

**IN** est équivalent à = **ANY**.

### Opérateur ALL

*expression operator* **ALL** (*subquery*)

Le côté droit est une sous-requête entre parenthèses qui ne doit renvoyer qu'une seule colonne. L'expression gauche est évaluée et comparée à chaque ligne du résultat de la sous-requête à l'aide de l'opérateur, ce qui doit renvoyer un résultat booléen. Le résultat de ALL est vrai (**TRUE**) si toutes les lignes renvoient **TRUE** (ce qui inclut le cas spécial de la sous-requête ne retournant aucune ligne). Le résultat est faux (**FALSE**) si un résultat faux est découvert.

**NOT IN** est équivalent à != ALL.

Par exemple, retrouver les réalisateurs dont tous les films font plus de deux heures :

```
SELECT * FROM realisateur
WHERE '2:00' < ALL (
  SELECT longueur FROM films
  WHERE films.rid = realisateur.rid)
```

Les réalisateurs ayant réalisé au moins un film de moins d'une heure :

```
SELECT * FROM realisateur
WHERE '1:00' > ANY (
  SELECT longueur FROM films
  WHERE films.rid = realisateur.rid)
```

Chapitre 5

# Conception

## Sommaire


**5.1 Introduction** . . . . . **70**
**5.2 Pourquoi une méthode de conception ?** . . . **70**
5.2.1 Les risques d'une mauvaise conception . . . 70
5.2.2 Éviter les anomalies . . . . . 71
5.2.3 Règles de normalisation . . . . . 72
**5.3 Modèle conceptuel** . . . . . **73**
5.3.1 Objets et classes . . . . . 74
5.3.2 Associations . . . . . 74
5.3.3 Attributs . . . . . 77
5.3.4 Classe faible . . . . . 78
5.3.5 Notes . . . . . 80
5.3.6 Héritage . . . . . 80
5.3.7 Contraintes . . . . . 81
**5.4 Modèle physique : le modèle Relationnel** . . **81**
5.4.1 Classe → table . . . . . 82
5.4.2 Association un-vers-plusieurs (ou un-vers-un) → clé étrangère . . . . . 82
5.4.3 Association plusieurs-vers-plusieurs ou association non-binaire → table associative . . 84
5.4.4 Classe faible . . . . . 86
5.4.5 Héritage . . . . . 86
5.4.6 Contraintes . . . . . 87
**5.5 Rétro-ingéniérie** . . . . . **87**

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

TAB. 5.1 : Base de données non normalisée

## 5.1 Introduction

La conception de bases de données s'articule autour de deux modèles : le *modèle conceptuel* et le *modèle physique*. Lors de l'élaboration d'une base de données, on réalise tout d'abord un modèle conceptuel. Il décrit les choix de gestion adoptés par l'entreprise. Ce niveau de description répond à la question « quoi ? », c'est-à-dire « que veut-on faire qui reste vrai quelles que soient les solutions d'organisation et les solutions techniques à mettre en œuvre ? » Il s'agit d'un modèle de conception « de haut niveau », déconnecté des futures solutions techniques, à la fois complet pour le développeur et compréhensible pour un non-spécialiste. Il est particulièrement utile pour élaborer une solution en accord avec un employeur, et sera au centre du cahier des charges. La réalisation de tels modèles sera étudiée plus en détail lors du module M2104 « Bases de la conception orientée objets » au deuxième semestre.

Lors de la réalisation technique, ce modèle sera converti automatiquement en *modèle physique*, adapté à la solution logicielle choisie (ici, une base de données relationnelle), et automatiquement convertible en code source (ici, en SQL). Notez qu'il existe des *ateliers de génie logiciel* qui offrent notamment la possibilité de tracer un modèle conceptuel, et qui transforment automatiquement ce modèle en code (SQL).

## 5.2 Pourquoi une méthode de conception ?

### 5.2.1 Les risques d'une mauvaise conception

Un des principaux objectifs d'une bonne conception est de structurer la base de données et de limiter le plus possible la redondance d'informations (même information recopiée plusieurs fois). Prenons par exemple la base de données simple du tableau 5.1, constituée d'une seule table (et pourrait donc être stockée dans un tableur, voire un simple fichier texte). Pouvez-vous identifier tous les problèmes potentiels de cette base de données pourtant très

simple ?

**Anomalies d'insertion.** Tout d'abord, on peut remarquer que rien ne peut empêcher de représenter plusieurs fois le même film. Pire : il est possible d'insérer plusieurs fois le même film, mais en le décrivant de manière différente (plusieurs réalisateurs différents, par exemple). Lors de la conception d'une base de données, il est important de se demander ce qui distingue deux films l'un de l'autre, et à quel moment on peut dire que la même information a été répétée. Peut-il y avoir deux films avec le même titre ? Si non, on doit pouvoir s'assurer qu'il n'y a pas deux lignes dans la table avec le même titre. Si oui, il faut absolument déterminer quel est l'ensemble des informations permettant de caractériser un film de manière unique.

**Anomalies de modification.** La redondance d'information entraine également des anomalies de mise à jour. Supposons que l'on modifie l'année de naissance de Hitchcock pour la ligne *Vertigo* et pas pour la ligne *Psychose*. On obtient alors des informations incohérentes. On peut d'ailleurs se poser la même question que précédemment : peut-on dire qu'il n'y a qu'un seul réalisateur nommé Hitchcock, et qu'il ne doit donc y avoir qu'une seule année de naissance pour ce réalisateur ?

**Anomalies de suppression.** Il est impossible dans cette base de stocker un metteur en scène sans film. Si on supprime le dernier film d'un metteur en scène, par exemple *Titanic*, on va effacer du même coup toute information sur James Cameron.

### 5.2.2 Éviter les anomalies

Une bonne méthode évitant les anomalies ci-dessus consiste à

- être capable de représenter individuellement les films et les réalisateurs, de manière à ce qu'une action sur l'un n'entraine pas systématiquement une action sur l'autre,
- définir une méthode d'identification d'un film ou d'un réalisateur, qui permette d'assurer que la même information est représentée une seule fois,
- préserver le lien entre les films et les réalisateurs, mais sans introduire de redondance.

Dans l'exemple précédent, la solution serait de distinguer les deux *classes d'objets* Film et Réalisateur, de les stocker dans deux tables distinctes, et de réaliser une *association* entre eux, via des *clés*. Si le principe de *normalisation* permet d'améliorer une base existante, le plus simple est encore de réaliser dès le départ une modélisation conceptuelle qui permet d'arriver naturellement à ce résultat.

### 5.2.3 Règles de normalisation

La normalisation est un ensemble de règles que doit vérifier une base de données pour limiter au maximum la redondance d'informations. Valider un ensemble de règles permet d'atteindre une certaine *forme normale*. Pour atteindre la deuxième forme normale, il faut d'abord valider la première, et ainsi de suite. Il y a en tout huit niveaux de forme normale.

Bien qu'il ne soit pas toujours souhaitable de limiter au maximum la redondance d'informations (il peut y avoir des problèmes de performances s'il faut recalculer sans cesse certaines informations, et il y a plus de risques de perte de données si celles-ci ne sont présentes qu'en un seul exemplaire), on cherchera en général à construire dans un premier temps un modèle le plus « simple » possible, et en cas de problème de performances ou de fragilité excessive des informations (quand même assez rares), d'introduire des solutions adéquates.

**Première forme normale :** tous les attributs sont atomiques (une seule information par champ) et constants dans le temps (on stocke une date de naissance, pas l'âge).

**Deuxième forme normale :** tous les attributs d'une relation (table) sont soit des identifiants (« clés primaires »), soit dépendent des identifiants de la table[1].

**Troisième forme normale :** tous les attributs d'une relation qui ne sont pas des identifiants doivent dépendre *directement* des identifiants (il ne doit pas y avoir d'attribut intermédiaire).

**Forme normale de Boyce-Codd :** les identifiants d'une relation ne doivent pas dépendre des autres attributs.

Une conception dont les relations respectent la forme normale de Boyce-Codd (*« The key, the whole key, nothing but the key »*) a très peu de chances de contenir des redondances d'information. Les formes normales supplémentaires permettent de détecter des cas plus particuliers de redondance.

**Quatrième forme normale :** les relations ne doivent pas pouvoir être décomposées en deux relations plus petites, de sorte que l'on puisse reconstituer la relation d'origine par jointure[2]. Par exemple, on modélise des informations sur les capacités d'intervention d'un garagiste sur certains types de pannes et certaines marques de voiture par une relation (Employé, Type panne, Marque). Si, en réalité, aucune panne n'est spécifique à une marque, il serait plus efficace de modéliser avec deux relations (Employé, Type panne) et (Employé, Marque). La jointure entre ces deux relations permet de retrouver la relation d'origine.

[1] On parle de dépendance fonctionnelle.
[2] On parle alors de dépendance multivaluée.

**Cinquième forme normale :** comme la précédente, mais avec plus de deux relations (seule la jointure *simultanée* de toutes les relations décomposées permet de retrouver la relation d'origine).

Les autres formes normales (domaine clé, sixième forme normale) sont rarement utilisées.

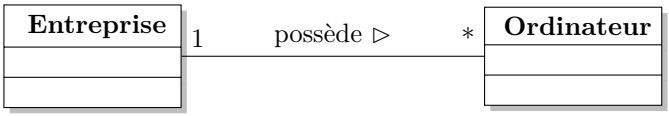
## 5.3 Modèle conceptuel : le diagramme de classes ou modèle E/A

On peut utiliser différentes syntaxes différentes pour réaliser un modèle conceptuel. Dans le cadre de ce cours, nous utiliserons la syntaxe du *diagramme de classes* de l'*UML* (*Unified Modelling Language*). Bien qu'en bases de données relationnelles, on utilise plus souvent la notion de *modèle E/A (Entité/Association)*, le principe est suffisamment proche du diagramme de classes pour ne pas avoir à introduire une nouvelle terminologie et notation pour ce cours. La notion de *type d'entité* du modèle E/A est extrêmement proche de la notion de *classe* en conception orientée objet. Une autre syntaxe couramment utilisée est celle du MCD (Modèle Conceptuel de Données) de la méthode Merise, ou certaines syntaxes spécifiques aux modèles E/A (*Crow's Foot*, par exemple). La principale différence entre les MCD Merise et *UML* est que la position des cardinalités est inversée.

Le diagramme de classes permet de décrire l'application visée. Il ne s'agit pas nécessairement d'une représentation exacte de la réalité, mais uniquement d'une représentation pertinente compte tenu des objectifs visés. Il est souvent utile de simplifier la réalité pour mieux répondre aux *besoins* définis.

**Utilisation de l'*UML* pour représenter un modèle E/A.** Pour information, voici une liste des principales restrictions de la syntaxe d'*UML* pour représenter un diagramme E/A :

- Nous n'utiliserons pas la notion de *paquetage* ; *PostgreSQL* et d'autres SGBD disposent de la notion de *schema* permettant d'organiser les objets de la base de données, mais nous avons choisi de ne pas utiliser cette fonctionnalité pour l'instant (cf [7], chapitre 5.7, pour plus d'information),
- Nous ne définirons pas de *méthodes* dans les classes : seuls les *attributs* nous intéressent,
- Nous n'utiliserons pas la notion de *visibilité* (public/protégé/privé),
- Nous ne distinguerons pas les *associations*, *compositions* et *agrégations*. Bien que ces notions puissent être distinguées en SQL (notamment au niveau des options **ON DELETE** des contraintes de clés étrangères), elles sont connues pour prêter à confusion,




Exemple 5.1 : Deux classes associées

- Les associations ne sont jamais *orientées* (le principe même des bases de données relationnelles permet de s'affranchir des notions d'orientation),
- Le stéréotype « id » permettra d'identifier les *identifiants* des classes, qui serviront à déterminer les *clés primaires* des tables (au cœur du principe des bases de données relationnelles),
- Nous utiliserons les *discriminants* d'UML pour modéliser les *classes faibles*, utiles en bases de données relationnelles pour modéliser des relations dotées d'identifiants composés (sans introduire d'identifiant artificiel).

### 5.3.1 Objets et classes

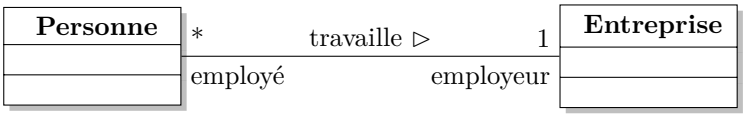
Un objet peut être définie comme une chose capable d'exister de manière indépendante, et pouvant être identifiée de manière unique. Un objet est une abstraction d'une réalité potentiellement complexe dans un domaine donné. On utilise habituellement le terme « objet » pour parler d'un aspect du monde réel pouvant être distingué d'autres aspects du monde réel. Un objet peut être un objet physique (une maison, une voiture…), un évènement (une vente, une révision automobile…), ou encore un concept (une transaction, une commande…) Les objets sont toujours des *noms* : un ordinateur, un employé, une chanson, un théorème mathématique…

Des objets partageant les même caractéristiques (plusieurs ordinateurs différents, plusieurs voitures…) sont regroupés par une *classe* d'objets. En *UML*, on représente une classe par un rectangle séparé en trois parties (cf exemple 5.1). La partie supérieure contient le nom de la classe. En base de données relationnelles, il faut de plus que chaque objet puisse être *identifié* de manière unique et immuable.

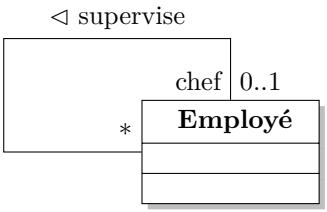
### 5.3.2 Associations

Une association permet de représenter un lien entre deux classes. Les associations sont toujours des *verbes*, reliant au moins deux noms. Par exemple : une association « possède » entre une entreprise et un ordinateur, une association « supervise » entre un employé et un département, une association « exécuter » entre un artiste et une œuvre, une association « a prouvé » entre un mathématicien et un théorème.

Les associations sont représentées par une ligne entre deux classes, éventuellement surmontée du nom de l'association et d'une petite flèche indiquant




Exemple 5.2 : Une association avec les rôles précisés




Exemple 5.3 : Une association réflexive

le sens de lecture (cf exemple 5.1 page 74).

### Rôles

Pour éviter tout risque de confusion, on peut préciser un *rôle* à chaque extrémité de l'association (cf exemple 5.2). C'est particulièrement utile dans le cas d'une *association réflexive* (reliant deux classes du même type, cf exemple 5.3).

### Cardinalités

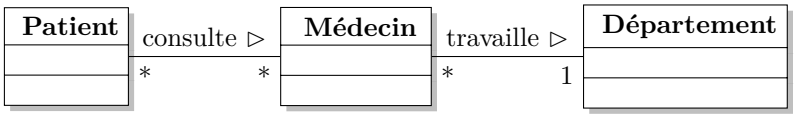
Les cardinalités sont extrêmement importantes en conception de bases de données. Elles définissent la manière dont les classes sont liées entre elles. Il y a trois types d'associations :

- un-vers-un (*one-to-one*),
- un-vers-plusieurs ou plusieurs-vers-un (*one-to-many* ou *many-to-one*),
- plusieurs-vers-plusieurs (*many-to-many*).

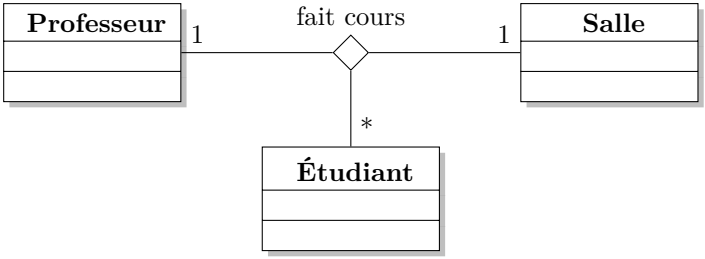
On représente les cardinalités en indiquant le chiffre « 1 » (un) ou une étoile « * » (plusieurs) à chaque extrémité d'une association. Le diagramme de l'exemple 5.4 page suivante se lit comme suit :

### Association plusieurs-vers-plusieurs entre médecin et patient :

- un patient peut consulter plusieurs médecins,
- un médecin peut être consulté par plusieurs patients,




Exemple 5.4 : Une association plusieurs-vers-plusieurs et une association plusieurs-vers-un




Exemple 5.5 : Association non-binaire

**Association plusieurs-vers-un entre médecin et département :**

- un médecin travaille dans un seul département,
- un département peut accueillir plusieurs médecins,

**Attention au sens de lecture pour l'association un-vers-plusieurs !** Notez qu'elle se lit assez naturellement en suivant le sens de l'association.

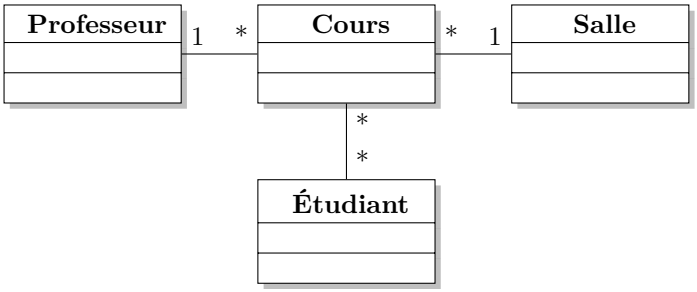
L'association un-vers-un est rarement utilisée : dans ce cas on peut généralement fusionner les classes.

Le choix des cardinalités est un *choix de conception* très important : il sera difficile de revenir dessus une fois le développement de l'application entamé. Posez-vous toujours la question : « Est-on bien sûr qu'il n'y aura *jamais* de médecin travaillant dans plusieurs départements ? » Bien qu'il soit tentant de mettre systématiquement des associations plusieurs-vers-plusieurs, moins contraignantes, il faut savoir que ce genre d'association est plus difficile à gérer, y compris par l'utilisateur final de l'application. Une application fonctionne généralement mieux quand elle est plus simple : évitez les usines à gaz !

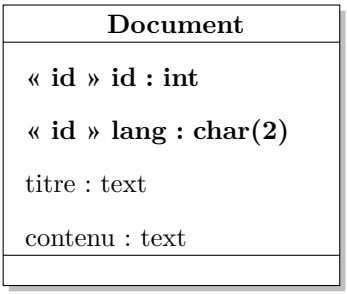
## Association non-binaire

L'exemple 5.5 montre une association entre plus de deux classes. L'association lie à la fois le professeur, la salle, et l'étudiant. Les cardinalités se lisent comme suit :

- pour un professeur et un étudiant donnés, il y a une et une seule salle,




EXEMPLE 5.6 : Alternative à l'association non-binaire



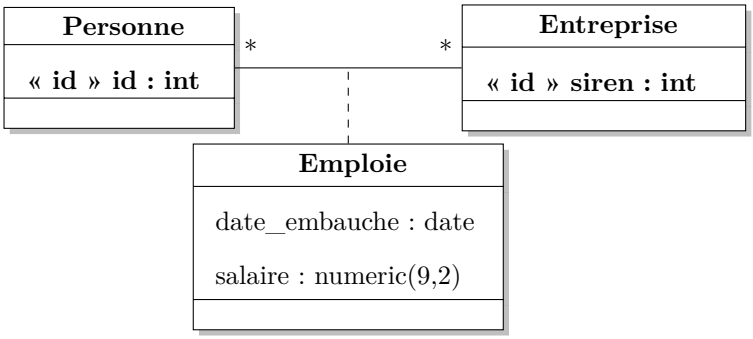

EXEMPLE 5.7 : Une classe avec un identifiant composite

- pour un étudiant et une salle donnés, il y a un et un seul professeur,
- pour un professeur et une salle donnés, il y a plusieurs étudiants.

Notez que ce genre d'association a tendance à prêter à confusion. On cherchera en général une modélisation alternative. L'exemple 5.6 permet de représenter les mêmes informations que le modèle de l'exemple 5.5 page 76 de manière plus claire et plus souple. Attention, la sémantique n'est pas tout à fait identique. Contrairement à l'exemple d'origine, l'exemple 5.6 permet de distinguer plusieurs cours avec le même professeur, dans la même salle, avec plusieurs groupes d'étudiants différents.

### 5.3.3 Attributs

Les attributs permettent de caractériser une classe ou une association. Chaque attribut a un *type* (cf section 4.3 page 43 pour une liste des types de données courants).




Exemple 5.8 : Une classe associative

### Attributs de classes

Comme montré dans l'exemple 5.7 page 77, les attributs des objets sont listés dans la partie intermédiaire du rectangle. Quand on développe un diagramme de classes destiné à être implanté dans une base de données relationnelle, il faut que chaque classe comprenne *obligatoirement* un *identifiant*, que l'on note par le stéréotype « id » face à l'attribut correspondant. L'identifiant doit permettre d'identifier de manière unique et immuable chaque objet. Le plus souvent, on introduit un numéro de série généré automatiquement.

Un identifiant peut être *composite*, c'est-à-dire qu'il peut être composée de plusieurs attributs. Notez qu'on considère toujours qu'il n'y a qu'une seule clé. Dans l'exemple 5.7 page 77, l'identifiant est le couple (id, lang) : il peut y avoir plusieurs documents avec le même id, plusieurs de la même langue, mais chaque couple (id, lang) est unique. Ici, cette construction permet de stocker simplement un même document (même id) en plusieurs langues.

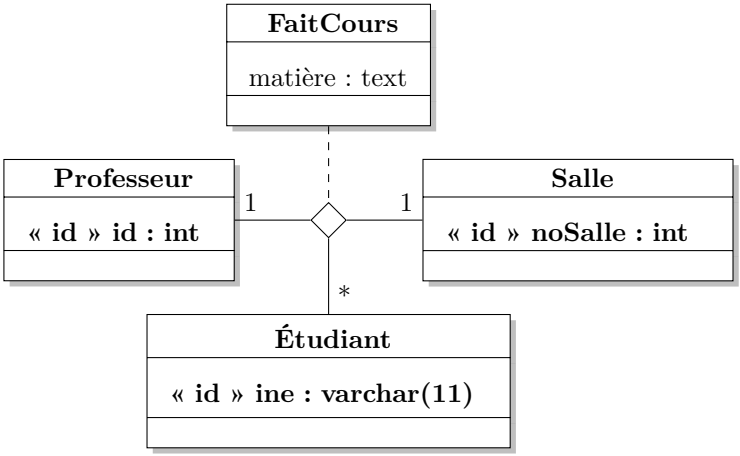
Le choix des identifiants aura un impact assez important sur les futures performances de la base de données : les attributs seront *indexées* afin de les retrouver plus facilement. Plus l'identifant est compact (idéalement, un simple nombre entier), plus le calcul des index sera efficace. Il vaut mieux éviter d'utiliser des champs texte, potentiellement longs, comme identifiants.

### Classe associative

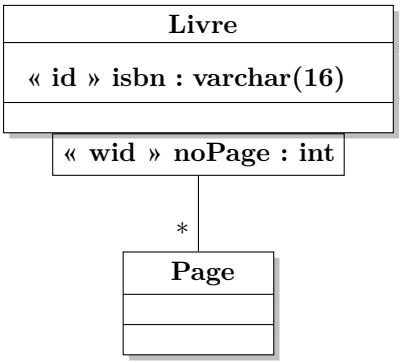
Les *classes associatives* permettent de mettre des attributs sur une association (cf exemples 5.8 à 5.9 pages 78–79). Notez que les classes associatives n'ont pas d'identifiant : on pourra le déduire des classes associées.

## 5.3.4 Classe faible

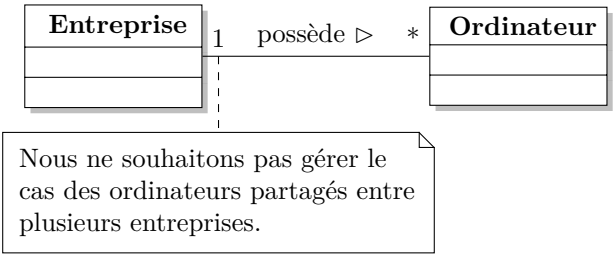
Une *classe faible* est une *classe composant* qui n'a pas d'identifiant unique à proprement parler. Elle est identifiée par rapport à son ou ses composite(s), à l'aide d'un ou plusieurs *qualifiant(s)* (ou *identifiant(s) faible(s)*). Un qualifiant en UML est représenté dans un cadre placé à l'extrémité de l'association.



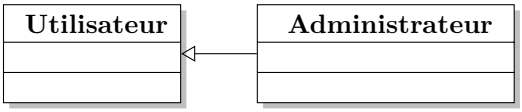

Exemple 5.9 : Une classe associative sur une association non-binaire




Exemple 5.10 : Une classe faible




EXEMPLE 5.11 : Une note dans un diagramme UML




EXEMPLE 5.12 : Une association d'héritage : un administrateur est un utilisateur

On peut relier plusieurs qualifiants identiques par une contrainte « {same} » (cf section 5.3.7 page suivante). La cardinalité du côté des composites est obligatoirement de « 1 » (l'association est obligatoire). Pour modéliser « un livre est composé de pages numérotées », on utilisera une modélisation avec une classe faible plutôt qu'une simple association un-vers-plusieurs, afin d'éviter l'introduction d'un identifiant artificiel au niveau des pages (cf exemple 5.10 page 79). Plusieurs pages peuvent avoir le même numéro, mais pas dans le même livre.

### 5.3.5 Notes

N'oubliez pas qu'il est toujours possible d'insérer des notes dans un diagramme *UML* (équivalentes aux commentaires dans un langage de programmation, cf exemple 5.11). Elles peuvent servir à justifier des choix de modélisation ou à lever des ambigüités.

### 5.3.6 Héritage

L'héritage représente une association « est-un ».

Bien qu'il soit possible de représenter des associations d'héritage (généralisation/spécialisation, cf exemple 5.12) avec des bases de données relationnelles, elles n'ont pas été conçues pour cela et cette notion peut poser des difficultés inattendues. (Des *bases de données orientées objets* sont à l'étude depuis le milieu des années 1990 mais peinent à percer.)

### 5.3.7 Contraintes

#### Cardinalités et associations optionnelles

On peut utiliser les cardinalités pour définir si une association est *optionnelle* ou non. Indiquer « 0..1 » ou « 0..* » en cardinalité indique que l'association est optionnelle (un médecin peut ne travailler pour aucun département, et un département peut n'employer aucun médecin). Indiquer « 1..1 » ou « 1..* » indique que l'association est obligatoire (un médecin doit toujours être associé à un département, et un département doit employer au moins un médecin). « 1 » seul est équivalent à « 1..1 », et « * » seul est équivalent à « 0..* ». Le choix de rendre une association optionnelle ou non a beaucoup moins d'importance que la nature de celle-ci, et pourra être remis en cause beaucoup plus facilement au cours du développement de l'application. Rendre une association obligatoire est une *contrainte* qui pourra poser des problèmes d'utilisabilité. Par exemple, si vous interdisez à un département de n'avoir aucun médecin, et à un médecin de ne travailler pour aucun département, il sera difficile de créer le premier médecin et le premier département. Plus généralement, les cardinalités de type « 1..* » sont délicates à implanter et à utiliser dans les BDR, et on essaiera de les éviter si possible.

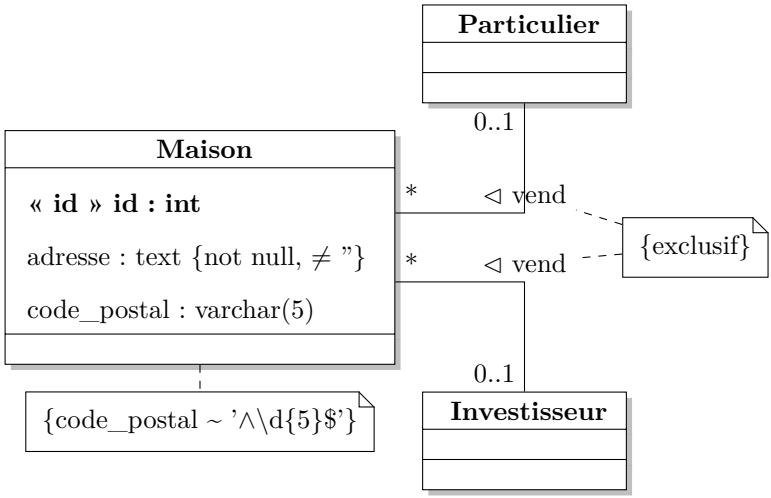
Finalement, il est possible de préciser un nombre minimal et maximal pour chaque extrémité de l'association. Par exemple, « 1..6 » signifie « de un à six ». En pratique, cela revient à une association un-vers-plusieurs ou plusieurs-vers-plusieurs avec une contrainte supplémentaire (assez difficile à implanter).

#### Contraintes supplémentaires

On peut définir des contraintes supplémentaires entre accolades {}, soit directement sur les attributs, soit en utilisant des notes. On peut écrire les contraintes en langage naturel, en SQL (comme dans l'exemple 5.13 page suivante) ou en *OCL* (*Object Constraint Language*) [1].

## 5.4 Modèle physique : le modèle Relationnel

Le diagramme de classes est un modèle *conceptuel*. Il permet de représenter simplement une abstraction du monde réel. Cependant, il est nécessaire de passer par une phase de *traduction* (ou *compilation*) pour être implanté dans un SGBDR. Le *modèle relationnel* est un *modèle physique*, c'est-à-dire un diagramme montrant directement les *tables*, les *clés primaires* et les *contraintes* telles qu'elles seront réellement implantées dans le SGBDR. La traduction du modèle relationnel en langage de définition des données SQL (**CREATE TABLE** ...) est ensuite trivial (cf section 2.4 page 23 et section 4.4.1 page 44).




Exemple 5.13 : Des contraintes dans un diagramme UML

La transformation du diagramme de classes en modèle relationnel se fait en appliquant un ensemble de règles. **Il n'y a plus de choix conceptuels à faire à ce niveau !** La plupart des ateliers de génie logiciel réalisent automatiquement cette transformation et permettent d'obtenir directement du code SQL à partir du diagramme de classes.

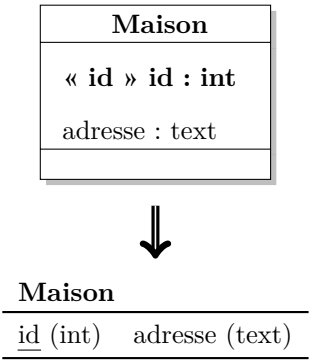
### 5.4.1 Classe → table

Chaque classe (sauf classes associatives) du diagramme de classes devient directement une relation, c'est-à-dire une table de la BDR. Chaque attribut de la classe devient une colonne de la table, de même type. Les éventuelles contraintes (en particulier les clés primaires) sont conservées. L'exemple 5.14 page ci-contre montre la transformation d'une classe en table.

### 5.4.2 Association un-vers-plusieurs (ou un-vers-un) → clé étrangère

Une association un-vers-plusieurs se traduit par l'introduction d'une *clé étrangère*. On rajoute tous les attributs composant la clé primaire de la classe « un » dans la table correspondant à la classe « plusieurs ». On ajoute ensuite une contrainte de clé étrangère (**FOREIGN KEY**/**REFERENCES**) afin d'assurer l'*intégrité référentielle*. Une association obligatoire se traduit par une contrainte « not null » au niveau de la clé étrangère.

Si l'association dispose d'une classe associative, on fusionne tous les attributs de la classe associative dans la classe « plusieurs ». Si l'association est optionnelle, on peut ajouter la contrainte suivante : si la clé étrangère




Exemple 5.14 : Compilation d'une classe en table

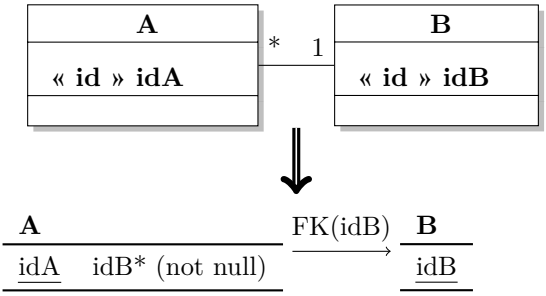



Fig. 5.1 : Compilation d'une association un-vers-plusieurs (association obligatoire)

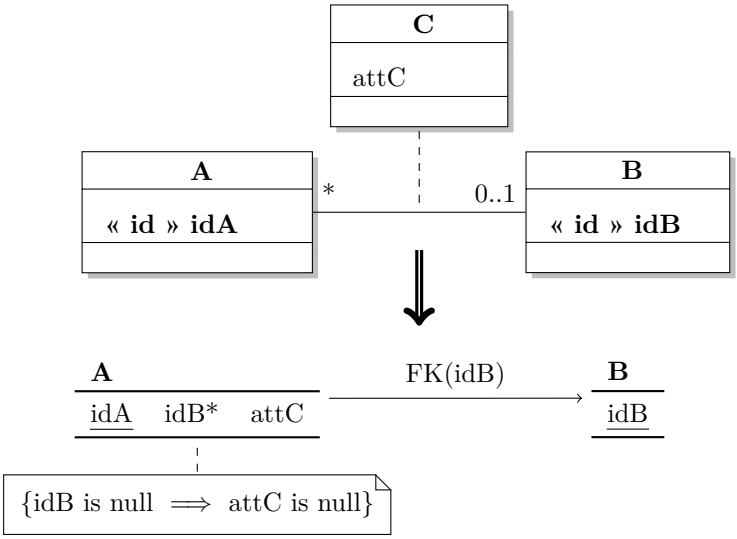



FIG. 5.2 : Compilation d'une association un-vers-plusieurs (association optionnelle et classe associative)

est **NULL**, alors tous les attributs de la classe associative doivent être **NULL** également.

**Attention au sens de l'association un-vers-plusieurs !** Observez la figure 5.1 page 83 et la figure 5.2 : la contrainte de clé étrangère « pointe » toujours vers la clé primaire de la classe placée du côté « un » de l'association un-vers-plusieurs. En effet, l'attribut rajouté dans la table A agit comme un *pointeur* en C/C++ (ou une *référence* en Java), et ne peut effectivement représenter qu'une association simple !

Dans le cas d'une association un-vers-un, il est souvent possible de simplifier le modèle en *fusionnant* les deux classes.

### 5.4.3 Association plusieurs-vers-plusieurs ou association non-binaire → table associative

Pour représenter une association non-binaire ou plusieurs-vers-plusieurs par des relations, il faut introduire une table supplémentaire, appelée *table associative*. Ses champs seront les clés primaires (complètes) de chaque classe liée par l'association (ce sont des clés étrangères). Comme toute table, une table associative doit obligatoirement avoir une clé primaire. Celle-ci est définie comme une clé primaire composite, composée de l'ensemble des clés primaires des tables liées par l'association. Les attributs de la table associative sont donc généralement à la fois clés primaires et étrangères.

S'il y a une classe associative, ses attributs sont placés dans la table asso-

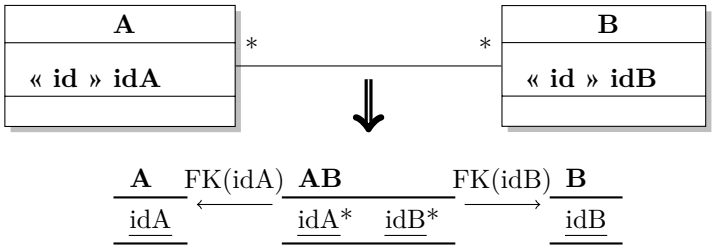



FIG. 5.3 : Compilation d'une association plusieurs-vers-plusieurs

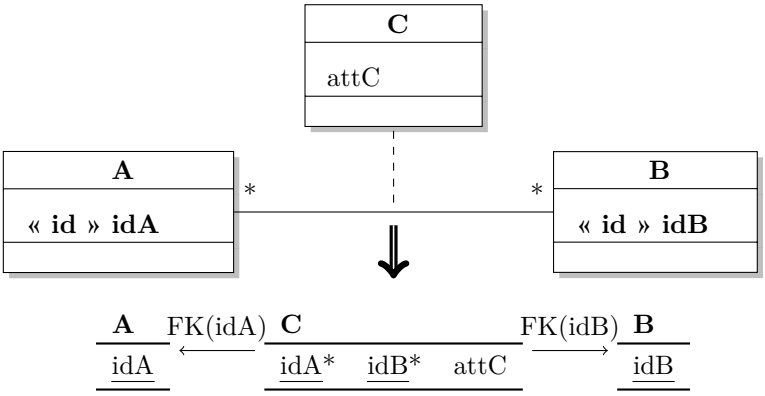



FIG. 5.4 : Compilation d'une association plusieurs-vers-plusieurs avec classe associative

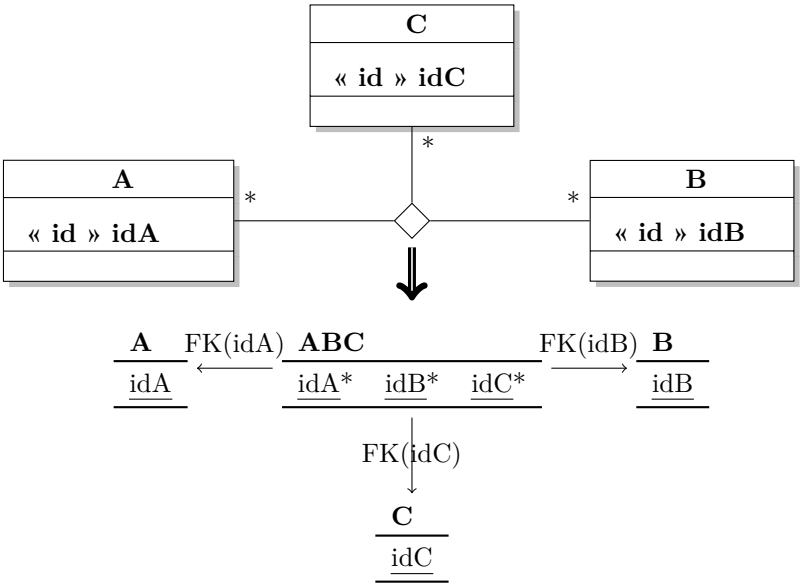



FIG. 5.5 : Compilation d'une association non-binaire

ciative. D'éventuelles cardinalités minimales ou maximales sont traduites en contraintes. Les figures 5.3 à 5.5 pages 85–86 résument ces règles de conversion.

### 5.4.4 Classe faible

La figure 5.6 page suivante résume les règles de conversion d'une classe faible. La clé primaire d'une table représentant une classe faible (**B** sur la figure) devient un composite formé de l'identifiant de la classe qualifiante (idA), et du qualifiant lui-même (l'*identifiant* idB). On ajoute de plus une clé étrangère de type (idA **REFERENCES** A).

### 5.4.5 Héritage

Dans une association d'héritage, toute la hiérarchie des spécialisations/-généralisations d'une classe partage le même identifiant. Une contrainte de clé étrangère permet d'assurer qu'un objet spécialisé dispose des données plus générales, comme dans la figure 5.7 page ci-contre.

Attention, la notion d'héritage n'est pas fidèlement représentée en bases de données relationnelle. Par exemple, il est possible de convertir un objet d'une classe en une autre classe plus spécialisée (simplement en ajoutant une ligne dans la table correspondant à la classe spécialisée), ce que l'on ne peut

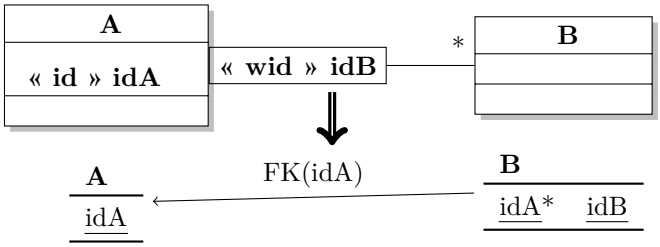



FIG. 5.6 : Compilation d'une classe faible

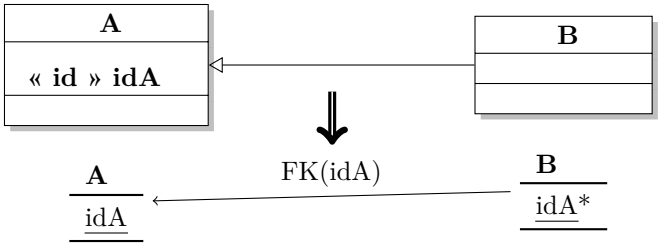



FIG. 5.7 : Compilation d'une association d'héritage

pas faire en programmation par objets classique.

### 5.4.6 Contraintes

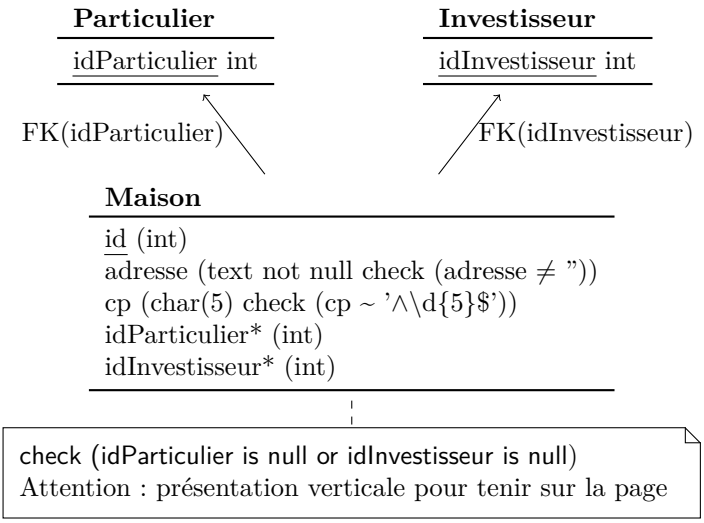
Il faut généralement adapter les contraintes à la nouvelle version du modèle. À ce stade, il une réécriture en SQL permet une implantation directe (cf exemple 5.15 page suivante).

## 5.5 Rétro-ingéniérie

La rétro-ingéniérie (*reverse engineering*) consiste à rétro-compiler un modèle physique (modèle relationnel voire code SQL) en modèle conceptuel (diagramme de classes). Comme deux diagrammes de classes différents peuvent donner lieu à deux modèles relationnels identiques, cette conversion n'est pas automatisable et nécessite une intervention humaine.

La rétro-ingéniérie est particulièrement utile comme travail préalable à la refonte d'un système d'information pour lequel on ne dispose pas des documents de conception.

Sauriez-vous trouver un diagramme de classes correspondant à la base de données de la page 3 ? Un exemple de solution est présenté sur la figure 5.8 page suivante.




EXEMPLE 5.15 : Modèle relationnel avec contraintes correspondant à l'exemple 5.13 page 82

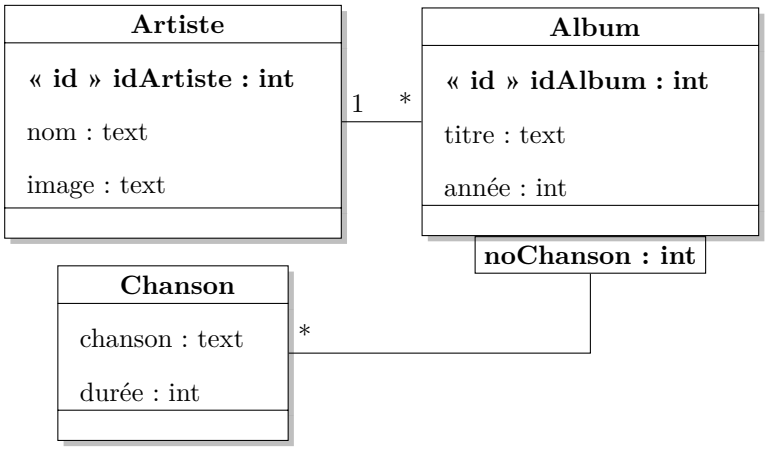



FIG. 5.8 : Un diagramme de classes correspondant à la base de données de la page 3.

# Bibliographie


[1] L. Audibert. *UML 2 – de l'apprentissage à la pratique (cours et exercices)*. Ellipses et http://laurent-audibert.developpez.com/Cours-UML/, 2009.

[2] O. Bonaventure et al. *APP0*. École Polytechnique de Louvain, sept. 2008.

[3] F. Celaia. *Quel SGBD choisir ?* http://fadace.developpez.com/sgbdcmp/. 2003–2010.

[4] E. F. Codd. « A Relational Model of Data for Large Shared Data Banks ». In : *Communications of the ACM* 13.6 (1970), p. 377–387.

[5] Ministère de l'Enseignement Supérieur et de la Recherche. *DUT Informatique. Programme Pédagogique National*. http://www.enseignementsup-recherche.gouv.fr/cid53575/programmes-pedagogiques-nationaux-d.u.t.html. 2013.

[6] The PostgreSQL Global Development Group. *PostgreSQL 9.4*. http://www.postgresql.org/. 2009–2015.

[7] The PostgreSQL Global Development Group et Communauté francophone de PostgreSQL. *Documentation PostgreSQL 9.4*. 1996–2015. URL : http://docs.postgresql.fr/9.4/.

# Annexe A

# Consignes de travail

Texte repris du livret APP0 de l'École Polytechnique de Louvain [2].

## A.1 Travailler en groupe

La formation d'OMGL3 se fait sous la forme d'APP (Apprentissage par Problèmes). Une des caractéristiques de cette méthode est d'optimiser la participation active de chaque étudiant. Individuellement, chacun d'entre vous contribue selon son style et ses ressources à la progression efficace de la rencontre et au climat constructif des échanges. De plus, pour faciliter le déroulement d'un tutorial, il est conseillé aux étudiants de remplir 3 rôles spécifiques :

### Animateur

- S'assure que le groupe suit les étapes prévues,
- Veille à ce que le contenu de la discussion soit noté par le secrétaire,
- Anime la discussion :
  - distribue la parole, suscite/sollicite la participation ou modère les interventions,
  - amène le groupe à clarifier les idées développées,
  - réalise des synthèses au besoin ;
- S'assure du respect du timing : informe le groupe régulièrement (« il nous reste 30 minutes pour cette séance »...)

**Scribe**

- Note au tableau l'essentiel des échanges (support et mémoire de la discussion du groupe),
- Ne filtre pas les informations notées,
- Organise le tableau en fonction des étapes (de manière à garder la trace de toute la réflexion → ne pas effacer).

**Secrétaire**

- Garde une trace écrite et complète de la production du groupe,
- Transmet cette trace à tous les membres du groupe et au tuteur.

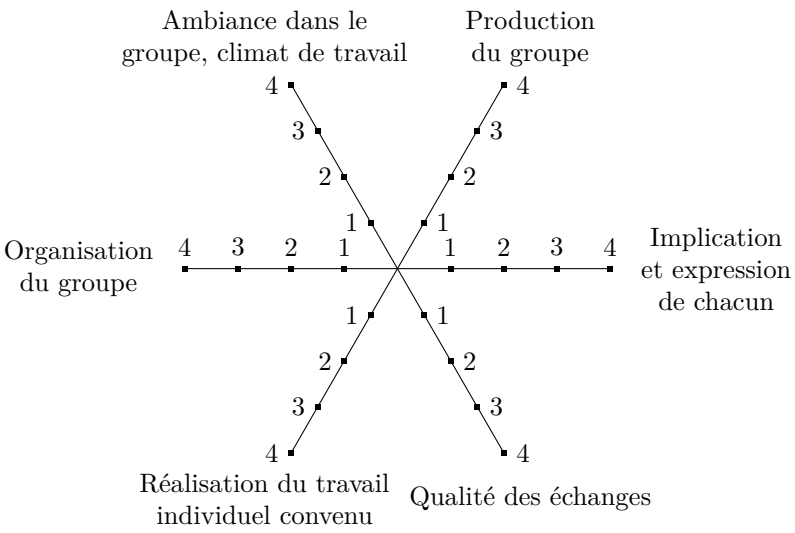
Lors des séances, l'enseignant fait office de *tuteur* :

- Il ne fait pas partie du groupe d'apprenants,
- Il guide le groupe :
  - l'empêche de s'égarer !
  - l'incite à aller plus loin…
- Il n'est pas là pour vous donner un cours (si c'était le cas, vous seriez tous regroupés en auditoire),
- Il connait la réponse au problème mais c'est à vous, étudiants, de faire le travail. Vous ne serez donc pas étonné qu'il refuse parfois de répondre directement aux questions que vous vous posez. Ce sera le cas notamment s'il estime que cette question n'a pas été débattue préalablement au sein du groupe.

## A.2 Travail individuel

Pourquoi faire du travail individuel ?

- Le vrai but est que tout le monde apprenne, pas uniquement que le problème soit bien résolu !
- Ce n'est pas le groupe qui doit devenir compétent mais bien chacun de ses membres !
- Le travail collectif est certes important mais l'APP vise à rendre chaque étudiant compétent.
- C'est la raison pour laquelle chaque APP fait l'objet d'une évaluation individuelle. Le travail réalisé entre les séances de groupe est la manière la plus efficace et la plus simple de se préparer à cette évaluation. Avant la fin de chaque problème, chaque étudiant sera amené à présenter sa solution individuelle aux autres membres du groupe.




# A.3 Évaluation du travail en groupe (questionnaire)

## A.3.1 Les axes (quelques critères d'évaluation)

Indiquez sur chacun des 6 axes figurant sur l'étoile (figure A.3.1) votre niveau d'appréciation générale entre 0 (« très insatisfaisant ») et 4 (« très satisfaisant »). Ensuite, reliez les points.

**Production du groupe.** Le groupe a produit quelque chose de satisfaisant et cette production est réellement le résultat d'un effort *collectif*.

**Implication et expression de chacun.** Chacun des participants a contribué de manière significative à l'efficacité du groupe, le groupe a donné l'occasion à chacun de ses membres d'exprimer son point de vue, les participants en retrait ont été sollicités.

**Qualité des échanges.** Il y eu suffisamment d'interactions entres les différents membres du groupe, ces échanges ont permis de faire émerger des points de vue différents pour traiter le problème, les temps de mises en commun ont permis à chacun de confronter sa compréhension du problème et des notions travaillées…

**Réalisation du travail individuel convenu.** Les membres du groupe ont fait leur part de travail individuel entre les séances, tous les membres du groupe ont mené à bien leurs responsabilités…

**Organisation du travail.** Le groupe est parvenu à coordonner ses activités, les réunions étaient efficaces, le groupe est resté centré sur la tâche à accomplir, le groupe a fait suffisamment usage du tableau, le groupe s'est réparti des rôles : un secrétaire a gardé des traces des échanges, un animateur a joué son rôle, le timing a été respecté…

**Ambiance dans le groupe, climat de travail.** Bonne entente entre les membres du groupe, les participants s'aident et s'encouragent mutuellement, le groupe est arrivé à surmonter ses divergences de vue, personne n'est arrivé à imposer son point de vue…

### A.3.2 Questions ouvertes

1. Déterminez deux points qui ont bien fonctionné pour le travail en groupe

2. Déterminez deux points qui ont mal fonctionné pour le travail en groupe

3. Quelles sont les leçons à tirer de cette expérience ? Si c'était à refaire, que feriez-vous – quel engagement prendriez-vous – pour que cela fonctionne mieux ? Pensez aux travaux de groupe qui se présenteront prochainement durant votre formation.

4. Êtes-vous satisfait des connaissances ou des compétences acquises lors de la résolution de ce problème ? Commentaires à propos de ce que vous avez appris en informatique.

5. Autres commentaires et suggestions à propos de ce problème.

# Annexe B

# Problème 1 : Gestion d'un club hippique (1)

## B.1 Le problème

Le club Hippique « Galop d'Azur » gère les inscriptions de ses adhérents aux stages qu'il organise autour d'une base de données dont vous trouverez des extraits : tableaux B.1 à B.2 pages 96–97. Les adhérents sont classés en 4 niveaux numérotés de 1 (débutant) à 4 (expert). Un adhérent ne pourra s'inscrire qu'à un stage de son niveau, voire d'un niveau inférieur. Un niveau (de 1 à 4) est aussi attribué aux chevaux car certains ne conviennent qu'à des cavaliers expérimentés alors que d'autres peuvent être montés par des cavaliers débutants.

## B.2 Travail demandé

### B.2.1 Préparation

L'objectif du travail en groupe est de préparer efficacement la réalisation **individuelle** du problème. N'oubliez pas que l'objectif prioritaire n'est pas de répondre correctement à toutes les questions, mais d'acquérir les compétences correspondant au module. *Ne vous répartissez pas les tâches :* chaque membre du groupe doit réfléchir à l'ensemble des questions, afin d'acquérir l'ensemble des compétences. Chacun d'entre vous doit repartir avec une trace écrite du travail réalisé (éventuellement sous forme de notes photocopiées ou photographiées), de sorte d'avoir toutes les informations nécessaires sous la main pour la réalisation individuelle.

**CATEGORIES**

| Code_categorie | Libelle_categorie | Montant_cotisation |
|---|---|---|
| AA | Adulte à l'Année | 900,00 € |
| AT | Adulte au Trimestre | 250,00 € |
| EA | Enfant à l'Année | 800,00 € |
| ET | Enfant au Trimestre | 250,00 € |

**STAGES**

| Num_stage | Du | Au | Prix_forfaitaire |
|---|---|---|---|
| 10 | 2013-09-13 | 2013-09-14 | 50,00 € |
| 15 | 2013-12-14 | 2013-12-19 | 300,00 € |

**VILLES**

| CP | Ville |
|---|---|
| 13001 | Marseille |
| 13002 | Marseille |
| 13500 | Martigues |
| 13700 | Marignane |

Tab. B.1 : Base de données (partie 1)

**CHEVAUX**

| Num_C | Nom_C | Sexe_C | Encolure | DN_C | Niv_C |
|---|---|---|---|---|---|
| 01 | Doumé | M | 140 | 2011-05-15 | 3 |
| 02 | Camomille | F | 169 | 2003-02-26 | 3 |
| 11 | Roméo | M | 158 | 2012-04-18 | 1 |
| 13 | Juliette | F | 145 | 2008-11-02 | 2 |

**INSCRIRE**

| Num_adherent | Num_stage | Num_cheval |
|---|---|---|
| 0165 | 10 | 13 |
| 0166 | 15 | 01 |
| 0165 | 15 | 11 |
| 0167 | 10 | 13 |

**ADHERENTS**

| Num_adherent | Nom_adherent | Prenom | Sexe | CP | Date_nais | Echeance_cot | Niveau | Cat |
|---|---|---|---|---|---|---|---|---|
| 0165 | MARTIN | Claude | M | 13500 | 1967-07-13 | 2014-05-29 | 04 | AA |
| 0166 | DUPOND | Louise | F | 13002 | 1974-11-25 | 2014-08-29 | 03 | AT |
| 0167 | FAVRE | Olivier | M | 13001 | 2000-06-06 | 2013-09-30 | 01 | ET |
| 0168 | FAVRE | Julie | F | 13001 | 2000-06-06 | 2013-09-30 | 01 | ET |

Tab. B.2 : Base de données (partie 2)

Avant de répondre aux questions suivantes, vous devez IMPÉRATIVEMENT lire le chapitre 2.

1. Comprendre la base de données Galop d'Azur proposée.
2. Rechercher la définition d'une clé primaire.
3. Pour chacune des tables décrites, indiquer quelle pourrait être la clé primaire. A-t-on le choix ?
4. Nous souhaitons ajouter une nouvelle ligne à la table **INSCRIRE** :

| Num_adherent | Num_stage | Num_cheval |
|---:|---:|---:|
| 0168 | 12 | 14 |

   Que faut-il faire avant de pouvoir ajouter cette ligne ?
5. Nous souhaitons ajouter une colonne à la table **INSCRIRE** : la colonne *Nom_cheval*. Est-ce possible ? Est-ce souhaitable ? Pourquoi ?
6. Rechercher la définition d'une clé étrangère. Identifier les clés étrangères existantes.

### B.2.2 Réalisation

Prise en main de *PostgreSQL* :

1. Lire le chapitre 1 (et particulièrement la partie 1.6).
2. Ouvrir un éditeur de texte pour écrire le script de création de la base de données.
3. Ouvrir un terminal pour lancer un client *PostgreSQL*.
4. Créer la base de données « Galop d'Azur ».
5. Créer les 6 tables. Ne pas oublier les clés primaires et étrangères.
6. Ajouter les contraintes nécessaires pour vérifier la bonne saisie des données.
7. Saisir le contenu des 6 tables.
8. Visualiser le contenu de chaque table.
9. Rédiger un rapport comportant les réponses aux parties préparation et réalisation à déposer sur Moodle à la fin de la séance.

Annexe C

# Problème 2 : Gestion d'un club hippique (2)

## C.1 Le problème

Reprenez la même base de données, Galop d'Azur, que celle utilisée dans le problème 1. Dans ce problème, nous allons utiliser le langage SQL pour effectuer des requêtes.

## C.2 Travail demandé

### C.2.1 Préparation

Avant de répondre aux questions suivantes, vous devez IMPÉRATIVEMENT lire le chapitre 3.

1. Qu'est-ce que SQL ?
2. Rechercher la définition d'une requête.
3. Préparer les requêtes suivantes *uniquement en SQL* et prévoir la réponse qui va être donnée :
   (a) liste des adhérents (nom et prénom) ;
   (b) liste par ordre alphabétique des adhérents (nom et prénom) ;
   (c) liste par ordre alphabétique des adhérents (n° et prénom) ;
   (d) liste des adhérentes (nom et prénom) ;
   (e) liste des adhérentes de la plus jeune à la plus âgée (nom, prénom) ;

(f) liste des cavaliers de niveaux 3 OU 4 ;

(g) liste des cavaliers de niveau 1 ET de sexe masculin ;

(h) liste des adhérents habitant Marseille (Nom, prénom, ville) ;

(i) liste des inscrits (nom, prénom) au stage 10 ainsi que le nom du cheval qui leur est attribué ;

(j) liste des adhérents dont le nom commence par F. ;

(k) liste des adhérents adultes ;

(l) liste des villes dont sont originaires les adhérents.
À quelle difficulté sommes nous confrontés ? Comment contourner cette difficulté ?

(m) Enregistrement d'un nouvel adhérent :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| FERAUD | Chantal | F | 13730 | 1996-06-03 | 2013-10-29 | 1 | EA |

À quelle difficulté allez-vous vous heurter ?

(n) Modification de l'adhérente Louise DUPOND qui vient de payer sa cotisation jusqu'au 2013-11-29 ;

(o) Suppression de l'adhérent MARTIN Claude qui a arrêté le cheval suite à une chute ;

(p) Suppression de tous les adhérents qui n'ont pas leur cotisation à jour ;

(q) Suppression du cheval nommé Juliette ;

(r) Création d'une nouvelle table :

**MONITEURS**

| Num_moniteur | Nom_moniteur | Prenom_moniteur |
|---|---|---|
| 1 | PEREZ | Nicolas |
| 2 | MENOTTI | Louise |

(s) Modification de la table **STAGES** avec les valeurs : stage 10 animé par Louise Menotti et stage 15 animé par Nicolas Perez ;

4. Qu'appelle-t-on langage d'interrogation des données ? Quelles sont les requêtes effectuées dans la question 3 qui en font partie ?

5. Qu'appelle-t-on langage de manipulation des données ? Quelles sont les requêtes effectuées dans la question 3 qui en font partie ?

6. Qu'appelle-t-on langage de définition des données ? Quelles sont les requêtes effectuées dans la question 3 qui en font partie ?

7. Qu'est-ce qu'une jointure ? Dans quel cas est-elle nécessaire ?

### C.2.2 Réalisation

Réaliser individuellement les questions de préparation et vérifier que les réponses apportées correspondent à la réalité.

Déposer sur Moodle un rapport comprenant les réponses aux questions.

Annexe D

# Problème 3 : Clients et factures

## D.1 Le problème

L'informatisation des stocks, clients, factures et commandes est une des applications les plus directes de l'informatique de gestion. Il est toujours important de pouvoir garder un œil sur l'état des stocks, garder une trace des ventes, garder le contact de ses clients, et l'informatique est un outil précieux pour accomplir efficacement toutes ces tâches. L'utilisation de bases de données relationnelles est toute indiquée pour réaliser ce genre d'application. La figure D.1 page suivante présente le modèle relationnel sous forme d'un extrait des tables de la base de données.

## D.2 Travail demandé

### D.2.1 Préparation

1. Analysez l'organisation des données. Proposez un diagramme de classes correspondant à ce modèle relationnel.

2. Il vous sera demandé d'implanter cette base de données dans le SGBD *PostgreSQL* en utilisant le langage SQL de modélisation des données.

3. On souhaite également contrôler la validité d'autres données. Comment s'assurer qu'une LigneFacture ne peut pas présenter de quantité négative ? Que les prix et les stocks ne peuvent pas être négatifs ? Que tous les produits aient un nom différent ? Pour tester les contraintes

**Client**

| idClient | nomClient | emailClient |
|---|---|---|
| 179 | Mario | mario@mushroomy.com |
| 198 | Peach | peach@mushroomy.com |

↑ FK(idClient)

**Facture**

| idFacture | idClient* | date |
|---|---|---|
| 19 879 | 179 | 1993-10-07 |
| 19 766 | 179 | 1994-08-26 |
| 19 789 | 198 | 1993-12-08 |

↑ FK(idFacture)

**LigneFacture**

| idFacture* | idProduit* | quantité |
|---|---|---|
| 19 879 | 3 | 5 |
| 19 879 | 7 | 2 |
| 19 789 | 6 | 3 |
| 19 879 | 2 | 8 |
| 19 766 | 3 | 1 |

↓ FK(idProduit)

**Produit**

| idProduit | nomProduit | prixHT | stock |
|---|---|---|---|
| 3 | Champi rouge | 1,15 | 12 |
| 7 | Champi vert | 3,98 | 18 |
| 2 | Étoile | 1,23 | 30 |
| 6 | Fleur de feu | 6,63 | 67 |

Fig. D.1 : Modèle relationnel de la base de données

d'intégrité, préparez des données supplémentaires susceptibles de violer ces contraintes (deux clients de même identifiant, deux fois le même produit pour la même facture, suppression d'un produit déjà facturé, etc.)

4. Un nouveau client, Luigi, souhaite effectuer une commande de trois fleurs de feu et deux champis rouges. Quelles lignes faut-il ajouter dans les tables ?

5. Luigi souhaite modifier sa commande : il veut seulement deux fleurs de feu, et souhaite remplacer les deux champis rouges par une étoile. Que faut-il faire ? Quelles sont les commandes SQL correspondantes ?

6. Recherchez les informations suivantes dans la base (dressez le tableau répondant à chaque question).

   (a) Liste des factures du client 179,

   (b) Nom du client 179,

   (c) Liste des factures par ordre de date croissant,

   (d) Produits dont le stock est inférieur à 15,

   (e) Liste des produits avec le prix TTC (avec une TVA à 5,5 %),

   (f) Valeur HT du stock de chaque produit,

   (g) Liste des champis (identifiant, intitulé, prix HT, stock),

   (h) Identifiants des produits ayant déjà fait l'objet d'une commande.

   (i) Données pour l'édition papier de la facture 19 879 : Indiquez la liste des produits commandés (numéro, intitulé, quantité, prix unitaire HT, prix total HT et TTC).

   (j) Montant total de chaque facture.

7. Déterminez les commandes SQL correspondant à chacune des requêtes de la question 6. Attention, les deux dernières sont un peu particulières et nécessitent de nouveaux mots-clés SQL. Essayez de bien comprendre comment les informations demandées sont calculées, et recherchez les mots-clés nécessaires dans le cours.

### D.2.2 Réalisation

Implantez la base de données et testez les contraintes d'intégrité. Essayez toutes les commandes SQL préparées en groupe. Vérifiez que le résultat correspond à ce qui était attendu, corrigez-les si nécessaire. *Copiez dans un fichier texte chaque requête correcte !*

### D.2.3 Présentation du travail réalisé

Votre groupe sera évalué via un rapport écrit ou lors d'une présentation orale, d'une durée de 8 minutes.

**Attention :** l'objectif de la présentation n'est pas ici de lister les réponses aux cas d'utilisation, mais d'identifier et d'expliquer les différentes opérations que vous avez utilisées, comment les exprimer en SQL, et d'identifier quelles requêtes nécessitaient chacune de ces opérations.

- Expliquez comment les données sont organisées,
- Expliquez comment s'assurer que des données ambigües ou incorrectes ne peuvent pas se trouver dans la base de données, donnez des exemples de données qui seront refusées par le SGBD.
- Expliquez en quoi consistent les opérations de projection, de restriction, de jointure et d'agrégat. Pour chacune de ces fonctionnalités, listez les requêtes les exploitant.

Annexe E

# Problème 4 : SI d'une agence de voyages

## E.1 Le problème

Une agence de voyages stocke l'ensemble de ses clients et des stations où ceux-ci sont susceptibles de séjourner dans un SGBDR. À chaque fois qu'un client séjourne dans une station, l'information est stockée dans la base. Chaque station propose un certain nombre d'activités. La figure E.1 page suivante montre le diagramme de classes représentant la base de données. On dispose de plus des restrictions suivantes :

1. Le prix d'une activité, le tarif d'une station, le nombre de places réservées et la capacité des stations sont positifs ou nuls ;
2. Un même client ne peut réserver qu'un seul séjour à une date donnée ;
3. Tous les champs doivent obligatoirement être renseignés.

L'application devra couvrir les cas d'utilisation suivants :

1. Lister toutes les stations connues ;
2. Lister les clients du Royaume Champignon ;
3. Lister les stations situées dans la région de Port Lacanaïe ou de l'Ile des Yoshi ;
4. Quelles sont les activités de la station « Gelato-les-Flots » ?
5. Lister tous les séjours du client Mario (en indiquant le nom de la station).

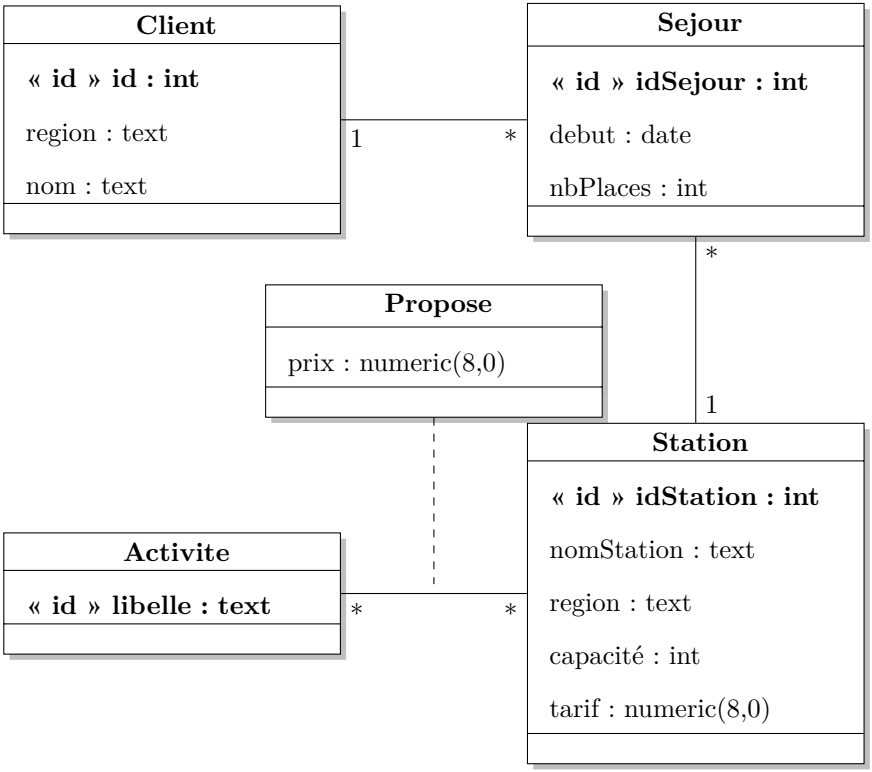



Fig. E.1 : La base de données

6. Lister tous les séjours ayant eu lieu sur la station « Yunnanville » (en indiquant le nom du client).
7. Quels sont les clients ayant séjourné à Yunnanville ?
8. Combien de stations sont situées hors du Royaume Champignon ?
9. Trier les stations en fonction du nombre d'activités proposées.
10. Donner le total dépensé par chaque client en séjours.
11. Quel est le chiffre d'affaire[1] de la société ?
12. Combien chaque station a-t-elle généré de CA ?

## E.2 Travail demandé

Implantez la base de données (avec les contraintes d'intégrité), et les requêtes correspondant aux différents cas d'utilisation. Les informations suivantes seront inscrites dans la base de données :

### Clients

1. Mario, du Royaume Champignon ;
2. Peach, du Royaume Champignon ;
3. Yoshi, de l'Ile des Yoshi.

### Stations

1. Yunnanville, région de Port Lacanaïe, 20 places, 1 200 PO ;
2. Gelato-les-Flots, région de l'Ile Delphino, 100 places, 2 270 PO ;
3. Château de Bowser, région du Royaume Champignon, 5 places, 15 750 PO.

### Séjours

1. Mario à Yunnanville ; 4 places à partir du 3 aout 1998 ;
2. Mario à Yunnanville ; 2 places à partir du 22 juillet 2004 ;
3. Princesse Peach à Gelato-les-Flots ; 2 places à partir du 3 septembre 2007.

[1] Le chiffre d'affaire correspond au total des sommes versées par les clients au titre des séjours.

**Activités**

1. Voile (diponible à Gelato-les-Flots pour 150 PO) ;
2. Plongée (disponible à Gelato-les-Flots pour 120 PO) ;
3. Spectacles (disponible à Yunnanville pour 50 PO) ;
4. Sauna (disponible à Yunnanville pour 20 PO ou au château de Bowser gratuitement).

## E.3 Préparation

Réalisez le modèle relationnel correspondant à la base de données, et préparez les commandes SQL de modélisation des données correspondantes, incluant les contraintes d'intégrité et les données.

Enfin, préparez les commandes SQL d'interrogation des données correspondant aux cas d'utilisation. Pour chaque commande SQL, indiquez le résultat attendu d'après les données fournies. Les requêtes sont de difficulté croissante. Assurez-vous que chaque membre du groupe ait bien compris une requête avant de passer à la suivante. Pour chaque requête, indiquez où se trouve dans le cours les ressources vous ayant permis de l'élaborer (normalement, dans les chapitres 3 et 4). Ce problème ne nécessite pas l'emploi de sous-requêtes ni d'opérations ensemblistes !

## E.4 Réalisation

Implantez la base dans *PostgreSQL*, testez les requêtes en SQL. Pensez à ajouter des données pour tester les cas limites (requêtes ne renvoyant aucune information, doublons, etc.). Une fois fonctionnelles, recopiez les au propre en prévision du rapport écrit ou orales.

## E.5 Mise en commun du travail et présentation du travail réalisé

Travaillez par deux pour comparer les requêtes SQL de chacun, puis réalisez une présentation écrite ou orale d'une durée de 8 minutes.

Annexe F

# Problème 5 : Réseau social et vie privée

## F.1 Le problème

Le réseau social *Google+* a mis en place un système de « cercles » de contacts afin de permettre un contrôle fin des contacts ayant la possibilité de lire tel ou tel message. Ainsi, il est possible d'envoyer des messages différents à ses amis et à ses collègues de travail.

Le principe est le suivant :

- Un utilisateur a la possibilité de créer plusieurs « cercles » ;
- Un cercle rassemble un nombre quelconque d'utilisateurs ;
- Un utilisateur peut poster un message à l'intention de l'un ou plusieurs de ses cercles ;
- Un utilisateur peut commenter les messages qu'il reçoit à plusieurs reprises.

Ce modèle peut être représenté par le diagramme de classes représenté sur la figure F.1 page suivante.

## F.2 Travail demandé

### F.2.1 Préparation

Dessinez le modèle relationnel correspondant au diagramme de classes de la figure F.1. Précisez les contraintes d'intégrité et préparez un jeu de test pour contrôler la correction de ces contraintes.

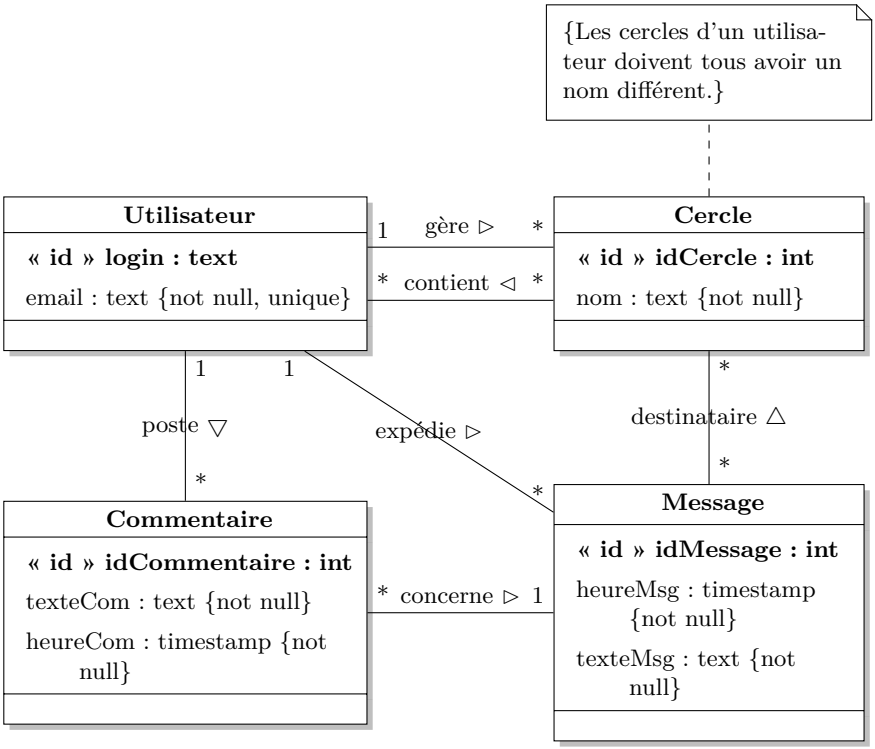



Fig. F.1 : Diagramme de classes du réseau social

Pour chacune des questions suivantes, préparez un jeu de test et la requête SQL permettant de réaliser le cas d'utilisation demandé. Pensez à tester des cas particuliers (lister les messages reçus alors qu'aucun n'a été envoyé, ou alors que des messages proviennent de plusieurs cercles, etc.)

1. Inscription d'un utilisateur ;
2. Création d'un cercle par un utilisateur ;
3. Ajout d'un contact à un cercle ;
4. Envoi d'un message à un cercle ;
5. Envoi d'un commentaire à un message ;
6. Modification d'un message ;
7. Suppression d'un message ;
8. Modification d'un commentaire ;
9. Liste des cercles d'un utilisateur (ordre alphabétique) ;
10. Liste des contacts d'un cercle (ordre alphabétique) ;
11. Liste des contacts d'un utilisateur (ordre alphabétique) ;
12. Liste des messages reçus (ordre chronologique) ;
13. Liste des commentaires à un message (ordre chronologique).

### F.2.2 Réalisation

Créez la base de données à partir du modèle relationnel, en essayant de ne pas se référer au diagramme de classes. N'oubliez pas d'implanter toutes les contraintes (clés primaires, clés étrangères et autres). Le modèle relationnel est-il suffisant pour implanter la bases de données ? Que manque-t-il ?

Testez les requêtes SQL. Relevez les requêtes SQL incorrectes, que leur manquait-il ?

### F.2.3 Mise en commun et présentation du travail réalisé

Contrôlez que le modèle relationnel et les requêtes préparées en groupe étaient corrects.

La présentation devra comporter le modèle relationnel, les requêtes de création de table (une seule en cas de présentation orale), les requêtes de manipulation des données, ainsi que les jeux de test exemples d'utilisation.

Annexe G

# Problème 6 : SI d'une société d'assurances

## G.1 Le problème

Une société d'assurances spécialisée dans l'automobile vous demande de repenser son système d'information :

> *« Nous souhaitons suivre les informations liées à nos assurés, leurs contrats et leurs sinistres. Nos assurés sont identifiés par un numéro unique, leur nom et leur adresse (rue, code postal, ville). Ils peuvent souscrire plusieurs contrats d'assurance auprès de notre société.*
>
> *Un contrat est identifié par un numéro de police unique. Il comporte une date de souscription, et diverses informations au sujet du véhicule à assurer : marque, modèle, numéro de carte grise, date de mise en circulation, numéro d'immatriculation et puissance fiscale. Chaque contrat bénéficie d'un bonus, qui évolue régulièrement. On souhaite conserver l'historique des bonus d'un client.*
>
> *Un contrat offre un ou plusieurs types de garantie prédéfinis, chacun avec une franchise et un plafond.*
>
> *Quand un assuré déclare un sinistre, on monte un dossier, identifié par son numéro. Il décrit la date du sinistre, le lieu, sa nature et les circonstances de l'accident ; éventuellement la présence de blessés. Chaque sinistre est rattaché à un contrat d'assurance, et peut mettre en jeu plusieurs types de garantie. Pour chaque type de garantie, on enregistre le montant à rembourser déclaré par*

*l'assuré, ainsi que le montant effectivement remboursé une fois que celui-ci a été calculé par nos services. Une fois traités, les dossiers sont marqués comme tels.*

*Nous avons identifié les règles de gestion suivantes :*

1. *Le code postal doit comporter exactement 5 chiffres ;*
2. *Tous les montants doivent être positifs ou nuls ;*
3. *Tous les champs (sauf le montant remboursé) doivent être renseignés ;*
4. *Les numéros d'immatriculation des voitures sont uniques,*
5. *La franchise doit être inférieure au plafond ;*
6. *La date de déclaration d'un sinistre doit être postérieure ou égale à la date de survenance ;*
7. *La suppression d'un sinistre entraine la suppression des montants mis en jeu ; la suppression d'un contrat entraine la suppression des informations concernant les types de garantie et l'historique des bonus. On ne doit pas pouvoir supprimer un assuré ayant encore des contrats, un contrat ayant fait l'objet de sinistres, ou un expert ayant suivi des sinistres.*

*À partir de ces données, nous devrons pouvoir obtenir les informations suivantes :*

1. *Rechercher un assuré (numéro, nom, adresse complète) à partir d'une partie de son nom et sa ville ; du numéro de police d'un de ses contrats ou du numéro de dossier d'un de ses sinistres.*
2. *Obtenir la liste les contrats, ainsi que des sinistres d'un assuré. Pour chaque assuré, le nombre de contrats et le nombre de sinistres.*
3. *La liste des assurés ayant fait l'objet d'une expertise par un expert donné.*
4. *Pour chaque sinistre, puis pour chaque assuré, le montant total déclaré et remboursé.*
5. *La liste des contrats souscrits dans l'année, pour lesquels le montant total remboursé est supérieur à 10 000 €.*
6. *Pour chaque contrat, son bonus actuel.*
7. *Les contrats pour lesquels la garantie de code 1 n'a pas été souscrite. »*

Votre chef de projet a réalisé un diagramme de classes pour représenter ce système d'information. Il est représenté sur la figure G.1 page ci-contre.

Il vous est demandé d'implanter cette base de données dans le SGBD *PostgreSQL*, avec ses contraintes d'intégrité, puis de réaliser les requêtes SQL correspondant aux différents cas d'utilisation identifiés.

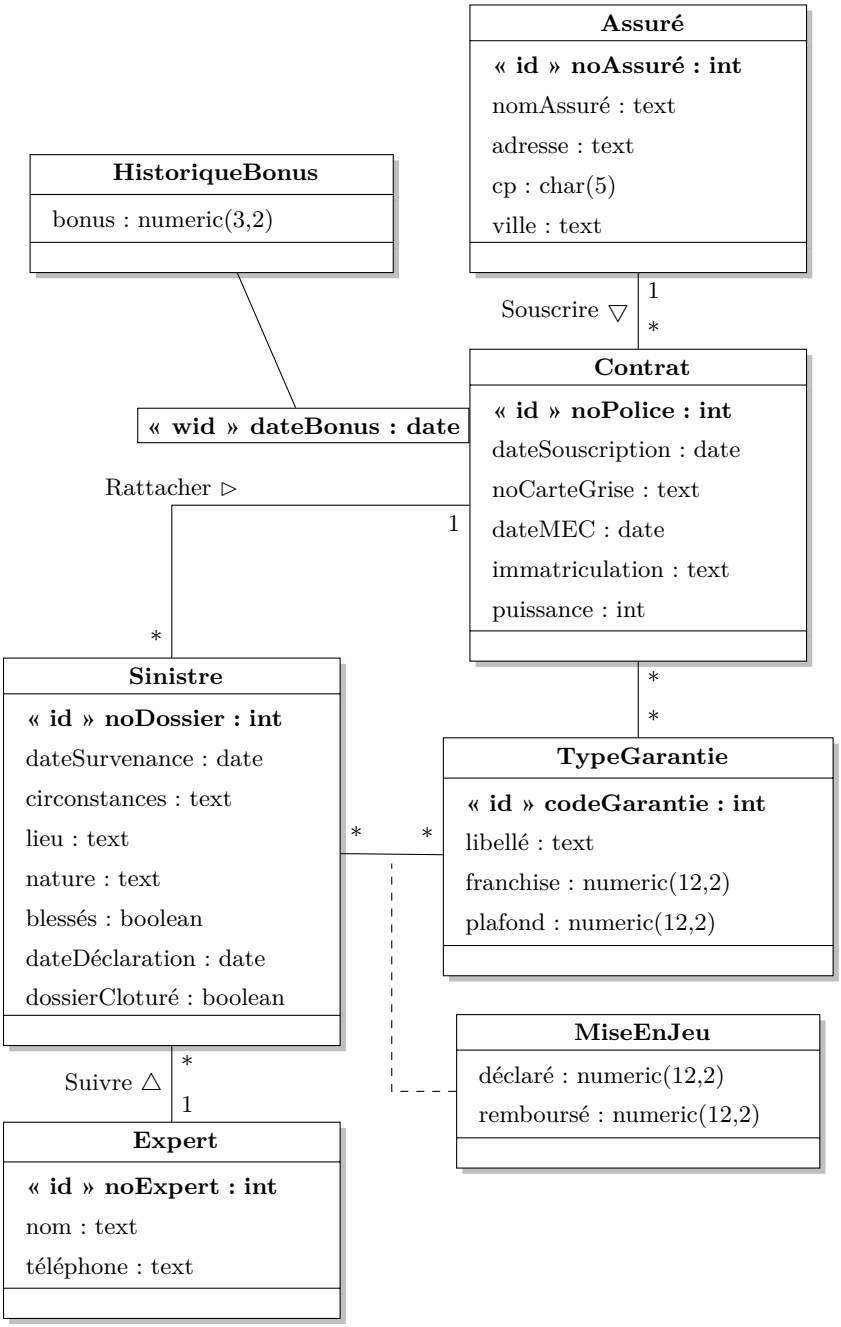



FIG. G.1 : Diagramme de classes du SI

## G.2 Travail demandé

### G.2.1 Préparation

Dans un premier temps, réalisez le modèle relationnel correspondant au diagramme de classes du chef de projet. Cherchez comment implémenter les tables et les contraintes d'intégrité. Préparez des jeux de test pour tester les contraintes d'intégrité.

Ensuite, préparez des jeux de test qui vous permettront de tester les cas d'utilisation demandé par le client. Dans chaque cas, prévoyez de tester le cas nominal et les cas exceptionnels. Par exemple pour calculer le nombre ou le montant des sinistres de chaque assuré, prévoyez des assurés avec 0, 1 ou plusieurs sinistres, contrats, etc.

Quand le jeu de test d'une requête est prêt, vous pouvez préparer la requête elle-même (certains cas ne peuvent pas être réalisés en une seule requête simple, identifiez-les). Si vous manquez de temps, notez simplement les tables mises en jeu et les opérations à réaliser (notamment les restrictions, projections, jointures ou agrégats).

### G.2.2 Réalisation

Implantez le modèle relationnel et les données correspondant aux jeux de test préparés. Testez les requêtes.

### G.2.3 Mise en commun et présentation du travail réalisé

Corrigez le cas échéant le modèle relationnel, les jeux de test et les requêtes préparées pour le compte-rendu.

Celui-ci devra comporter le modèle relationnel, les requêtes de création de table (une seule en cas de présentation orale), et les requêtes avec leurs jeux de test (3 ou 4 seulement en cas de présentation orale).

Annexe H

# Problème 7 : Ligue de football

Vous devez reprendre une base de données existante sur la gestion des effectifs, des matchs et des cartons d'une ligue de football. La base de données est fournie sur *Moodle* au format *LibreOffice Calc*, un aperçu est donné ci-avant.

## H.1 Analyse de l'existant

On souhaite tout d'abord extraire les statistiques suivantes (préparez les requêtes SQL correspondantes) :

1. Nombre de cartons par joueur sur la saison,
2. Nombre de cartons par équipe,
3. Nombre de victoires à domicile pour chaque équipe.

À partir de la structure des tableaux H.1 à H.3 page suivante, reconstituez le modèle relationnel (en retrouvant notamment les clés primaires et étrangères), puis le diagramme de classes correspondant à cette base. Quelle(s) critique(s) pourriez-vous faire à propos de cette base de données ?

## H.2 Extension de la base

Voici les besoins identifiés :

> *« Dans l'avenir, nous envisageons de maintenir des informations spécifiques à chaque équipe (par ex., son budget, le nombre d'abonnements, son chiffre d'affaire). L'ajout de ces informations est inutile à ce stade, mais nous souhaitons nous assurer*

| nom | prenom | equipe |
|---|---|---|
| Franccois | Julien | Metz |
| Le Roux | Christophe | Rennes |
| Gonzales | Arnaud | Auxerre |
| Tassali | Kamal | Sochaux |
| Maoulida | Toifilou | Montpellier |
| Sergio | Paulo | Montpellier |
| Trapasso | Marcello | Sochaux |
| Liebus | Johan | Metz |
| Tuzzio | Eduardo | Marseille |
| Rodriguez | Bruno | Lens |
| Nalis | Lilian | Bastia |
| Moldovan | Viorel | Nantes |
| Basto | Bruno | Bordeaux |
| Reyes | Pedro | Auxerre |
| | ... | |

TAB. H.1 : Extrait de la table *Effectif*

| local | visit | butLocal | butVisit |
|---|---|---|---|
| Nantes | Guingamp | 2 | 0 |
| Bordeaux | Rennes | 2 | 1 |
| Bastia | Guingamp | 3 | 0 |
| Marseille | Troyes | 0 | 1 |
| Montpellier | Guingamp | 2 | 1 |
| Lorient | Nantes | 1 | 2 |
| Auxerre | Montpellier | 1 | 0 |
| Nantes | Sochaux | 0 | 0 |
| Lens | Monaco | 1 | 0 |
| Lyon | Auxerre | 3 | 0 |
| Montpellier | Sochaux | 0 | 0 |
| Sochaux | Lyon | 2 | 1 |
| Sochaux | Lorient | 2 | 2 |
| Bordeaux | Lyon | 0 | 1 |
| | ... | | |

TAB. H.2 : Extrait de la table *Match*

| nom | prenom | local | visit | couleur |
|---|---|---|---|---|
| Laurent | Pierre | Bastia | Bordeaux | Jaune |
| Durand | Eric | Montpellier | Rennes | Jaune |
| Nielsen | Morten | Guingamp | Bastia | Rouge |
| Rool | Cyril | Guingamp | Monaco | Jaune |
| Rodriguez | Julien | Monaco | Sedan | Rouge |
| Capron | Eddy | Sedan | PSG | Rouge |
| Pochettino | Mauricio | PSG | Marseille | Jaune |
| Berson | Mathieu | Rennes | Nantes | Rouge |
| Lachuer | Yann | Nantes | Auxerre | Jaune |
| Lucas | | Sochaux | Rennes | Jaune |
| Fretard | Jerome | Auxerre | Sedan | Jaune |
| van Handenhoven | Gunter | Bordeaux | Metz | Rouge |
| Crucet | Stephane | Sochaux | Lorient | Jaune |
| Guyot | Laurent | Troyes | Guingamp | Jaune |
| | | ... | | |

TAB. H.3 : Extrait de la table *Carton*

> *de la flexibilité de la solution proposée. Il en va de même pour les informations relatives aux joueurs, nous ne pouvons plus nous contenter de lister les effectifs, les matchs et les cartons sur une saison. Aujourd'hui, nous arrivons difficilement à gérer l'ensemble de la première division, cependant nous souhaiterions que cette refonte nous permette de gérer également la seconde division en partageant la même base de données. »*

Réalisez le diagramme de classes correspondant à ces spécifications. Pensez à justifier largement les choix effectués ! Ensuite, préparez le modèle relationnel correspondant.

## H.3 Migration des données

Commencez par importer les données dans *PostgreSQL*. Le plus simple est de commencer par exporter les données depuis *LibreOffice Calc* au format `.csv` (*Comma Separated Values* c'est-à-dire, des fichiers texte dont les colonnes sont séparées par des virgules). Créez dans *PostgreSQL* les tables dont la structure correspond à la base de données d'origine (utilisez le modèle relationnel reconstitué lors de la phase d'analyse). Vous pouvez ensuite importer des données `.csv` dans une table *PostgreSQL* dans *pgAdmin* (menu Outils → Importer…) ou psql avec une commande de type :

```
\copy table from 'fichier.csv' with null as '\0' csv header
```

L'option **null as** '\0' permet de transformer les champs vides en chaines de caractères vides (pour valider les contraintes **NOT NULL**) ; l'option csv header indique que les données sont au format `.csv` et qu'il faut ignorer la première ligne (c'est un en-tête). Notez les difficultés que vous rencontrez lors de l'importation des données. Quel est l'avantage à utiliser un SGBD par rapport à un simple tableau ?

Une fois les données importées, migrez les vers les nouvelles tables. N'oubliez pas que vous pouvez combiner une instruction **INSERT** avec une requête **SELECT** pour copier des données d'une table à l'autre (cf cours). Vous pouvez exploiter les champs de type **serial** ou les *séquences* pour générer d'éventuelles clés numériques.

Pour tester votre nouvelle base, adaptez tout d'abord les requêtes de la section H.1. Ensuite, préparez les requêtes SQL répondant à ces besoins :

1. Nombre de victoires total pour chaque équipe,
2. Nombre de buts marqués à domicile pour chaque équipe,
3. Nombre total de buts encaissés sur la saison.

# Glossaire

**Algèbre Relationnelle** L'algèbre relationnelle est un concept mathématique de la théorie des ensembles qui consiste à définir des opérations sur les relations. Les relations sont au cœur du fonctionnement des BDR.

***Application Programming Interface*** (*API*). « Interface de programmation » en français, mais le terme n'est pas d'usage répandu. Ensemble de fonctions, procédures ou classes mises à disposition des programmes informatiques par une bibliothèque logicielle, un système d'exploitation ou un service. La connaissance des API est indispensable à l'interopérabilité entre les composants logiciels. Dans le cas typique d'une bibliothèque, il s'agit généralement de fonctions considérées comme utiles pour d'autres composants. Pour l'accès aux bases de données, on utilise par exemple les *API ODBC* (*Open Database Connectivity*) ou *JDBC* (*Java Database Connectivity*).

**Base de Données** (BD ou *DB*). Une base de données est un ensemble d'informations reliées entre elles de manière cohérente, et stockées de manière permanente dans un système informatique.

**Base de Données Relationnelle** (BDR ou *RDB*). Une BD basée sur la notion de relations, introduite par E. F. Codd [4]. Plus de 80 % des BD déployées aujourd'hui sont des bases de données relationnelles. Cf aussi Base de Données et **Relation !**.

***Database*** (*DB*). Cf Base de Données.

***Database Management System*** (*DBMS*). Cf Système de Gestion de Bases de Données.

**Déclencheur** Procédure stockée déclenchée automatiquement lors de certaines requêtes (`INSERT`, `DELETE`…)

**Entité/Association** (Modèle E/A, **ER!** *Model*). Modélisation à haut niveau, sous forme de graphes (nœuds et arêtes), du système d'information.

**Langage Déclaratif** Un langage déclaratif est un langage de programmation permettant de décrire le résultat attendu sans contexte ni aucun état interne. En d'autres termes, on décrit le *quoi*, c'est-à-dire le problème, et non pas le *comment*, comme on le ferait avec un langage impératif classique comme le C ou le Java.

**Procédure stockée** Procédure programmée en langage déclaratif (SQL) ou impératif (PL/PgSQL, par exemple), stockée par le SGBD et exécutée directement par celui-ci.

***Relational Database*** (*RDB*). Cf Base de Données Relationnelle.

***Relational Database Management System*** (RDBMS). Cf système de gestion de bases de données relationnelles.

**Système de Gestion de Bases de Données** (SGBD ou *DBMS*). Serveur (logiciel) ou API permettant la gestion des bases de données (administration, stockage, interrogation, mise à jour, etc.) Un Système de Gestion de Bases de Données Relationnelles (SGBDR, *RDBMS*) est un SGBD basé sur le modèle relationnel.

***View*** Cf Vue.

**Vue** (*View*). Table virtuelle, mise à jour en temps réel, obtenue à partir des tables « réelles » de la BDD. Certains SGBD permettent la mise à jour des données d'une vue et la répercutent sur les tables.

# Index


Administration, 8
Agrégats, 38, 66
**ALL**, 68
**ALTER TABLE**, 48
Anomalie de conception, 71
**ANY**, 67
Association, 74
Association optionnelle, 81

Base (créer), 10
Base de Données, 2
Base de Données Relationnelle, 2

Cardinalités, 75
**CHECK**, 27, 46
Classe, 74
Classe faible, 78, 86
Clé faible, 86
Clé primaire, 22, 24, 46
Clé étrangère, 25
**CONSTRAINT**, 46
Contrainte de colonne, 24
Contrainte de table, 24
Contraintes, 24
**CREATE TABLE**, 44
**CREATE TYPE**, 44

*Database*, 2
*Database Management System*, 4
**DEFAULT**, 28
**DEFAULT** (**CREATE TABLE**), 46
**DEFAULT** (**INSERT**), 53
**DELETE**, 54
**DISTINCT**, 36, 58
Droits d'accès, 11, 14
**DROP TABLE**, 29, 51
Dépendance fonctionnelle, 72

Entité, 74
Entité/Association, 73
**ENUM**, 44
**EXCEPT**, 60
**EXISTS**, 67
Expressions rationnelles, 65

**FOREIGN KEY**, 25, 47
Formes normales, 72
**FROM**, 58

**GRANT**, 14
**GROUP BY**, 39, 60
Groupements, 38

**HAVING**, 60

Identifiant, 78
Identifiant faible, 78
**ILIKE**, 65
**IN**, 67
**INSERT**, 52
**INTERSECT**, 60
Intégrité référentielle, 8

**JOIN**, 36, 59
**JOIN ON**, 36
**JOIN USING**, 36
Jointure, 36

Jointures externes, 40

Langage déclaratif, 7
**LEFT JOIN**, 40
**LIKE**, 65
**LIMIT**, 61

Modèle relationnel, 21, 81

**NATURAL JOIN**, 36
Normalisation, 72
**NOT NULL**, 27

Objet, 74
**OFFSET**, 61
**ON DELETE**, 47
Opérateurs, 64
**ORDER BY**, 61

**PRIMARY KEY**, 24, 46
Produit cartésien, 36
Projection, 36
psql, 12

**REFERENCES**, 25, 47
*Relational Database*, 2
*Relational Database Management System*, 4
Restriction, 35
**RETURNING**, 52
**REVOKE**, 14

**SELECT**, 56
Sous-requêtes, 67
SQL, 42
Synopsis, 43
Système de Gestion de Bases de Données, 4–8
Système de Gestion de Bases de Données Relationnelles, 4

Type d'entité, 74
Types de données, 43

UML, 73
**UNION**, 60
**UNIQUE**, 27, 46
**UPDATE**, 55
Utilisateur (créer, modifier), 10

Valeurs par défaut, 28
Vue, 6

*Weak Id*, 78, 86
**WHERE**, 35, 59