Base de Données et langage SQL

(IUT, département informatique, 1re année)

Laurent AUDIBERT

Institut Universitaire de Technologie de Villetaneuse – Département Informatique
Avenue Jean-Baptiste Clément
93430 Villetaneuse
Adresse électronique : *laurent[dot]audibert[at]iutv[dot]univ-paris13[dot]fr*

Avant-propos

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

Les bases de données relationnelles constituent l'objet de ce cours. Ces bases sont conçues suivant le modèle relationnel, dont les fondations théoriques sont solides, et manipulées en utilisant l'algèbre relationnelle. Il s'agit, à ce jour, de la méthode la plus courante pour organiser et accéder à des ensembles de données. Nous décrivons le *modèle relationnel*, le passage du modèle entités-associations au modèle relationnel et enfin l'*algèbre relationnelle* dans le chapitre 3.

Le chapitre 4 est entièrement consacré au langage SQL (*Structured Query Language*) qui peut être considéré comme le langage d'accès normalisé aux bases de données relationnelles. Ce langage est supporté par la plupart des systèmes de gestion de bases de données commerciaux (comme *Oracle*) et du domaine libre (comme *PostgreSQL*). Nous détaillons dans ce chapitre les instructions du langage de définition de données et celles du langage de manipulation de données.

Différents exercices de travaux dirigés et de travaux pratiques ponctuent ce cours. Des exemples de corrections de certains des exercices sont regroupés dans la dernière partie du document (chapitre 5).

Ce document constitue le support du cours « Base de Données et langage SQL » dispensé aux étudiants du département d'informatique de l'institut universitaire de technologie de Villetaneuse en semestre décalé. Ce support a été réalisé en utilisant les ouvrages cités en bibliographie.

Vous trouverez ce document en ligne (pour avoir la dernière version par exemple) à l'adresse suivante :

`http://www-lipn.univ-paris13.fr/~audibert/pages/enseignement/cours.htm`

Table des matières

Chapitre 1

Introduction aux bases de données

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

1.1.1 Notion de base de données

Description générale

Il est difficile de donner une définition exacte de la notion de base de données. Une définition très générale pourrait être :

Définition 1.1 -Base de données- *Un ensemble organisé d'informations avec un objectif commun.*

Peu importe le support utilisé pour rassembler et stocker les données (papier, fichiers, etc.), dès lors que des données sont rassemblées et stockées d'une manière organisée dans un but spécifique, on parle de base de données.

Plus précisément, on appelle base de données un ensemble structuré et organisé permettant le stockage de grandes quantités d'informations afin d'en faciliter l'exploitation (ajout, mise à jour, recherche de données). Bien entendu, dans le cadre de ce cours, nous nous intéressons aux bases de données informatisées.

Base de données informatisée

Définition 1.2 -Base de données informatisée- *Une base de données informatisée est un ensemble structuré de données enregistrées sur des supports accessibles par l'ordinateur, représentant des informations du monde réel et pouvant être interrogées et mises à jour par une communauté d'utilisateurs.*

Le résultat de la conception d'une base de données informatisée est une description des données. Par description on entend définir les propriétés d'ensembles d'objets modélisés dans la base de données et non pas d'objets particuliers. Les objets particuliers sont créés par des programmes d'applications ou des langages de manipulation lors des insertions et des mises à jour des données.

Cette description des données est réalisée en utilisant un modèle de données[1]. Ce dernier est un outil formel utilisé pour comprendre l'organisation logique des données.

La gestion et l'accès à une base de données sont assurés par un ensemble de programmes qui constituent le *Système de gestion de base de données* (SGBD). Nous y reviendrons dans la section 1.2. Un SGBD est caractérisé par le modèle de description des données qu'il supporte (hiérarchique, réseau, relationnel, objet : cf. section 1.1.2). Les données sont décrites sous la forme de ce modèle, grâce à un Langage de Description des Données (LDD). Cette description est appelée *schéma*.

Une fois la base de données spécifiée, on peut y insérer des données, les récupérer, les modifier et les détruire. C'est ce qu'on appelle manipuler les données. Les données peuvent être manipulées non seulement par un Langage spécifique de Manipulation des Données (LMD) mais aussi par des langages de programmation classiques.

[1] cf. section 1.1.2 pour une présentation générale de plusieurs modèles de données. Le modèle entités-associations est présenté dans la section 2 et le modèle relationnel dans la section 3.1

Enjeux

Les bases de données ont pris une place importante en informatique, et particulièrement dans le domaine de la gestion. L'étude des bases de données à conduit au développement de concepts, méthodes et algorithmes spécifiques, notamment pour gérer les données en mémoire secondaire (*i.e.* disques durs)[2]. En effet, dès l'origine de la discipline, les informaticiens ont observé que la taille de la RAM ne permettait pas de charger l'ensemble d'une base de données en mémoire. Cette hypothèse est toujours vérifiée car le volume des données ne cesse de s'accroître sous la poussée des nouvelles technologies du WEB.

Ainsi, les bases de données de demain devront être capables de gérer plusieurs dizaines de téra-octets de données, géographiquement distribuées à l'échelle d'Internet, par plusieurs dizaines de milliers d'utilisateurs dans un contexte d'exploitation changeant (on ne sait pas très bien maîtriser ou prédire les débits de communication entre sites) voire sur des nœuds volatiles. En physique des hautes énergies, on prédit qu'une seule expérience produira de l'ordre du péta-octets de données par an.

Comme il est peu probable de disposer d'une technologie de disque permettant de stocker sur un unique disque cette quantité d'informations, les bases de données se sont orientées vers des architectures distribuées ce qui permet, par exemple, d'exécuter potentiellement plusieurs instructions d'entrée/sortie en même temps sur des disques différents et donc de diviser le temps total d'exécution par un ordre de grandeur.

1.1.2 Modèle de base de données

Modèle hiérarchique

Une base de données hiérarchique est une forme de système de gestion de base de données qui lie des enregistrements dans une structure arborescente de façon à ce que chaque enregistrement n'ait qu'un seul possesseur (par exemple, une paire de chaussures n'appartient qu'à une seule personne).

Les structures de données hiérarchiques ont été largement utilisées dans les premiers systèmes de gestion de bases de données conçus pour la gestion des données du programme Apollo de la NASA. Cependant, à cause de leurs limitations internes, elles ne peuvent pas souvent être utilisées pour décrire des structures existantes dans le monde réel.

Les liens hiérarchiques entre les différents types de données peuvent rendre très simple la réponse à certaines questions, mais très difficile la réponse à d'autres formes de questions. Si le principe de relation « 1 vers N » n'est pas respecté (par exemple, un malade peut avoir plusieurs médecins et un médecin a, *a priori*, plusieurs patients), alors la hiérarchie se transforme en un réseau.

Modèle réseau

Le modèle réseau est en mesure de lever de nombreuses difficultés du modèle hiérarchique grâce à la possibilité d'établir des liaisons de type *n-n*, les liens entre objets pouvant exister sans restriction. Pour retrouver une donnée dans une telle modélisation, il faut connaître le chemin d'accès (les liens) ce qui rend les programmes dépendants de la structure de données

Ce modèle de bases de données a été inventé par C.W. Bachman. Pour son modèle, il reçut en 1973 le prix Turing.

Modèle relationnel

Une base de données relationnelle est une base de données structurée suivant les principes de l'algèbre relationnelle.

Le père des bases de données relationnelles est Edgar Frank Codd. Chercheur chez IBM à la fin des année 1960, il étudiait alors de nouvelles méthodes pour gérer de grandes quantités de données car les modèles et les logiciels de l'époque ne le satisfaisait pas. Mathématicien de formation, il était persuadé qu'il pourrait utiliser des branches spécifiques des mathématiques (la théorie des ensembles

[2] Il faut savoir que les temps d'accès à des disques durs sont d'un ordre de grandeur supérieur (disons 1000 fois supérieur) aux temps d'accès à la mémoire RAM. Tout gestionnaire de base de données doit donc traiter de manière particulière les accès aux disques.

et la logique des prédicats du premier ordre) pour résoudre des difficultés telles que la redondance des données, l'intégrité des données ou l'indépendance de la structure de la base de données avec sa mise en œuvre physique.

En 1970, Codd (1970) publia un article où il proposait de stocker des données hétérogènes dans des tables, permettant d'établir des relations entre elles. De nos jours, ce modèle est extrêmement répandu, mais en 1970, cette idée était considérée comme une curiosité intellectuelle. On doutait que les tables puissent être jamais gérées de manière efficace par un ordinateur.

Ce scepticisme n'a cependant pas empêché Codd de poursuivre ses recherches. Un premier prototype de Système de gestion de bases de données relationnelles (SGBDR) a été construit dans les laboratoires d'IBM. Depuis les années 80, cette technologie a mûri et a été adoptée par l'industrie. En 1987, le langage SQL, qui étend l'algèbre relationnelle, a été standardisé.

C'est dans ce type de modèle que se situe ce cours de base de données.

Modèle objet

La notion de *bases de données objet* ou *relationnel-objet* est plus récente et encore en phase de recherche et de développement. Elle sera très probablement ajoutée au modèle relationnel.

1.2 Système de gestion de base de données (SGBD)

1.2.1 Principes de fonctionnement

La gestion et l'accès à une base de données sont assurés par un ensemble de programmes qui constituent le Système de gestion de base de données (SGBD). Un SGBD doit permettre l'ajout, la modification et la recherche de données. Un système de gestion de bases de données héberge généralement plusieurs bases de données, qui sont destinées à des logiciels ou des thématiques différents.

Actuellement, la plupart des SGBD fonctionnent selon un mode client/serveur. Le serveur (sous entendu la machine qui stocke les données) reçoit des requêtes de plusieurs clients et ceci de manière concurrente. Le serveur analyse la requête, la traite et retourne le résultat au client. Le modèle client/serveur est assez souvent implémenté au moyen de l'interface des sockets (voir le cours de réseau) ; le réseau étant Internet.

Une variante de ce modèle est le modèle ASP (Application Service Provider). Dans ce modèle, le client s'adresse à un mandataire (broker) qui le met en relation avec un SGBD capable de résoudre la requête. La requête est ensuite directement envoyée au SGBD sélectionné qui résout et retourne le résultat directement au client.

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

1.2.2 Objectifs

Des objectifs principaux ont été fixés aux SGBD dès l'origine de ceux-ci et ce, afin de résoudre les problèmes causés par la démarche classique. Ces objectifs sont les suivants :

Indépendance physique : La façon dont les données sont définies doit être indépendante des structures de stockage utilisées.

Indépendance logique : Un même ensemble de données peut être vu différemment par des utilisateurs différents. Toutes ces visions personnelles des données doivent être intégrées dans une vision globale.

Accès aux données : L'accès aux données se fait par l'intermédiaire d'un Langage de Manipulation de Données (LMD). Il est crucial que ce langage permette d'obtenir des réponses aux requêtes en un

temps « raisonnable ». Le LMD doit donc être optimisé, minimiser le nombre d'accès disques, et tout cela de façon totalement transparente pour l'utilisateur.

Administration centralisée des données (intégration) : Toutes les données doivent être centralisées dans un réservoir unique commun à toutes les applications. En effet, des visions différentes des données (entre autres) se résolvent plus facilement si les données sont administrées de façon centralisée.

Non redondance des données : Afin d'éviter les problèmes lors des mises à jour, chaque donnée ne doit être présente qu'une seule fois dans la base.

Cohérence des données : Les données sont soumises à un certain nombre de contraintes d'intégrité qui définissent un état cohérent de la base. Elles doivent pouvoir être exprimées simplement et vérifiées automatiquement à chaque insertion, modification ou suppression des données. Les contraintes d'intégrité sont décrites dans le Langage de Description de Données (LDD).

Partage des données : Il s'agit de permettre à plusieurs utilisateurs d'accéder aux mêmes données au même moment de manière transparente. Si ce problème est simple à résoudre quand il s'agit uniquement d'interrogations, cela ne l'est plus quand il s'agit de modifications dans un contexte multi-utilisateurs car il faut : permettre à deux (ou plus) utilisateurs de modifier la même donnée « en même temps » et assurer un résultat d'interrogation cohérent pour un utilisateur consultant une table pendant qu'un autre la modifie.

Sécurité des données : Les données doivent pouvoir être protégées contre les accès non autorisés. Pour cela, il faut pouvoir associer à chaque utilisateur des droits d'accès aux données.

Résistance aux pannes : Que se passe-t-il si une panne survient au milieu d'une modification, si certains fichiers contenant les données deviennent illisibles ? Il faut pouvoir récupérer une base dans un état « sain ». Ainsi, après une panne intervenant au milieu d'une modification deux solutions sont possibles : soit récupérer les données dans l'état dans lequel elles étaient avant la modification, soit terminer l'opération interrompue.

1.2.3 Niveaux de description des données ANSI/SPARC

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

Le niveau externe correspond à la perception de tout ou partie de la base par un groupe donné d'utilisateurs, indépendamment des autres. On appelle cette description le *schéma externe* ou *vue*. Il peut exister plusieurs schémas externes représentant différentes vues sur la base de données avec des possibilités de recouvrement. Le niveau externe assure l'analyse et l'interprétation des requêtes en primitives de plus bas niveau et se charge également de convertir éventuellement les données brutes, issues de la réponse à la requête, dans un format souhaité par l'utilisateur.

Le niveau conceptuel décrit la structure de toutes les données de la base, leurs propriétés (*i.e.* les relations qui existent entre elles : leur sémantique inhérente), sans se soucier de l'implémentation physique ni de la façon dont chaque groupe de travail voudra s'en servir. Dans le cas des SGBD relationnels, il s'agit d'une vision tabulaire où la sémantique de l'information est exprimée en utilisant les concepts de relation, attributs et de contraintes d'intégrité. On appelle cette description le *schéma conceptuel*.

Le niveau interne ou physique s'appuie sur un système de gestion de fichiers pour définir la politique de stockage ainsi que le placement des données. Le niveau physique est donc responsable du choix de l'organisation physique des fichiers ainsi que de l'utilisation de telle ou telle méthode d'accès en fonction de la requête. On appelle cette description le *schéma interne*.

1.2.4 Quelques SGBD connus et utilisés

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

PostgreSQL : `http ://www.postgresql.org/` – dans le domaine public ;

MySQL : `http ://www.mysql.org/` – dans le domaine public ;
Oracle : `http ://www.oracle.com/` – de Oracle Corporation ;
IBM DB2 : `http ://www-306.ibm.com/software/data/db2/`
Microsoft SQL : `http ://www.microsoft.com/sql/`
Sybase : `http ://www.sybase.com/linux`
Informix : `http ://www-306.ibm.com/software/data/informix/`

1.3 Travaux Dirigés – Sensibilisation à la problématique des bases de données

1.3.1 Introduction

Objectifs

L'objectif de ce TD est de se faire une idée de l'intérêt de toute la théorie sur la conception des bases de données et de l'intérêt de l'utilisation des systèmes de gestion de base de données. En d'autres termes, nous allons essayer d'apporter des éléments de réponse à la question :

> *« Pourquoi dois-je m'embêter avec toute cette théorie et ces connaissances à assimiler alors que je sais très bien manipuler un fichier, y stocker des informations et les y retrouver avec mon langage de programmation favoris ? »*

Contexte

Supposons que vous ayez à développer une application de gestion d'une bibliothèque. Tous les livres de la bibliothèque possèdent un numéro de livre, un titre, un ou plusieurs auteurs et un éditeur. Lorsqu'une personne emprunte un livre, il faut mémoriser son nom, son prénom, son numéro de téléphone, son adresse, la date de l'emprunt et la date de retour une fois ce dernier réalisé. Toutes les informations doivent être conservées pour garder un historique des emprunts.

1.3.2 Approche naïve

Une solution simple et naïve . . .

Certains d'entre vous ont une expérience des bases de données (il s'agit vraiment de quelque chose d'incontournable aujourd'hui) ou une expérience importante en développement logiciel. Dans le cadre de cet exercice, oubliez toutes vos connaissances et vos réflexions sur le sujet.

1. Votre application va devoir stocker toutes les informations mentionnées dans l'introduction (section *Contexte*), et de manière persistante, donc en utilisant un fichier. Quelle est la solution de stockage des données la plus naïve et la plus naturelle venant immédiatement à l'esprit ?

. . . mais pas sans conséquences

Supposons que nous adoptions la solution naïve et naturelle suivante :
- Nous créons un fichier texte comportant à l'origine une ligne par livre.
- Dans chaque ligne, on trouve les informations *titre*, `auteur`, `éditeur`, *numéro du livre* séparées par une tabulation.
- Quand une personne emprunte un livre, on complète la ligne du livre en question par les champs *nom*, *prénom*, *téléphone*, *adresse* et *date-emprunt* toujours en séparant ces informations par une tabulation.
- Lorsqu'une personne retourne un livre, il suffit d'ajouter un dernier champs *date-retour* sur la ligne du livre en question.
- Quand un livre est emprunté une nouvelle fois, on crée une nouvelle ligne avec toutes les informations concernant le livre et la personne qui l'emprunte. Bien entendu, le bibliothécaire ne ressaisit pas tout, l'application va chercher la plupart de ces informations dans le fichier.

En fait, on peut voir ce fichier texte comme un tableau de chaînes de caractères dont l'entête des colonnes seraient les suivantes :

Titre	Auteur	Éditeur	N°Livre	Nom	Prénom	Téléphone	Adresse	Date-emprunt	Date-retour

Supposons que l'application de gestion de bibliothèque fonctionne correctement et stocke toutes ses données dans un fichier comme celui que nous venons de décrire. Nous allons nous pencher sur les inconvénients et les conséquences inhérentes à une telle approche.

L'application fonctionne maintenant depuis 10 ans. Le nombre de personnes inscrites à la bibliothèque est relativement constant (bien que l'on constate un roulement) et de 5000 personnes en moyenne par an. Un abonné emprunte en moyenne 5 livres par mois.

2. Quel est, approximativement, le nombre de lignes du fichier des données ?
3. Quelle est la taille approximative du fichier sachant que chaque caractère occupe 1 octet et qu'une ligne contient, en moyenne, 150 caractères ?
4. Supposons qu'une personne est abonnée depuis l'origine de l'application. Elle prévient le bibliothécaire que son prénom est mal orthographié. Combien de lignes, approximativement, doivent être modifiées pour corriger cette erreur dans tout le fichier de données ?
5. Lorsqu'un abonné emprunte un livre, le bibliothécaire saisit simplement le numéro du livre et le nom et le prénom de l'abonné. L'application se charge alors de parcourir le fichier pour rechercher les informations manquantes concernant le livre et l'abonné afin d'écrire, à la fin du fichier, la nouvelle ligne concernant l'emprunt. Dans le pire des cas, l'application doit parcourir tout le fichier. Supposons qu'un accès au fichier coûte $10ms$, qu'une lecture de ligne coûte $6ms$ et qu'une recherche sur la ligne pour trouver le numéro du livre ou le nom et le prénom de l'abonné coûte $1ms$. Quel est, dans le pire des cas, le temps mis par l'application pour compléter les informations saisies par le bibliothécaire ?
6. Énumérez ou résumez tous les problèmes que la représentation des données choisie (le fichier de données) semble poser.

1.3.3 Affinement de la solution

Il est évident que la solution naïve décrite dans la section précédente pose de nombreux problèmes. Elle est totalement inacceptable pour une application sérieuse bien qu'elle soit encore largement employée dans des cas de petite taille (comme par exemple, dans la plupart des fichiers bibliographiques LaTeX).

Un premier affinage de la solution de la section précédente consiste à utiliser non pas un fichier unique mais quatre fichiers distincts :

- Un premier fichier est dédié au stockage des informations concernant les livres de la bibliothèque.
- Un second fichier est dédié au stockage des informations concernant les abonnés.
- Les informations stockées dans le troisème fichier vont permettre de faire la correspondance entre les deux premiers pour signifier qu'un livre donné est en cours de prêt par un abonné donné depuis une date donnée.
- Enfin, un dernier fichier va permettre de stocker l'historique des prêts. Il est similaire au troisième fichier, mais il comporte en plus une information relative à la date de retour du livre.

7. Précisez le format et les informations stockées dans chacun de ces quatre fichiers.
8. Quels sont les avantages de cette nouvelle solution ?
9. Intéressons-nous au premier fichier (celui concernant les livres). Quels problèmes diagnostiquez-vous dans ce fichier ?
10. Le format de ce fichier permet-il de prendre en compte des livres co-écrits par plusieurs auteurs ?
11. Quelle solution proposez-vous ?

1.3.4 Que retenir de ce TD ?

Les problèmes les plus courants rencontrés dans des bases de données mal conçues peuvent être regroupés selon les critères suivants :

Redondance des données – Certains choix de conception entraînent une répétition des données lors de leur insertion dans la base. Cette redondance est souvent la cause d'anomalies provenant de la complexité des insertions.

C'est, par exemple, le cas de la première organisation proposée : dès qu'un abonné emprunte un livre, il faut dupliquer toutes les information concernant l'abonné et le livre emprunté ! Au contraire, dans la deuxième solution, seuls les numéros indispensables à la distinction d'un livre et d'un abonné sont répétés dans le cas d'un emprunt.

Incohérence en modification – La redondance de l'information entraîne également des risques en cas de modification d'une donnée car on oublie fréquemment de modifier toutes ses occurrences.

Anomalie d'insertion – Une mauvaise conception peut parfois empêcher l'insertion d'une information, faute de connaître la valeur de tous ses champs. Pour remédier à ce problème, certains SGBD introduisent une valeur non typée qui signifie que la valeur d'un attribut est inconnue ou indéterminée. Cette valeur (appelée usuellement NULL) indique réellement une valeur inconnue et non une chaîne de caractères vide ou un entier égal à zéro.

Dans la première solution proposée, insérer un nouvel abonné qui n'a jamais emprunté de livre peut poser des problèmes. Une solution serait d'insérer des champs vides (suite de tabulations consécutives) au début de la ligne.

Anomalie de suppression – Enfin, une mauvaise conception peut entraîner, lors de la suppression d'une information, la suppression d'autres informations, sémantiquement distinctes, mais indissociables dans la modélisation adoptée.

Par exemple, dans la première solution proposée, si l'on désire supprimer toutes les traces d'un livre dans le fichier de données, on fera complètement disparaître tous les abonnés qui n'ont emprunté que ce livre.

Bien d'autres enjeux, que ceux que nous avons abordés, sont inhérents aux bases de données. Ces enjeux ont été survolés dans la section 1.2.2 et concernent la gestion des bases de données : indépendance physique, indépendance logique, accès aux données, administration centralisée des données, cohérence des données, partage des données, sécurité des données, résistance aux pannes, etc.

La conception des bases de données est donc un problème complexe. La gestion de ces bases constitue également un problème complexe. Or, ces deux problèmes sont extrêmement récurrents puisque les bases de données se trouvent aujourd'hui au cœur de tous les systèmes d'information. C'est pourquoi tout ces problèmes ont été largement étudiés et des solutions fiables et éprouvées ont été trouvées. De nombreux travaux ont ainsi permis de mettre au point une théorie permettant la conception de bases de données *bien formées*. C'est la problématique que nous abordons dans le chapitre 2. La problématique de la gestion des bases de données trouve une solution dans l'utilisation d'un SGBD.

Pour toutes ces raisons, j'espère que l'intérêt la théorie sur la conception des bases de données ainsi que l'intérêt de l'utilisation des systèmes de gestion de base de données deviennent évidant pour vous.

Chapitre 2

Conception des bases de données : le modèle entités-associations

2.1 Introduction

2.1.1 Pourquoi une modélisation préalable ?

Il est difficile de modéliser un domaine sous une forme directement utilisable par un SGBD. Une ou plusieurs modélisations intermédiaires sont donc utiles, le modèle entités-associations constitue l'une des premières et des plus courantes. Ce modèle, présenté par Chen (1976), permet une description naturelle du monde réel à partir des concepts d'entité et d'association[1]. Basé sur la théorie des ensembles et des relations, ce modèle se veut universel et répond à l'objectif d'indépendance données-programmes. Ce modèle, utilisé pour la phase de conception, s'inscrit notamment dans le cadre d'une méthode plus générale et très répandue : *Merise*.

2.1.2 Merise

MERISE (Méthode d'Étude et de Réalisation Informatique pour les Systèmes d'Entreprise) est certainement le langage de spécification le plus répandu dans la communauté de l'informatique des systèmes d'information, et plus particulièrement dans le domaine des bases de données. Une représentation Merise permet de valider des choix par rapport aux objectifs, de quantifier les solutions retenues, de mettre en œuvre des techniques d'optimisation et enfin de guider jusqu'à l'implémentation. Reconnu comme standard, Merise devient un outil de communication. En effet, Merise réussit le compromis difficile entre le souci d'une modélisation précise et formelle, et la capacité d'offrir un outil et un moyen de communication accessible aux non-informaticiens.

Un des concepts clés de la méthode Merise est la séparation des données et des traitements. Cette méthode est donc parfaitement adaptée à la modélisation des problèmes abordés d'un point de vue fonctionnel[2]. Les données représentent la *statique* du système d'information et les traitements sa *dynamique*. L'expression conceptuelle des données conduit à une modélisation des données en *entités* et en *associations*. Dans ce cours, nous écartons volontairement la modélisation des traitements puisque nous ne nous intéressons à la méthode Merise que dans la perspective de la modélisation de bases de données.

Merise propose une démarche, dite par niveaux, dans laquelle il s'agit de hiérarchiser les préoccupations de modélisation qui sont de trois ordres : la conception, l'organisation et la technique. En effet, pour aborder la modélisation d'un système, il convient de l'analyser en premier lieu de façon globale et de se concentrer sur sa fonction : c'est-à-dire de s'interroger sur ce qu'il fait avant de définir comment

[1] Dans la littérature, on utilise indifféremment le terme *relation* ou le terme *association*, on parle donc de modèle entités-relations (E-R) ou de modèle entités-associations (E-A). Nous préférons utiliser le terme *association* plutôt que le terme *relation* pour limiter la confusion avec les relations du modèle relationnel.

[2] *A contrario*, Merise n'est pas adapté à la modélisation des problèmes abordés d'une manière orientée objet (dans ce cas, il faut, par exemple, utiliser UML).

il le fait. Ces niveaux de modélisation sont organisés dans une double approche données/traitements. Les trois niveaux de représentation des données, puisque ce sont eux qui nous intéressent, sont détaillés ci-dessous.

Niveau conceptuel : le *modèle conceptuel des données (MCD)* décrit les entités du monde réel, en terme d'objets, de propriétés et de relations, indépendamment de toute technique d'organisation et d'implantation des données. Ce modèle se concrétise par un *schéma entités-associations* représentant la structure du système d'information, du point de vue des données.

Niveau logique : le *modèle logique des données (MLD)* précise le modèle conceptuel par des choix organisationnels. Il s'agit d'une transcription (également appelée dérivation) du MCD dans un formalisme adapté à une implémentation ultérieure, au niveau physique, sous forme de base de données relationnelle ou réseau, ou autres (cf. section 1.1.2). Les choix techniques d'implémentation (choix d'un SGBD) ne seront effectués qu'au niveau suivant.

Niveau physique : le *modèle physique des données (MPD)* permet d'établir la manière concrète dont le système sera mis en place (SGBD retenu).

2.2 Éléments constitutifs du modèle entités-associations

La représentation du modèle entités-associations s'appuie sur trois concepts de base :
- l'objet ou entité,
- l'association,
- la propriété.

L'objet est une entité ayant une existence propre. L'association est un lien ou relation entre objets sans existence propre. La propriété est la plus petite donnée d'information décrivant un objet ou une association.

2.2.1 Entité

FIG. 2.1 – Représentation graphique d'un exemple de type-entité.

Définition 2.1 -entité- *Une entité est un objet, une chose concrète ou abstraite qui peut être reconnue distinctement et qui est caractérisée par son unicité.*

Exemples d'entité : Jean Dupont, Pierre Bertrand, le livre que je tiens entre les mains, la Ferrari qui se trouve dans mon garage, etc.

Les entités ne sont généralement pas représentées graphiquement.

Définition 2.2 -type-entité- *Un type-entité désigne un ensemble d'entités qui possèdent une sémantique et des propriétés communes.*

Les personnes, les livres et les voitures sont des exemples de type-entité. En effet, dans le cas d'une personne par exemple, les informations associées (*i.e.* les *propriétés*), comme le nom et le prénom, ne changent pas de nature.

Une entité est souvent nommée occurrence ou instance de son type-entité.

La figure 2.1 montre la représentation graphique d'un exemple de type-entité (*Personne*) sans ses propriétés associées.

Les type-entité *Personne*, caractérisé par un nom et un prénom, et *Voiture*, caractérisé par un nom et une puissance fiscale, ne peuvent pas être regroupés car ils ne partagent leurs propriétés (le prénom est

une chaîne de caractères et la puissance fiscale un nombre). Les type-entité *Personne*, caractérisé par un nom et un prénom, et *Livre*, caractérisé un titre et un auteur, possèdent tous les deux deux attributs du type *chaîne de caractères*. Pourtant, ces deux type-entités ne peuvent pas être regroupés car ils ne partagent pas une même sémantique : le nom d'une personne n'a rien à voir avec le titre d'un livre, le prénom d'une personne n'a rien à voir avec un auteur.

Par abus de langage, on utilise souvent le mot entité en lieu et place du mot type-entité, il faut cependant prendre garde à ne pas confondre les deux concepts.

2.2.2 Attribut ou propriété, valeur

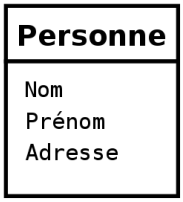


FIG. 2.2 – Représentation graphique d'un exemple de type-entité comportant trois attributs

Définition 2.3 **-attribut, propriété-** *Un attribut (ou une propriété) est une caractéristique associée à un type-entité ou à un type-association.*

Exemples d'attribut : le nom d'une personne, le titre d'une livre, la puissance d'une voiture.

Définition 2.4 **-valeur-** *Au niveau du type-entité ou du type-association, chaque attribut possède un domaine qui définit l'ensemble des valeurs possibles qui peuvent être choisies pour lui (entier, chaîne de caractères, booléen, . . .). Au niveau de l'entité, chaque attribut possède une* ***valeur*** *compatible avec son domaine.*

La figure 2.2 montre la représentation graphique d'un exemple de type-entité (*Personne*) avec trois attributs.

Règle 2.5 *Un attribut ne peut en aucun cas être partagé par plusieurs type-entités ou type-associations.*

Règle 2.6 *Un attribut est une donnée élémentaire, ce qui exclut des données calculées ou dérivées.*

Règle 2.7 *Un type-entité et ses attributs doivent être cohérents entre eux (*i.e. *ne traiter que d'un seul sujet).*

Par exemple, si le modèle doit comporter des informations relatives à des articles et à leur fournisseur, ces informations ne doivent pas coexister au sein d'un même type-entité. Il est préférable de mettre les informations relatives aux articles dans un type-entité *Article* et les informations relatives aux fournisseurs dans un type-entité *Fournisseur*. Ces deux type-entités seront probablement ensuite reliés par un type-association.

2.2.3 Identifiant ou clé

Définition 2.8 **-identifiant, clé-** *Un identifiant (ou clé) d'un type-entité ou d'un type-association est constitué par un ou plusieurs de ses attributs qui doivent avoir une valeur unique pour chaque entité ou association de ce type.*

Il est donc impossible que les attributs constituant l'identifiant d'un type-entité (respectivement type-association) prennent la même valeur pour deux entités (respectivement deux associations) distinctes. Exemples d'identifiant : le numéro de sécurité sociale pour une personne, le numéro d'immatriculation pour une voiture, le code ISBN d'un livre pour un livre (mais pas pour un exemplaire).

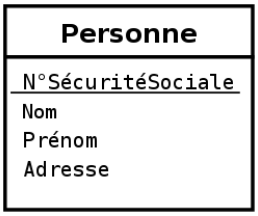


FIG. 2.3 – Représentation graphique d'un exemple de type-entité comportant quatre attributs dont un est un identifiant : deux personnes peuvent avoir le même nom, le même prénom et le même âge, mais pas le même numéro de sécurité sociale.

Règle 2.9 *Chaque type-entité possède au moins un identifiant, éventuellement formé de plusieurs attributs.*

Ainsi, chaque type-entité possède au moins un attribut qui, s'il est seul, est donc forcément l'identifiant.

Dans la représentation graphique, les attributs qui constituent l'identifiant sont soulignés et placés en tête (cf. figure 2.3).

2.2.4 Association ou relation

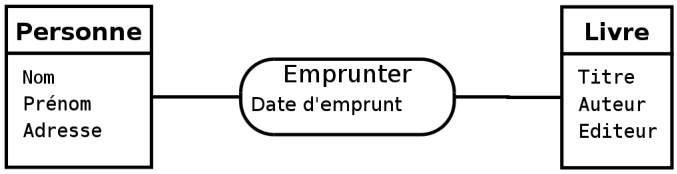


FIG. 2.4 – Représentation graphique d'un exemple de type-association liant deux type-entités.

Définition 2.10 -association- *Une association (ou une relation) est un lien entre plusieurs entités.*

Exemples d'association : l'emprunt par l'étudiant Tanidute du 3^{e} exemplaire du livre « Maîtrisez SQL ».

Les associations ne sont généralement pas représentées graphiquement.

Définition 2.11 -type-association- *Un type-association (ou un type-relation) désigne un ensemble de relations qui possèdent les mêmes caractéristiques. Le type-association décrit un lien entre plusieurs type-entités. Les associations de ce type-association lient des entités de ces type-entités.*

Comme les type-entités, les type-associations sont définis à l'aide d'attributs qui prennent leur valeur dans les associations.

Règle 2.12 *Un attribut peut être placé dans un type-association uniquement lorsqu'il dépend de toutes les entités liées par le type-association.*

Un type-association peut ne pas posséder d'attribut explicite et cela est relativement fréquent, mais on verra qu'il possède au moins des attributs implicites.

Exemples de type-association : l'emprunt d'un livre à la bibliothèque.

Une association est souvent nommée occurrence ou instance de son type-association.

La figure 2.4 montre la représentation graphique d'un exemple de type-association.

Par abus de langage, on utilise souvent le mot association en lieu et place du mot type-association, il faut cependant prendre garde à ne pas confondre les deux concepts.

Définition 2.13 **-participant-** *Les type-entités intervenant dans un type-association sont appelés les participants de ce type-association.*

Définition 2.14 **-collection-** *L'ensemble des participants d'un type-association est appelé la collection de ce type-association.*

Cette collection comporte au moins un type-entité (cf. section 2.3.2), mais elle peut en contenir plus, on parle alors de type-association *n-aire* (quand $n = 2$ on parle de type-association binaire, quand $n = 3$ de type-association ternaire, . . .).

Définition 2.15 **-dimension ou arité d'un type-association-** *La dimension, ou l'arité d'un type-association est le nombre de type-entités contenu dans la collection.*

Comme un type-entité, un type-association possède forcément un identifiant, qu'il soit explicite ou non.

Règle 2.16 *La concaténation des identifiants des type-entités liés à un type-association constitue un identifiant de ce type-association et cet identifiant n'est pas mentionné sur le modèle (il est implicite).*

Cette règle implique que deux instances d'un même type-association ne peuvent lier un même ensemble d'entités.

Souvent, un sous-ensemble de la concaténation des identifiants des type-entités liés suffit à identifier le type-association.

On admet également un identifiant plus naturel et explicite, à condition qu'il ne soit qu'un moyen d'exprimer plus simplement cette concaténation.

2.2.5 Cardinalité

FIG. 2.5 – Représentation graphique des cardinalités d'un type-association. Dans cet exemple pédagogique, on suppose qu'un livre ne peut posséder qu'un auteur.

Définition 2.17 **-cardinalité-** *La cardinalité d'une patte reliant un type-association et un type-entité précise le nombre de fois minimal et maximal d'interventions d'une entité du type-entité dans une association du type-association. La cardinalité minimale doit être inférieure ou égale à la cardinalité maximale.*

Exemple de cardinalité : une personne peut être l'auteur de 0 à n livre, mais un livre ne peut être écrit que par une personne (cf. figure 2.5).

Règle 2.18 *L'expression de la cardinalité est obligatoire pour chaque patte d'un type-association.*

Règle 2.19 *Une cardinalité minimal est toujours* 0 *ou* 1 *et une cardinalité maximale est toujours* 1 *ou n.*

Ainsi, si une cardinalité maximale est connue et vaut 2, 3 ou plus, alors nous considérons qu'elle est indéterminée et vaut n. En effet, si nous connaissons n au moment de la conception, il se peut que cette valeur évolue au cours du temps. Il vaut donc mieux considérer n comme inconnue dès le départ. De la même manière, on ne modélise pas des cardinalités minimales qui valent plus de 1 car ces valeurs sont également susceptibles d'évoluer. Enfin, une cardinalité maximale de 0 n'a pas de sens car elle rendrait le type-association inutile.

Les seuls cardinalités admises sont donc :

0,1 : une occurrence du type-entité peut exister tout en étant impliquée dans aucune association et peut être impliquée dans au maximum une association.

0,n : c'est la cardinalité la plus ouverte ; une occurrence du type-entité peut exister tout en étant impliquée dans aucune association et peut être impliquée, sans limitation, dans plusieurs associations.

1,1 : une occurrence du type-entité ne peut exister que si elle est impliquée dans exactement (au moins et au plus) une association.

1,n : une occurrence du type-entité ne peut exister que si elle est impliquée dans au moins une association.

Une cardinalité minimale de 1 doit se justifier par le fait que les entités du type-entité en questions ont besoin de l'association pour exister. Dans tous les autres cas, la cardinalité minimale vaut 0. Ceci dit, la discussion autour d'une cardinalité minimale de 0 ou de 1 n'est intéressante que lorsque la cardinalité maximale est 1. En effet, nous verrons que, lors de la traduction vers un schéma relationnel (cf. section 3.1.3), lorsque la cardinalité maximale est n, nous ne ferons pas la différence entre une cardinalité minimale de 0 ou de 1.

Remarques

La seule difficulté pour établir correctement les cardinalités est de se poser les question dans le bon sens. Pour augmenter le risque d'erreurs, il faut noter que, pour les habitués, ou les futurs habitués, du modèle UML, les cardinalités d'un type-association sont « à l'envers » (par référence à UML) pour les type-associations binaires et « à l'endroit » pour les n-aires avec $n > 2$.

La notion de cardinalité n'est pas définie de la même manière dans le modèle Américain et dans le modèle Européen (Merise). Dans le premier n'existe que la notion de cardinalité maximale.

Avec un SGBD relationnel, nous pourrons contraindre des cardinalités à des valeurs comme 2, 3 ou plus en utilisant des déclencheurs (*trigger*, cf. section **??**).

2.3 Compléments sur les associations

2.3.1 Associations plurielles

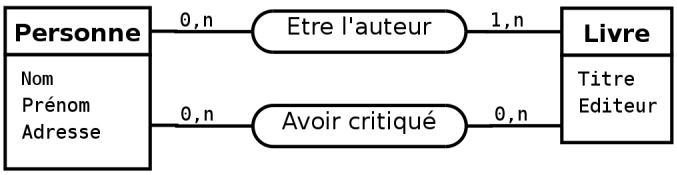


FIG. 2.6 – Exemple d'associations plurielles entre un type-entité *Personne* et un type-entité *Livre*. Sur ce schéma, un type-association permet de modéliser que des personnes écrivent des livres et un autre que des personnes critiquent (au sens de critique littéraire) des livres.

Deux mêmes entités peuvent être plusieurs fois en association (c'est le cas sur la figure 2.6).

2.3.2 Association réflexive

Les type-associations réflexifs sont présents dans la plupart des modèles.

Définition 2.20 -Type-association réflexif- *Un type-association est qualifié de réflexif quand il matérialise une relation entre un type-entité et lui-même (cf. figure 2.7).*

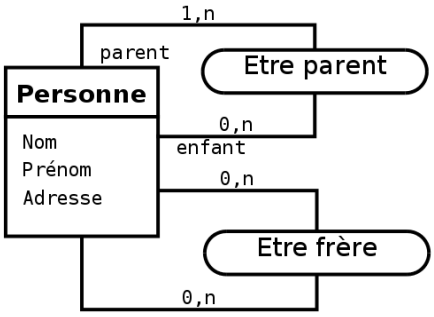


FIG. 2.7 – Exemple d'associations reflexives sur le type-entité *Personne*. Le premier type-association permet de modéliser la relation parent/enfant et le deuxième type-association la relation de fraternité.

Une occurrence de ce type-association (*i.e.* une association) associe généralement une occurrence du type-association (*i.e.* une entité) à une autre entité du même type. Cette relation peut être symétrique, c'est le cas du type-association *Etre frère* sur la figure 2.7, ou ne pas l'être, comme le type-association *Etre parent* sur cette même figure. Dans le cas où la relation n'est pas symétrique, on peut préciser les rôles sur les pattes du type-association comme pour la relation *Etre parent* de la figure 2.7. L'ambiguïté posée par la non-symétrie d'un type-association réflexif sera levée lors du passage au modèle relationnel (cf. section 3.1.3).

2.3.3 Association n-aire ($n > 2$)

Dans la section 2.2.4 nous avons introduit la notion de type-association *n-aire*. Ce type-association met en relation n type-entités. Même s'il n'y a, en principe, pas de limite sur l'arité d'un type-association, dans la pratique on ne va rarement au-delà de trois. Les associations de degré supérieur à deux sont plus difficiles à manipuler et à interpréter, notamment au niveau des cardinalités.

Exemple d'association n-aire inappropriée

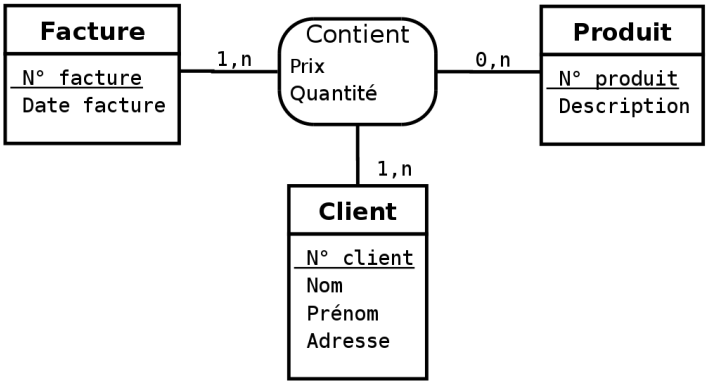


FIG. 2.8 – Exemple de type-association ternaire inapproprié.

Le type-association ternaire *Contient* associant les type-entités *Facture*, *Produit* et *Client* représenté sur la figure 2.8 est inapproprié puisqu'une facture donnée est toujours adressée au même client. En effet, cette modélisation implique pour les associations (instances du type-association) *Contient* une répétition du numéro de client pour chaque produit d'une même facture.

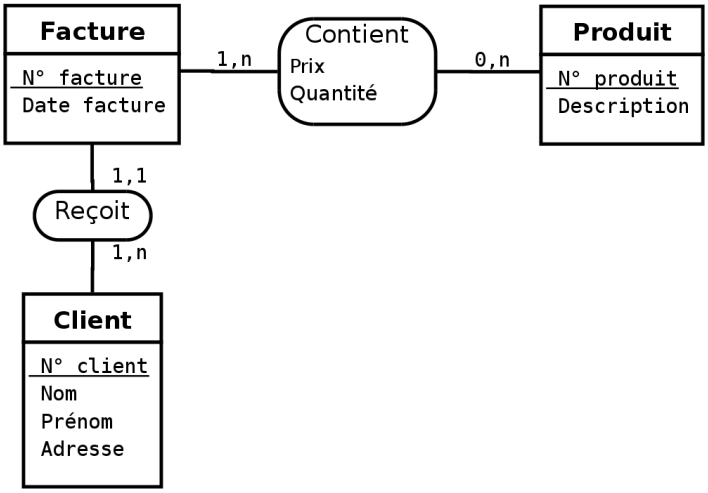


FIG. 2.9 – Type-association ternaire de la figure 2.8 corrigé en deux type-associations binaires.

La solution consiste à éclater le type-association ternaire *Contient* en deux type-associations binaires comme représenté sur la figure 2.9.

Décomposition d'une association n-aire

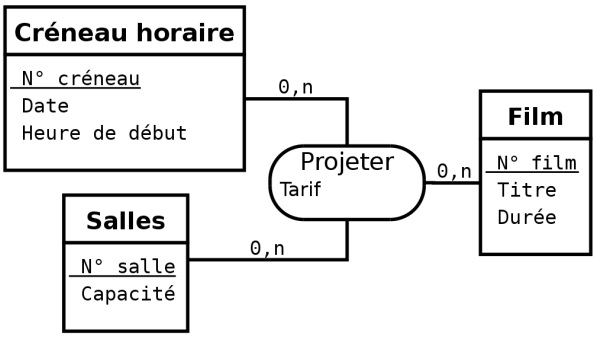


FIG. 2.10 – Exemple de type association ternaire entre des type-entités *Créneau horaire*, *Salle* et *Film*.

La figure 2.10 nous montre un exemple de type-association ternaire entre les type-entités *Créneau horaire*, *Salle* et *Film*. Il est toujours possible de s'affranchir d'un type-association n-aire ($n > 2$) en se ramenant à des type-associations binaires de la manière suivante :

- On remplace le type-association n-aire par un type-entité et on lui attribut un identifiant.
- On crée des type-associations binaire entre le nouveau type-entité et tous les type-entités de la collection de l'ancien type-association n-aire.
- La cardinalité de chacun des type-associations binaires créés est $1,1$ du côté du type-entité créé (celui qui remplace le type-association n-aire), et $0,n$ ou $1,n$ du côté des type-entités de la collection de l'ancien type-association n-aire.

La figure 2.11 illustre le résultat de cette transformation sur le schéma de la figure 2.10.

L'avantage du schéma de la figure 2.11 est de rendre plus intelligible la lecture des cardinalités. Il ne faut surtout pas le voir comme un aboutissement mais comme une étape intermédiaire avant d'aboutir au schéma de la figure 2.10 (cf. règle 2.27). Ainsi, le mécanisme, que nous venons de détailler ci-dessus,

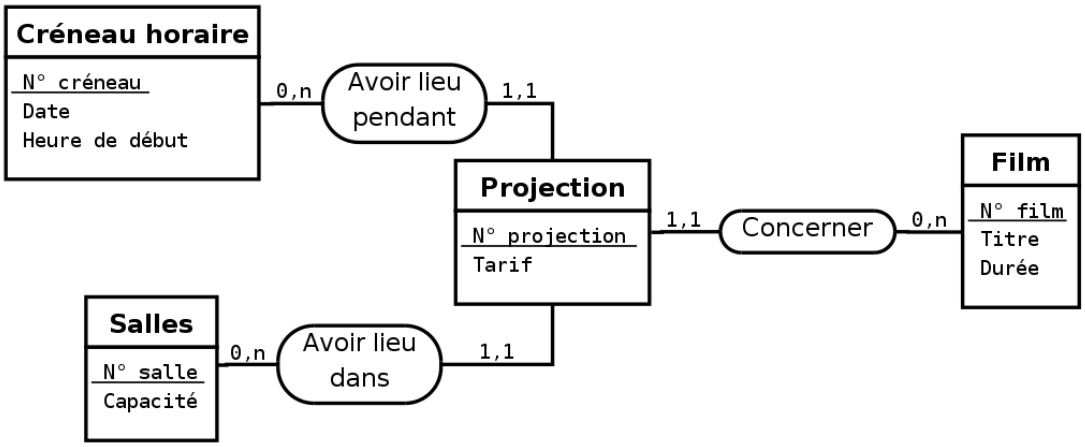


FIG. 2.11 – Transformation du type-association ternaire de la figure 2.10 en un type-entité et trois type-associations binaires.

de passage d'un type-association n-aire ($n > 2$) à un type-entité et n type-associations binaires est tout à fait réversible à condition que :

- toutes les pattes des type-associations binaires autour du type-entité central ont une cardinalité maximale de 1 au centre et de n à l'extérieur ;
- les attributs du type-entité central satisfont la règle de bonne formation des attributs de type-association (cf. section 2.5.2).

Détection d'une erreur de modélisation par décomposition d'une association n-aire

Passer par cette étape intermédiaire ne comportant pas de type-association n-aire ($n > 2$) peut, dans certains cas, éviter d'introduire un type-association n-aire inapproprié. Imaginons par exemple un type-association ternaire *Vol* liant trois type-entités *Avion*, *Trajet* et *Pilote* comme représenté sur la figure 2.12.

La transformation consistant à supprimer le type-association ternaire du modèle de la figure 2.12 produit le modèle de la figure 2.13. Ce modèle fait immédiatement apparaître une erreur de conception qui était jusque là difficile à diagnostiquer : généralement, à un vol donné sont affectés plusieur pilotes (par exemple le commandant de bord et un copilote) et non pas un seul.

Le modèle correct modélisant cette situation est celui de la figure 2.14 où le type-entité *Vol* ne peut être transformé en un type-association ternaire *Vol* comme sur la figure 2.12.

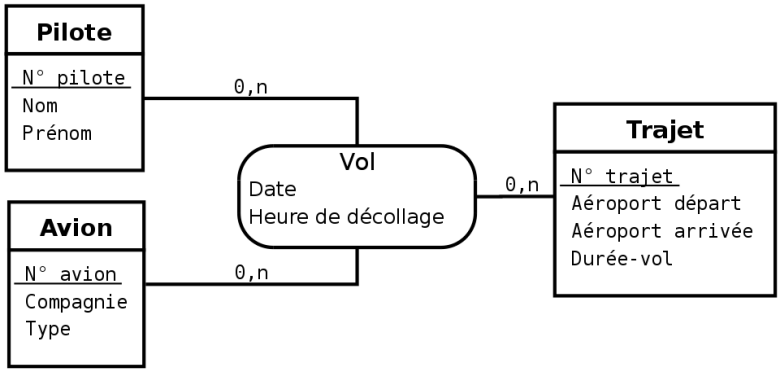


FIG. 2.12 – Modèle représentant un type-association ternaire *Vol* liant trois type-entités *Avion*, *Trajet* et *Pilote*.

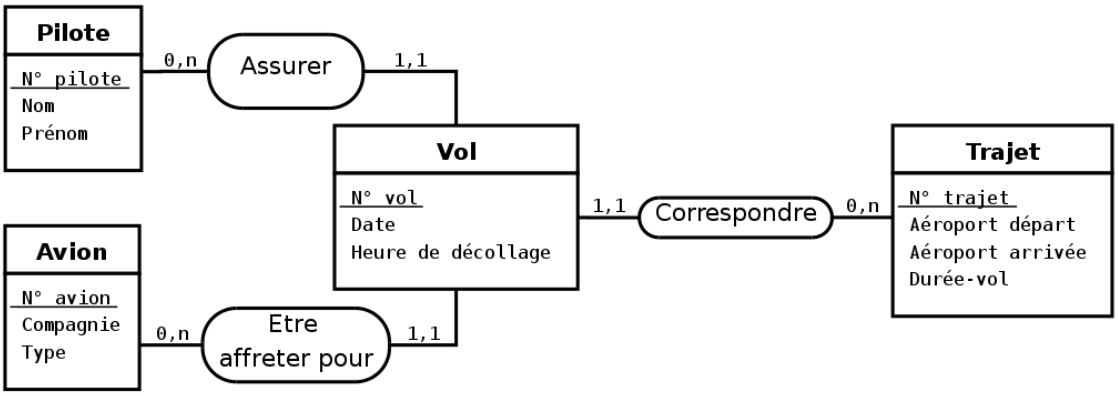


FIG. 2.13 – Transformation du type-association ternaire de la figure 2.12 en un type-entité et trois type-associations binaires.

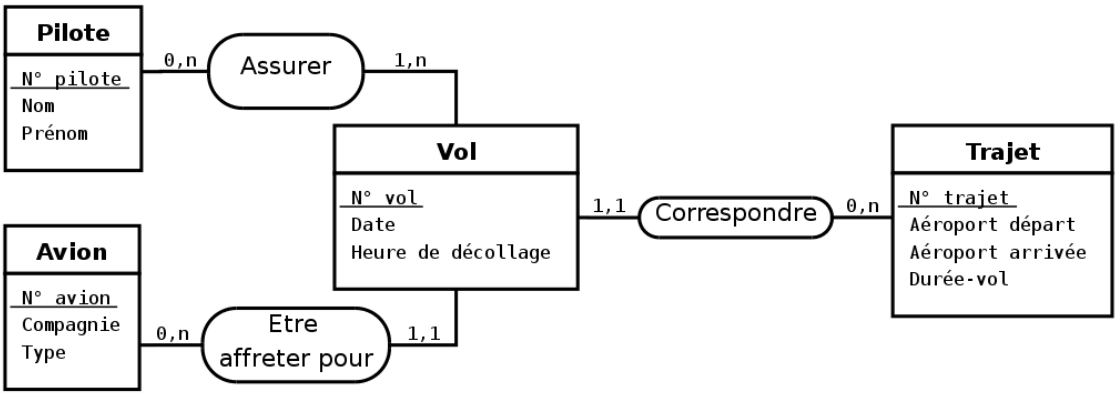


FIG. 2.14 – Modèle de la figure 2.13 corrigé au niveau des cardinalités.

2.4 Travaux Dirigés – Modèle entités-associations (1re partie)

2.4.1 Attention aux attributs multiples

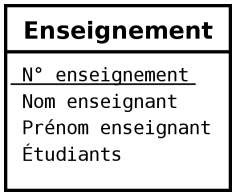


FIG. 2.15 – Modélisation incorrecte d'un enseignement.

On désire modéliser par un modèle entités-associations le fait qu'un enseignement est dispensé par un enseignant à plusieurs étudiants qui ne suivent qu'un enseignement. On vous propose la modélisation représentée sur la figure 2.15.

1. Critiquez cette modélisation.
2. Proposez-en une correcte.

2.4.2 Étudiants, cours, enseignants, salles, . . .

Modélisez indépendamment les situations suivantes :

3. Plusieurs cours sont offerts. Un cours peut être suivi par plusieurs étudiants et un étudiant peut s'inscrire à plusieurs cours. Pour chaque cours, on veut connaître la liste des étudiants et leur note (chaque cours ne comporte qu'une seule évaluation).
4. Plusieurs cours sont offerts. Un cours est dispensé par un seul enseignant et un enseignant peut dispenser plusieurs cours. Pour chaque cours, on veut connaître l'enseignant qui le dispense.

On s'intéresse maintenant à la modélisation d'une situation globale et plus complexe :
- Il existe plusieurs matières (mathématiques, sciences-physiques, français, anglais, philosophie).
- Plusieurs cours sont offerts et il peut y avoir plusieurs cours de la même matière.
- Un cours est dispensé par un, et un seul, enseignant et correspond à une matière.
- Un enseignant peut dispenser plusieurs cours dans la même matière ou dans des matières différentes.
- Un étudiant peut s'inscrire à plusieurs cours.
- Un cours est toujours dispensé dans une même salle, mais une salle peut recevoir plusieurs cours (successivement).
- Chaque cours ne comporte qu'une seule évaluation.

5. Proposez un modèle entités-associations permettant de modéliser la situation décrite ci-dessus.

2.4.3 Deux instances d'un même type-association ne peuvent lier un même ensemble d'entités

Considérons la modélisation de la figure 2.16 qui exprime qu'un client commande des produits chez un fournisseur.

6. Imaginons qu'un même client commande un même produit chez un même fournisseur plus d'une fois. Cette situation est-elle compatible avec le modèle ?
7. Proposez une amélioration de ce modèle.

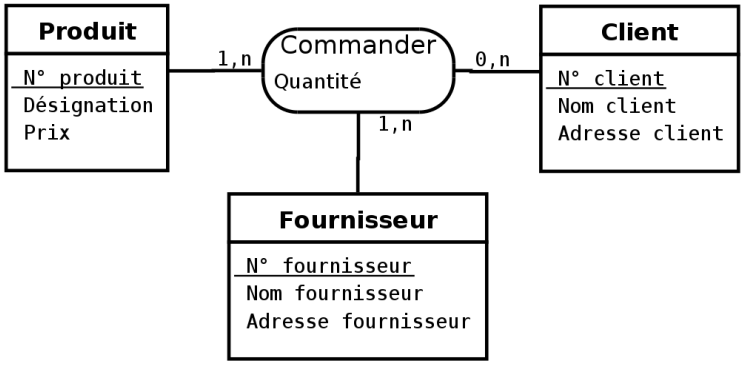


FIG. 2.16 – Le type-association *Commande* lie les type-entités *Produit*, *Client* et *Fournisseur*.

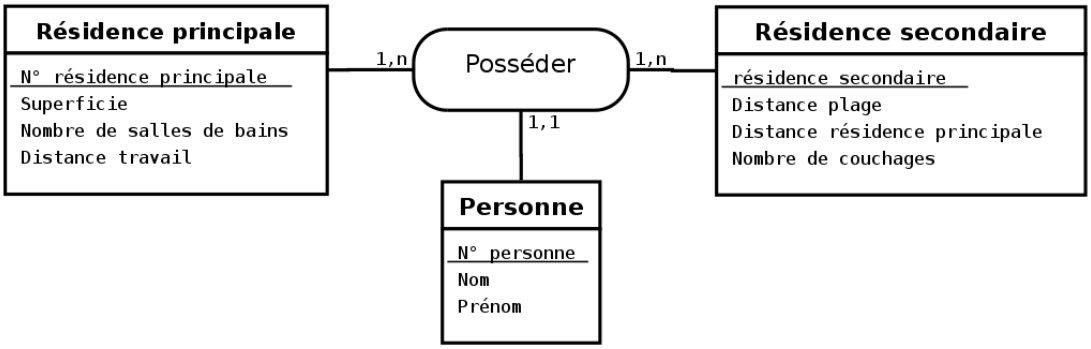


FIG. 2.17 – Modélisation des résidences principales et secondaires d'un ensemble de personnes.

2.4.4 Comprenez-vous les type-associations n-aire ?

On désire créer une base de données sur les résidences principales et secondaires d'un échantillon de la population possédant exactement une résidence principale et une résidence secondaire. Dans cette base, si une personne ne peut posséder plus d'une résidence, une résidence peut très bien appartenir à plusieurs personnes. Pour modéliser cette situation, on vous propose le modèle de la figure 2.17.

8. Expliquez la cardinalité 1 – 1 de l'une des pattes du type-association ternaire.
9. Critiquez cette solution.
10. Proposez un modèle corrigé.

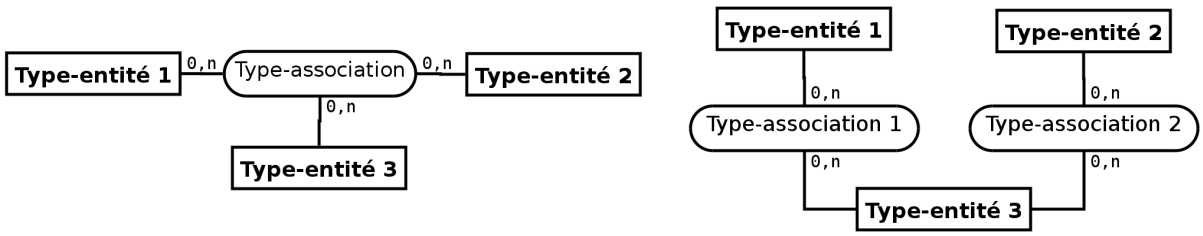


FIG. 2.18 – Ces deux modélisations ne sont pas des alternatives.

11. Les deux modèles de la figure 2.18 ne sont pas équivalents. Expliquez pourquoi.
12. Dans le cours, section 2.3.3, est exposé un exemple de détection d'une erreur de modélisation grâce à la décomposition d'une association n-aire. Discutez avec le chargé de TD du problème exposé dans cette section et illustré par le modèle de la figure 2.12.

2.4.5 Cas d'une bibliothèque (1re partie)

Une petite bibliothèque souhaite informatiser la gestion de son fonds documentaire et de ses emprunts. Dans cette perspective, le bibliothécaire, qui n'est pas un informaticien, a rédigé le texte suivant :

> Grâce à cette informatisation, un abonné devra pouvoir retrouver un livre en connaissant son titre. Il doit aussi pouvoir connaître la liste des livres d'un auteur. Un abonné a le droit d'emprunter au maximum dix ouvrages simultanément. Les prêts sont accordés pour une durée de quinze jours. La gestion des prêts doit permettre de connaître, à tout moment, la liste des livres détenus par un abonné, et inversement, de retrouver le nom des abonnés détenant un livre absent des rayons. Un livre peut être écrit par plusieurs auteurs. Chaque livre est acheté en un ou plusieurs exemplaires.

13. Identifiez, dans le texte ci-dessus, les mots devant se concrétiser par des entités, des associations ou des attributs.
14. Proposez un modèle entités-associations permettant de modéliser la situation décrite ci-dessus.

2.5 Règles de bonne formation d'un modèle entités-associations

La *bonne formation* d'un modèle entités-associations permet d'éviter une grande partie des sources d'incohérences et de redondance. Pour être bien formé, un modèle entités-associations doit respecter certaines règles et les type-entités et type-associations doivent être normalisées. Un bon principe de conception peut être formulé ainsi : « **une seule place pour chaque fait** ».

Bien que l'objectif des principes exposés dans cette section soit d'aider le concepteur à obtenir un diagramme entités-associations bien formé, ces principes ne doivent pas être interprété comme des lois. Qu'il s'agisse des règles de bonne formation ou des règles de normalisation, il peut exister, très occasionnellement, de bonnes raisons pour ne pas les appliquer.

2.5.1 Règles portant sur les noms

Règle 2.21 *Dans un modèle entités-associations, le nom d'un type-entité, d'un type-association ou d'un attribut doit être unique.*

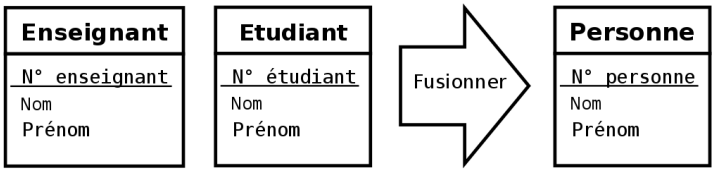


FIG. 2.19 – La présence des deux type-entités *Enseignant* et *Etudiant* est symptomatique d'une modélisation inachevée. A terme, ces deux type-entités doivent être fusionnés en un unique type-entité *Personne*. Référez vous à la règle 2.25 pour plus de précisions concernant cette erreur de modélisation.

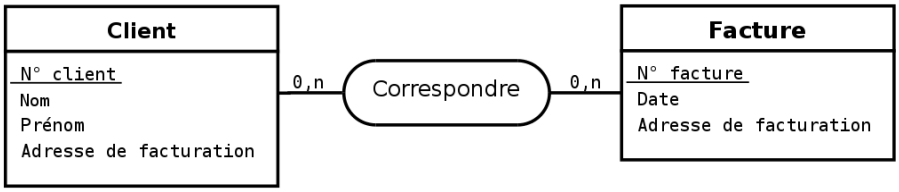


FIG. 2.20 – Ici, les attributs *Adresse de facturation* sont redondants. Cette situation doit être évitée à tout prix car elle entraîne un gaspillage d'espace mémoire mais aussi et **surtout un grand risque d'incohérence**. En effet, que faire si, dans le cadre d'une occurrence du type-association *Correspondre*, la valeurs des deux attributs *Adresse de facturation* diffèrent ?

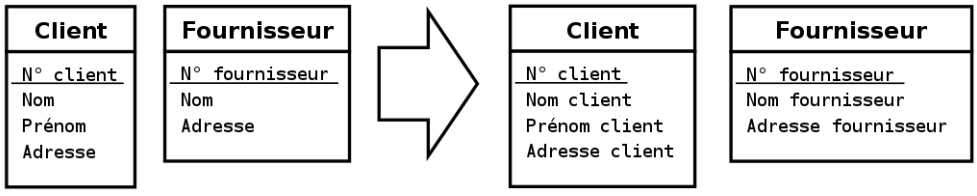


FIG. 2.21 – Dans cette situation, les deux attributs *Adresse* doivent simplement être renommés en *Adresse client* et *Adresse fournisseur*. Il en va de même pour les deux attributs *Nom*.

Lorsque des attributs portent le même nom, c'est parfois le signe d'une modélisation inachevée (figure 2.19) ou d'une redondance (figure 2.20). Sinon, il faut simplement ajouter au nom de l'attribut le nom du type-entité ou du type-association dans lequel il se trouve (figure 2.21). Il faut toutefois remarquer que le dernier cas décrit n'est pas rédhibitoire et que les SGDB Relationnel s'accommodent très bien de relations comportant des attributs de même nom. L'écriture des requêtes sera tout de même plus lisible si les attributs ont tous des noms différents.

2.5.2 Règles de normalisation des attributs

Règle 2.22 *Il faut remplacer un attribut multiple en un type-association et un type-entité supplémentaires.*

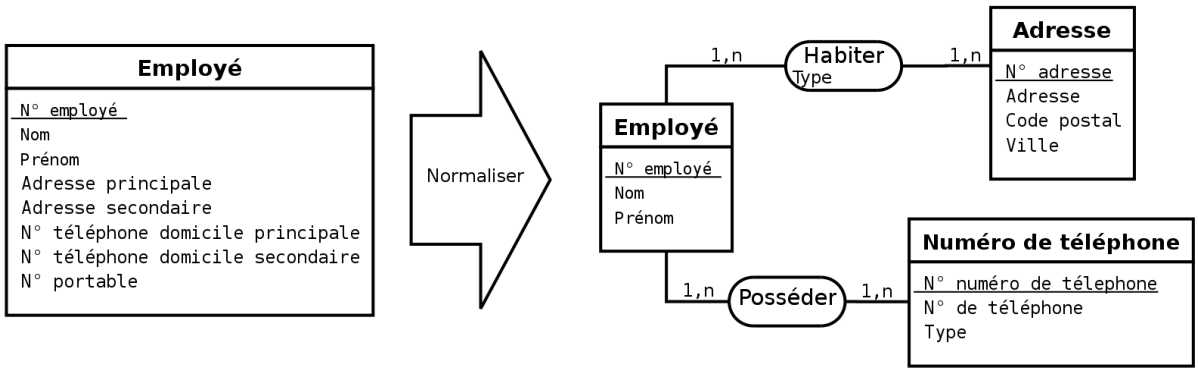


FIG. 2.22 – Remplacement des attributs multiples en un type-association et un type-entité et décomposition des attributs composites.

En effet, les attributs multiples posent régulièrement des problèmes d'évolutivité du modèle. Par exemple, sur le modèle de gauche de la figure 2.22, comment faire si un employé possède deux adresses secondaires ou plusieurs numéros de portable ?

Il est également intéressant de décomposer les attributs composites comme l'attribut *Adresse* par exemple. Il est en effet difficile d'écrire une requête portant sur la ville où habitent les employés si cette information est noyée dans un unique attribut *Adresse*.

Règle 2.23 *Il ne faut jamais ajouter un attribut dérivé d'autres attributs, que ces autres attributs se trouvent dans le même type-entité ou pas.*

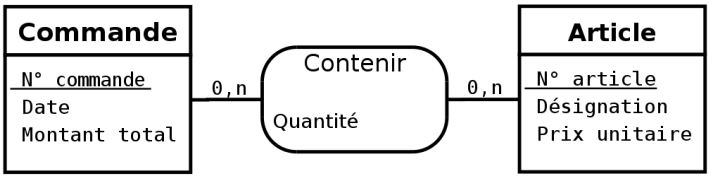


FIG. 2.23 – Il faut supprimer l'attribut *Montant total* du type-entité *Commande* car on peut le calculer à partir des attributs *Quantité* du type association *Contenir* et *Prix unitaire* du type-entité *Article*.

En effet, les attributs dérivés induisent un risque d'incohérence entre les valeurs des attributs de base et celles des attributs dérivés. La figure 2.23 illustre le cas d'un attribut *Montant total* dans un type-entité *Commande* qui peut être calculé à partir des attributs *Quantité* du type association *Contenir* et *Prix unitaire* du type-entité *Article*. Il faut donc supprimer l'attribut *Montant total* dans le type-entité *Commande*. D'autres attributs dérivés sont également à éviter comme l'âge, que l'on peut déduire de la

date de naissance et de la date courante. Il faut cependant faire attention aux pièges : par exemple, le code postal ne détermine ni le numéro de département ni la Ville[3]

Comme nous l'avons déjà dit (cf. règle 2.12), les attributs d'un type-association doivent dépendre directement des identifiants de *tous* les type-entités de la collection du type-association.

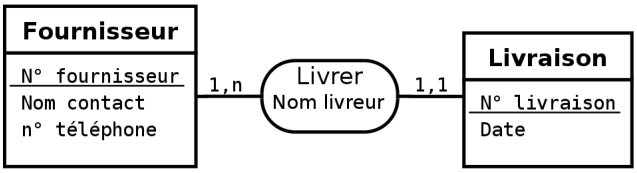


FIG. 2.24 – Comme la cardinalité maximale du type-association *Livrer* est 1 du côté du type-entité *Livraison*, l'attribut *Nom livreur* de *Livrer* doit être déplacé dans *Livraison*.

Par exemple, sur la figure 2.23, l'attribut *Quantité* du type-association *Contenir* dépend bien à la fois de l'identifiant *N° commande* et de *N° article* des type-entités de la collection de *Contenir*. Inversement, sur cette même figure, l'attribut *Prix-unitaire* ne dépend que de *N° article* du type-entité *Article*, il ne pourait donc pas être un attribut du type-association *Contenir*. Une conséquence immédiate de cette règle est qu'un type association dont la cardinalité maximale de l'une des pattes est 1 ne peut pas posséder d'attribut. Si elle en possédait, ce serait une erreur de modélisation et il faudrait les déplacer dans le type-entité connecté à la patte portant la cardinalité maximale de 1 (cf. figure 2.24).

Règle 2.24 *Un attribut correspondant à un type énuméré est généralement avantageusement remplacé par un type-entité.*

Par exemple, sur la figure 2.25, l'attribut *Type* caractérise le type d'une émission et peut prendre des valeurs comme : *actualité*, *culturelle*, *reportage*, *divertissement*, etc. Remplacer cet attribut par un type-entité permet, d'une part, d'augmenter la cohérence (en s'affranchissant, par exemple, des variations du genre *culturelle*, *culture*, *Culture*, ...) et d'autre part, si les cardinalités le permettent, de pouvoir affecter plusieurs types à une même entité (ex : *actualité* et *culturelle*)

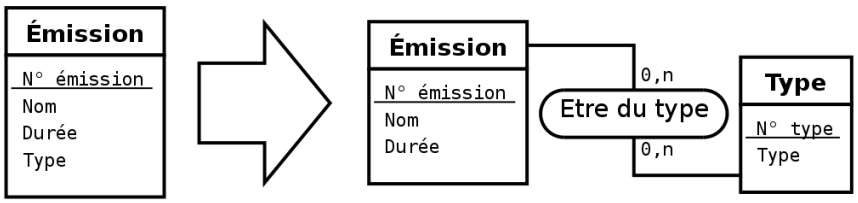


FIG. 2.25 – Un attribut correspondant à un type énuméré est généralement avantageusement remplacé par un type-entité..

2.5.3 Règles de fusion/suppression d'entités/associations

Règle 2.25 *Il faut factoriser les type-entités quand c'est possible.*

La spécialisation du type-entité obtenu peut se traduire par l'introduction d'un attribut supplémentaire dont l'ensemble des valeurs possibles est l'ensemble des noms des type-entités factorisés (figure 2.26).

[3] Le code postal en France identifie le bureau distributeur qui achemine le courrier dans une commune. En conséquence et d'après cette définition, il n'existe pas de relation entre le code postal et le code du département de la commune. Par exemple,

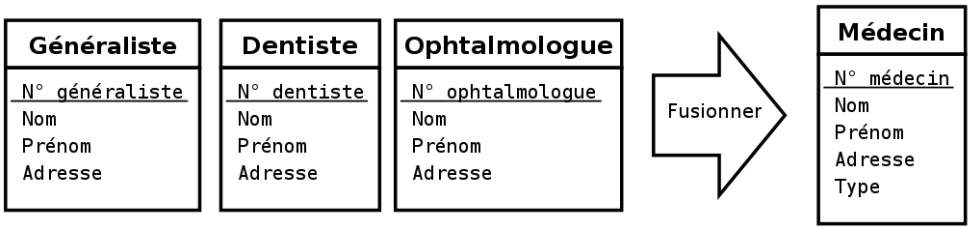


FIG. 2.26 – Il faut factoriser les type-entités quand c'est possible, éventuellement en introduisant un nouvel attribut.

Mais l'introduction d'un attribut supplémentaire n'est pas forcément nécessaire ou souhaitable. Par exemple, sur le modèle entités-associations final de la figure 2.27, on peut distinguer les entités qui correspondent à des écrivains ou des abonnés en fonction du type de l'association, *Ecrire* ou *Emprunter*, que l'entité en question entretient avec une entité du type *Livre*. Ne pas introduire d'attribut permet en outre de permettre à une personne d'être à la fois un *Abonné* et un *Écrivain*.

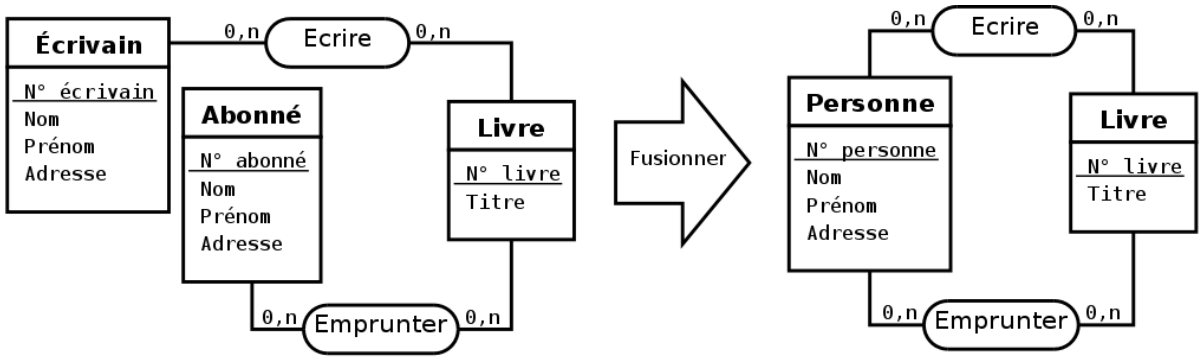


FIG. 2.27 – Il faut factoriser les type-entités quand c'est possible, mais l'introduction d'un attribut supplémentaire n'est pas toujours nécessaire. Remarque : ce diagramme est intentionnellement simplifié à outrance.

Règle 2.26 *Il faut factoriser les type-associations quand c'est possible.*

Cette règles est le pendant pour les type-associations de la règle 2.25 qui concerne les type-entités. La spécialisation du type-association obtenu peut se traduire par l'introduction d'un attribut supplémentaire dont l'ensemble des valeurs possibles est l'ensemble des noms des type-associations factorisés.

La figure 2.28 montre un exemple de multiplication inutile de type-associations.

Règle 2.27 *Un type-entité remplaçable par un type-association doit être remplacé.*

Par exemple, le type-entité *Projection* de la figure 2.11 page 25 doit être remplacé par le type-association ternaire *Projeter* pour aboutir au schéma de la figure 2.10 page 24.

Règle 2.28 *Lorsque les cardinalités d'un type-association sont toutes* 1, 1 *c'est que le type-association n'a pas lieu d'être.*

---

la commune *La Feuillade*, dont le code postal est 19600, est située dans le département de la *Dordogne* (24). Dans cette non correspondance entre code postal et département, il y a toute la Corse ! Il n'y a pas non plus de correspondance biunivoque entre le code postal et une ville. Une commune peut avoir plusieurs codes postaux, un code postal peut recouvrir plusieurs communes.

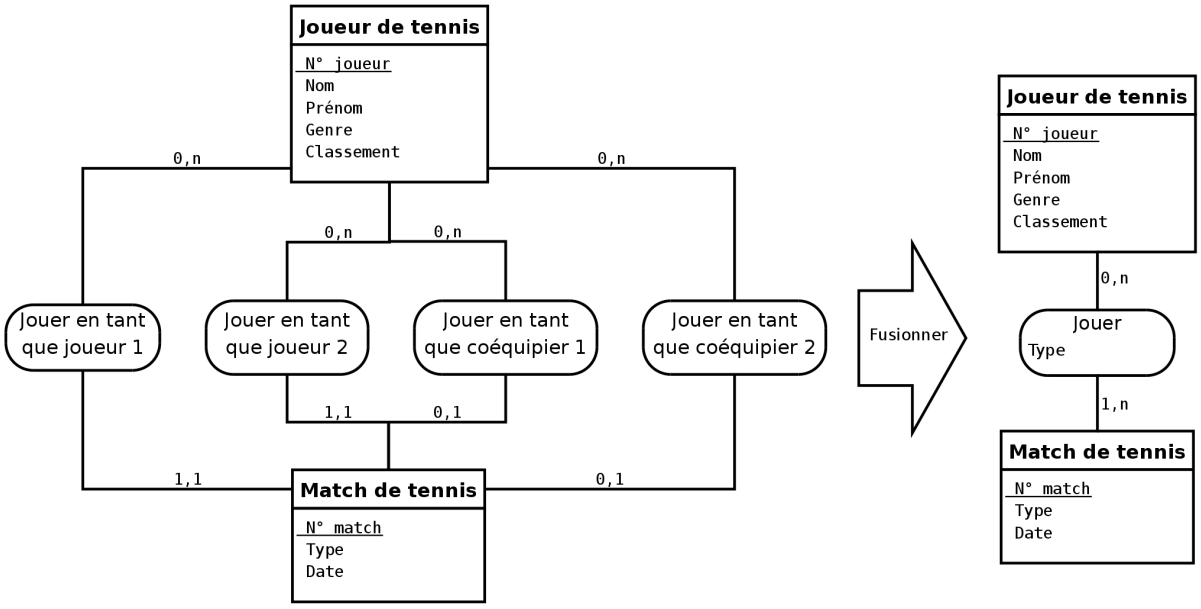


FIG. 2.28 – Un seul type-association suffit pour remplacer les quatre type-associations *Jouer en tant que . . .*

Il faut aussi se poser la question de l'intérêt du type-association quand les cardinalités maximale sont toutes de 1.

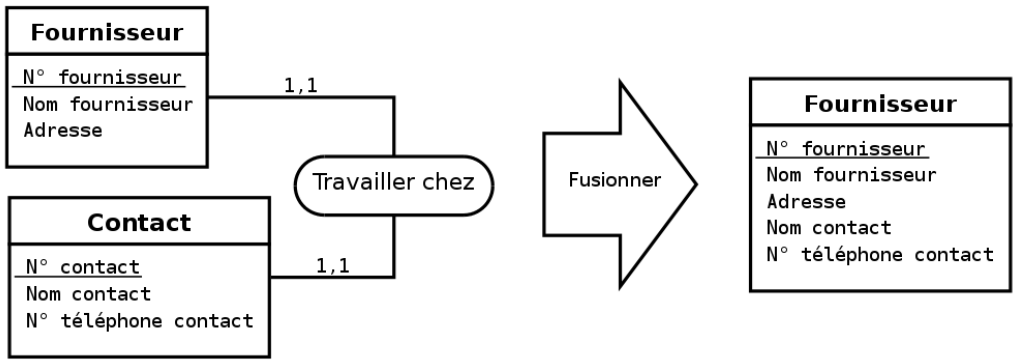


FIG. 2.29 – Lorsque les cardinalités d'un type-association sont toutes 1,1 c'est qu'il s'agit d'un type-association fantôme.

Lorsque les cardinalités d'un type-association sont toutes 1,1, le type-association doit généralement être supprimé et les type-entités correspondant fusionnés comme l'illustre la figure 2.29. Néanmoins, même si toutes ses cardinalités maximale sont de 1, il est parfois préférable de ne pas supprimer le type-association, comme dans l'exemple de la figure 2.30.

Règle 2.29 *Il faut veiller à éviter les type-associations redondants. En effet, s'il existe deux chemins pour se rendre d'un type-entité à un autre, alors ces deux chemins doivent avoir deux significations ou deux durées de vie distinctes. Dans le cas contraire, il faut supprimer le chemin le plus court puisqu'il est déductible des autres chemins.*

Par exemple, dans le modèle représenté sur la figure 2.31, si un client ne peut pas régler la facture d'un autre client, alors le type-association *Payer* est redondant et doit purement et simplement être supprimé

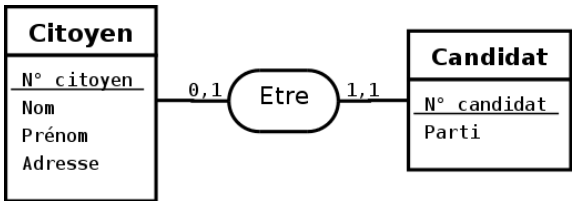


FIG. 2.30 – Même si toutes les cardinalités maximale sont de 1, il vaut mieux conserver le type-association *Etre*.

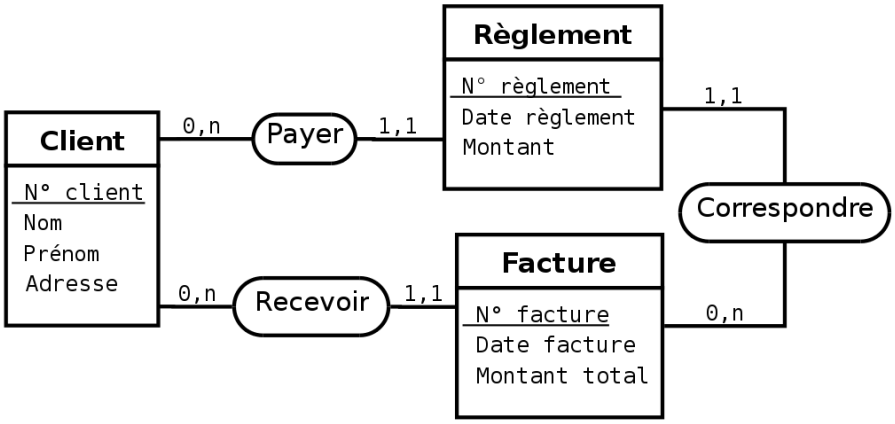


FIG. 2.31 – Si un client ne peut pas régler la facture d'un autre client, alors le type-association *Payer* est inutile.

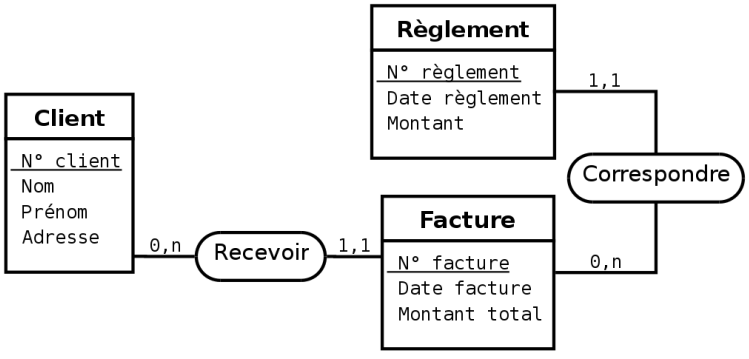


FIG. 2.32 – Solution au problème de la redondance du type-association de la figure 2.31.

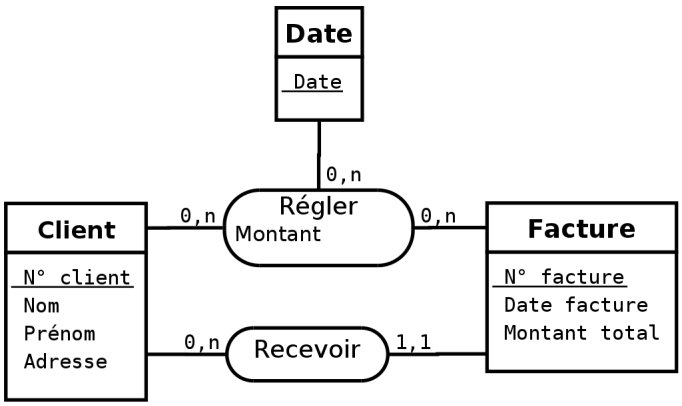


FIG. 2.33 – Dans le modèle de la figure 2.31, si un client peut régler la facture d'un autre client, il faut remplacer le type-entité *Règlement* par un type-association *Régler*.

du modèle (cf. figure 2.32). On pourra toujours retrouver le client qui a effectué un règlement en passant par la facture correspondante.

Par contre, si un client peut régler la facture d'un autre client, alors c'est la règle 2.27 qu'il faut appliquer : on remplace le type-entité *Règlement* par un type-association *Régler* (cf. figure 2.33).

2.5.4 Normalisation des type-entités et type-associations

Introduction

Les formes normales sont différent stades de qualité qui permettent d'éviter la redondance, source d'anomalies. La normalisation peut être aussi bien effectuée sur un modèle entités-associations, où elle s'applique sur les type-entités et type-associations, que sur un modèle relationnel.

Il existe 5 formes normales principales et deux extensions. Plus le niveau de normalisation est élevé, plus le modèle est exempte de redondances. Un type-entité ou un type-association en forme normale de niveau n est automatiquement en forme normale de niveau $n-1$. Une modélisation rigoureuse permet généralement d'aboutir directement à des type-entités et type-associations en forme normale de Boyce-Codd.

Nous avons décidé de présenter deux fois cette théorie de la normalisation :

- Une première fois, dans le cadre du modèle entités-associations (la présente section 2.5.4), en privilégiant une approche plus intuitive qui n'introduit pas explicitement la notion de dépendance fonctionnelle (et encore moins les notions de dépendance multivaluée et de jointure). Nous nous arrêterons, dans cette section, à la forme normale de Boyce-Codd.
- Puis une seconde fois, dans le cadre de modèle relationnel (section 3.2), en privilégiant une approche plus formelle s'appuyant sur la définition des dépendances fonctionnelle, multivaluée et de jointure. Nous irons alors jusqu'à la cinquième forme normale.

Première forme normale (1FN)

Définition 2.30 -Première forme normale (1FN)- *Un type-entité ou un type-association est en première forme normale si tous ses attributs sont élémentaires, c'est-à-dire non décomposables.*

Un attribut composite doit être décomposés en attributs élémentaires (comme l'attribut *Adresse* sur la figure 2.34) ou faire l'objet d'une entité supplémentaire (comme l'attribut *Occupants* sur la figure 2.34.

L'*élémentarité* d'un attribut est toutefois fonction des choix de gestion. Par exemple, la propriété *Adresse* peut être considérée comme élémentaire si la gestion de ces adresses est globale. Par contre, s'il faut pouvoir considérer les codes postaux, les noms de rues, . . ., il convient d'éclater la propriété *Adresse*

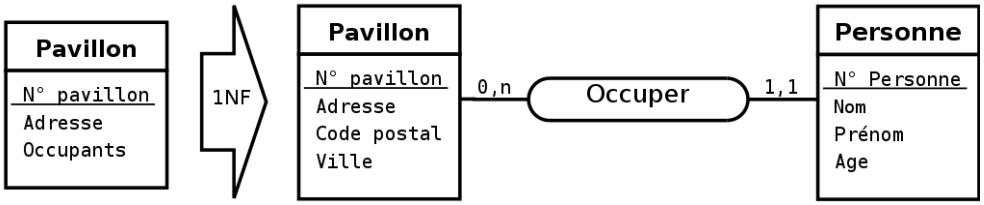


FIG. 2.34 – Exemple de normalisation en première forme normale.

en *Adresse* (au sens numéro d'appartement, numéro et nom de rue, ...), *Code postal* et *Ville*. En cas de doute, il est préférable (car plus général) d'éclater une propriété que d'effectuer un regroupement.

Deuxième forme normale (2FN)

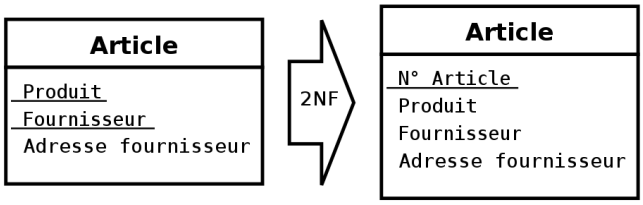


FIG. 2.35 – Exemple de normalisation en deuxième forme normale. On suppose qu'un même fournisseur peut fournir plusieurs produits et qu'un même produit peut être fourni par différents fournisseurs.

Définition 2.31 -Deuxième forme normale (2FN)- *Un type-entité ou un type-association est en deuxième forme normale si, et seulement si, il est en première forme normale et si tout attribut n'appartenant pas à la clé dépend de la totalité de cette clé.*

Autrement dit, les attributs doivent dépendre de l'ensemble des attributs participant à la clé. Ainsi, si la clé est réduite à un seul attribut, ou si elle contient tous les attributs, le type-entité ou le type-association est, par définition, forcément en deuxième forme normale.

La figure 2.35 montre un type-entité *Article* décrivant des produits provenant de différents fournisseurs. On suppose qu'un même fournisseur peut fournir plusieurs produits et qu'un même produit peut être fourni par différents fournisseurs. Dans ce cas, les attributs *Produit* ou *Fournisseur* ne peuvent constituer un identifiant du type-entité *Article*. Par contre, le couple *Produit/Fournisseur* constitue bien un identifiant du type-entité *Article*. Cependant, l'attribut *Adresse fournisseur* ne dépend maintenant que d'une partie de la clé (*Fournisseur*). Opter pour une nouvelle clé arbitraire réduite à un seul attribut *N° article* permet d'obtenir un type-entité *Article* en deuxième forme normale. On va voir dans ce qui suit que cette solution n'a fait que déplacer le problème.

Troisième forme normale (3FN)

Définition 2.32 -Troisième forme normale (3FN)- *Un type-entité ou un type-association est en troisième forme normale si, et seulement si, il est en deuxième forme normale et si tous ses attributs dépendent directement de sa clé et pas d'autres attributs.*

Cette normalisation peut amener à désimbriquer des type-entités cachées comme le montre la figure 2.36.

Un type-entité ou un type-association en deuxième forme normale avec au plus un attribut qui n'appartient pas à la clé est, par définition, forcément en troisième forme normale.

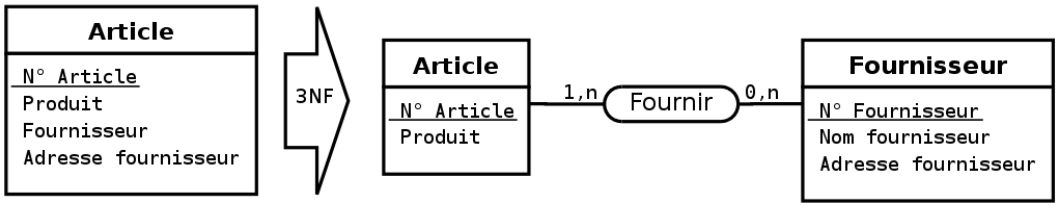


FIG. 2.36 – Exemple de normalisation en troisième forme normale. Dans cet exemple, l'attribut *Adresse fournisseur* dépend de l'attribut *Fournisseur*.

Forme normale de Boyce-Codd (BCNF)

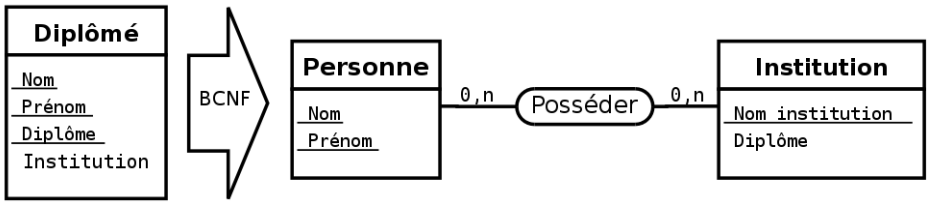


FIG. 2.37 – Exemple de normalisation en forme normale de Boyce-Codd.

Définition 2.33 -Forme normale de Boyce-Codd (BCNF)- *Un type-entité ou un type-association est en forme normale de Boyce-Codd si, et seulement si, il est en troisième forme normale et si aucun attribut faisant partie de la clé dépend d'un attribut ne faisant pas partie de la clé.*

Intéressons-nous, par exemple (cf. figure 2.37), à un type-entité *Diplômé* modélisant des personnes (*Nom* et *Prénom*) possédant un diplôme (*Diplôme*) d'une institution (*Institution*). On suppose qu'il n'y a pas d'homonyme, qu'une même personne ne possède pas deux fois le même diplôme mais qu'elle peut posséder plusieurs diplômes différents. Une institution ne délivre qu'un type de diplôme, mais un même diplôme peut être délivré par plusieurs institutions (par exemple, plusieurs écoles d'ingénieurs délivrent des diplômes d'ingénieur). Une clé possible pour le type-entité *Diplômé* est donc *Nom, Prénom, Diplôme*. Le type-entité obtenu est en troisième forme normale, mais une redondance subsiste car l'attribut *Institution* détermine l'attribut *Diplôme*. Le type-entité *Diplômé* n'est donc pas en forme normale de Boyce-Codd.

Un modèle en forme normale de Boyce-Codd est considéré comme étant de qualité suffisante pour une implantation.

Autres formes normales

Il existe d'autres formes normales. La quatrième et la cinquième forme normale sont présentées dans la section 3.2 dans le cadre du modèle relationnel.

2.6 Élaboration d'un modèle entités-associations

2.6.1 Étapes de conceptions d'un modèle entités-associations

Pour concevoir un modèle entités-associations, vous devrez certainement passer par une succession d'étapes. Nous les décrivons ci-dessous dans l'ordre chronologique. Sachez cependant que la conception d'un modèle entités-associations est un travail non linéaire. Vous devrez régulièrement revenir à une étape précédente et vous n'avez pas besoin d'en avoir terminé avec une étape pour commencer l'étape suivante.

Recueil des besoins – C'est une étape primordiale. Inventoriez l'ensemble des données à partir des documents de l'entreprise, d'un éventuel cahier des charges et plus généralement de tous les supports de l'information. N'hésitez pas à poser des questions.

Tri de l'information – Faites le tri dans les données recueillies. Il faut faire attention, à ce niveau, aux problèmes de synonymie/polysémie. En effet, les attributs ne doivent pas être redondants. Par exemple, si dans le langage de l'entreprise on peut parler indifféremment de *référence d'article* ou de *n° de produit* pour désigner la même chose, cette caractéristique ne devra se concrétiser que par un unique attribut dans le modèle. Inversement, on peut parler d'adresse pour désigner l'adresse du fournisseur et l'adresse du client, le contexte permettant de lever l'ambiguïté. Par contre, dans le modèle, il faudra veiller à bien distinguer ces deux caractéristiques par deux attributs distincts.

Un autre exemple est celui d'une entreprise de production fabricant des produits à destination d'une autre société du même groupe. Il se peut que dans ce cas, le prix de production (*i.e.* le coût de revient industriel) soit le même que prix de vente (aucune marge n'est réalisée). Même dans ce cas où les deux caractéristiques sont identiques pour chaque entité (prix de production égale prix de vente), il faut impérativement les scinder en deux attributs au niveau du type-entité *Produit*. Sinon, cette égalité factuelle deviendrait une contrainte imposée par le modèle, obligeant alors l'entreprise de production à revoir son système le jour où elle décidera de réaliser une marge (prix de production inférieure au prix de vente).

Identification des type-entités – Le repérage d'attributs pouvant servir d'identifiant permet souvent de repérer un type-entité. Les attributs de ce type-entité sont alors les attributs qui dépendent des attributs pouvant servir d'identifiant.

Attention, un même concept du monde réel peut être représenté dans certains cas comme un attribut et dans d'autres cas comme un type-entité, selon qu'il a ou non une existence propre. Par exemple, la marque d'une automobile peut être vue comme un attribut du type-entité *Véhicule* de la base de données d'une préfecture mais aussi comme un type-entité *Constructeur automobile* dans la base de données du Ministère de l'Industrie.

Lorsqu'on ne parvient pas à trouver d'identifiant pour un type-entité, il faut se demander s'il ne s'agit pas en fait d'un type-association. Si ce n'est pas le cas, un identifiant arbitraire numérique entier peut faire l'affaire.

Identification des type-associations – Identifiez les type-associations reliant les type-entités du modèle. Le cas échéant, leur affecter les attributs correspondant.

Il est parfois difficile de faire un choix entre un type-entité et un type-association. Par exemple, un mariage peut être considéré comme un type-association entre deux personnes ou comme un type-entité pour lequel on veut conserver un numéro, une date, un lieu, . . ., et que l'on souhaite manipuler en tant que tel.

Étudiez également les cardinalités des type-associations retenus. Lorsque toutes les pattes d'un type-association portent la cardinalité $1,1$, il faut se demander si ce type-association et les type-entités liés ne décrivent pas en fait un seul type-entité (cf. règle 2.29).

Vérification du modèle – Vérifiez que le modèle respecte bien les règles que nous avons énoncés et les définitions concernant la normalisation des type-entités et des type-associations. Le cas échéant, opérez les modifications nécessaires pour que le modèle soit bien formé.

Remarque : pour faciliter la lecture du schéma, il est assez courant de ne pas y faire figurer les attributs ou de ne conserver que ceux qui font partie des identifiants. Les attributs cachés doivent alors absolument être spécifiés dans un document à part.

2.6.2 Conseils divers

Concernant le choix des noms

Pour les type-entités, choisissez un nom commun décrivant le type-entité (ex : Étudiant, Enseignant, Matière). Certain préfèrent mettre le nom au pluriel (ex : Étudiants, Enseignants, Matières). Restez cependant cohérents, soit tous les noms de type-entité sont au pluriel, soit ils sont tous au singulier.

Pour les type-association, choisissez un verbe à l'infinitif, éventuellement à la forme passive ou accompagné d'un adverbe (ex : Enseigner, Avoir lieu dans).

Pour les attributs, utilisez un nom commun au singulier éventuellement accompagné du nom du type-entité ou du type-association dans lequel il se trouve (ex : nom de client, numéro d'article).

Concernant le choix des identifiants des type-entités

Évitez les identifiants composés de plusieurs attributs (comme, par exemple, un identifiant formé par les attributs *nom* et *prénom* d'un type-association *Personne*) car :

- ils dégradent les performances du SGBD,
- mais surtout l'unicité supposée par une telle démarche finit généralement, tôt ou tard, par être démentie !

Évitez les identifiants susceptibles de changer au cours du temps (comme la plaque d'immatriculation d'un véhicule).

Évitez les identifiants du type chaîne de caractère.

En fait, il est souvent préférable de choisir un identifiant arbitraire de type entier pour les type-entités. Cet identifiant deviendra une clé primaire dans le schéma relationnel et le SGBD l'incrémentera automatiquement lors de la création de nouvelles instances. L'inconvénient de cette pratique est qu'il devient possible de se retrouver avec deux instances du type-entités représentant le même objet mais avec deux numéros différents. Malgré cette inconvénient, cette politique de l'identifiant reste largement avantageuse dans la pratique et permet, en outre, de s'affranchir (en la satisfaisant automatiquement) de la deuxième forme normale (cf. section 2.5.4).

Bien distinguer les concepts de données et de traitements

La modélisation conceptuelle de données exclut la représentation des traitements futurs sur ces données. Toutefois, elle nécessite la connaissance de ces traitements pour prévoir les données élémentaires indispensables à ceux-ci. En conséquence, il existe une confusion fréquente entre les concepts de données et de traitements. Par exemple, la facturation est un traitement qui nécessite de connaître toutes les caractéristiques d'une commande. Par contre, la facturation ne se traduit ni par un type-entité, ni par un type-association dans le schéma entités-associations.

2.7 Travaux Dirigés – Modèle entités-associations (2epartie)

2.7.1 Mais qui a fait cette modélisation ?

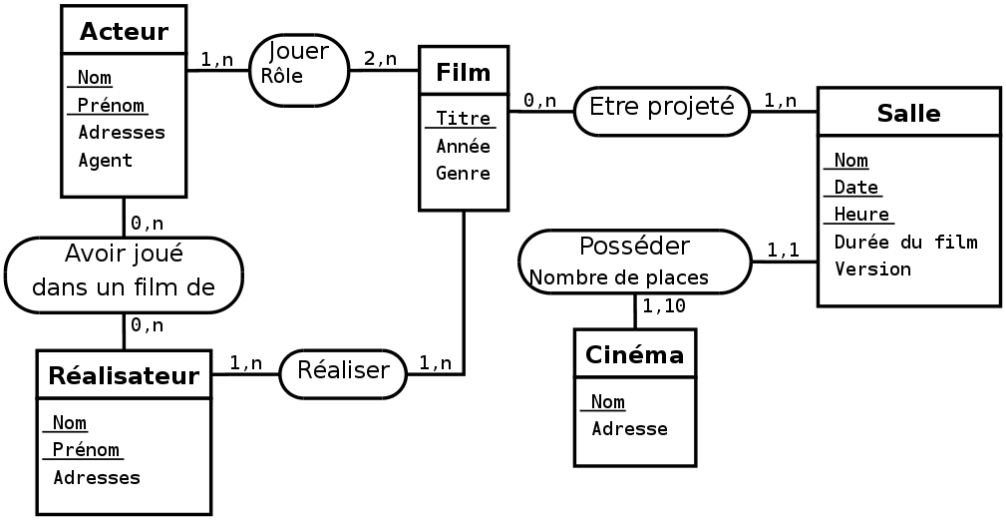


FIG. 2.38 – Ce modèle entités-associations n'est pas en bonne forme !

Le modèle entités-associations de la figure 2.38 pose de nombreux problèmes.

1. Identifiez les erreurs de modélisation et les incohérences dont souffre ce modèle. Précisez à chaque fois la règle ou la définition enfreinte et réfléchissez à la correction à apporter au modèle.
2. Proposez un modèle corrigé bien formé.

2.7.2 Cas d'une bibliothèque (2e partie)

Une petite bibliothèque souhaite informatiser la gestion de son fonds documentaire et de ses emprunts. Dans cette perspective, le bibliothécaire, qui n'est pas un informaticien, a rédigé le texte suivant :

> Grâce à cette informatisation, un abonné devra pouvoir retrouver un livre en connaissant son titre. Il doit aussi pouvoir connaître la liste des livres d'un auteur, la liste des auteurs d'un livre ainsi que son éditeur. Chaque livre est acheté en un ou plusieurs exemplaires. Attention, un livre est parfois édité plusieurs fois, éventuellement par des éditeurs différents. Pour s'abonner, une personne doit verser une caution et laisser ses coordonnées. Suivant le montant de sa caution, un abonné a le droit d'emprunter entre deux et dix ouvrages simultanément. Les prêts sont accordés pour une durée de quinze jours. La gestion des prêts doit permettre de connaître, à tout moment, la liste des livres détenus par un abonné, et inversement, de retrouver le nom des abonnés détenant un livre absent des rayons. La gestion du fonds documentaire doit permettre de connaître pour chaque exemplaire sa date d'achat, son état et s'il est disponible en rayon dans la bibliothèque.

15. Identifiez, dans le texte ci-dessus, les mots devant se concrétiser par des entités, des associations ou des attributs.
16. Proposez un modèle entités-associations **bien formé** permettant de modéliser la situation décrite ci-dessus.

2.7.3 Cas d'une entreprise de dépannage

Une entreprise de dépannage possède plusieurs services spécialisés regroupant chacun un certain nombre d'employés. Les employés ne travaillent que dans un service, ils ont une fonction dans l'entreprise, éventuellement un supérieur et des subalternes. Leur salaire dépend de leur fonction et de leur ancienneté au sein de l'entreprise. En plus du petit outillage courant, l'entreprise de dépannage dispose de gros matériels demandant une qualification particulière aux salariés susceptibles de l'utiliser. Tous les salariés ne sont pas qualifiés pour l'utilisation de tout le matériel. Ce matériel est référencé au niveau de l'entreprise. Un matériel particulièrement complexe est référencé comme un tout et, le cas échéant, par composants, les composant étant eux-mêmes parfois décomposables. Une intervention de dépannage se fait toujours à la demande d'un client et sous la direction d'un responsable. Une intervention de dépannage se décompose en un certain nombre d'actes de dépannage faisant intervenir un employé. Chaque acte de dépannage comporte un coût. Lorsqu'un employé participe à un acte de dépannage, la date de début et de fin de la participation de l'employé est notée.

Proposez un modèle entités-associations bien formé permettant de modéliser la situation décrite ci-dessus.

Chapitre 3

Bases de données relationnelles

3.1 Introduction au modèle relationnel

3.1.1 Présentation

Le modèle relationnel a déjà été introduit dans la section 1.1.2.

Dans ce modèle, les données sont représentées par des tables, sans préjuger de la façon dont les informations sont stockées dans la machine. Les tables constituent donc la *structure logique*[1] du modèle relationnel. Au niveau physique, le système est libre d'utiliser n'importe quelle technique de stockage (fichiers séquentiels, indexage, adressage dispersé, séries de pointeurs, compression, ...) dès lors qu'il est possible de relier ces structures à des tables au niveau logique. Les tables ne représentent donc qu'une abstraction de l'enregistrement physique des données en mémoire.

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

Les objectifs du modèle relationnel sont :
- proposer des schémas de données faciles à utiliser ;
- améliorer l'indépendance logique et physique (cf. section 1.2.2) ;
- mettre à la disposition des utilisateurs des langages de haut niveau ;
- optimiser les accès à la base de données ;
- améliorer l'intégrité et la confidentialité ;
- fournir une approche méthodologique dans la construction des schémas.

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

Au modèle relationnel est associée a la théorie de la normalisation des relations qui permet de se débarrasser des incohérences au moment de la conception d'une base de données relationnelle.

3.1.2 Éléments du modèle relationnel

Définition 3.1 **-attribut-** *Un attribut est un identificateur (un nom) décrivant une information stockée dans une base.*

Exemples d'attribut : l'âge d'une personne, le nom d'une personne, le numéro de sécurité sociale.

Définition 3.2 **-Domaine-** *Le domaine d'un attribut est l'ensemble, fini ou infini, de ses valeurs possibles.*

[1] Le terme *structure logique* englobe ici le niveau conceptuel et les niveaux externes d'ANSI/SPARC (cf. section 1.2.3) et correspond approximativement au niveau logique de Merise (cf. section 2.1.2).

Par exemple, l'attribut *numéro de sécurité sociale* a pour domaine l'ensemble des combinaisons de quinze chiffres et *nom* a pour domaine l'ensemble des combinaisons de lettres (une combinaison comme cette dernière est généralement appelée chaîne de caractères ou, plus simplement, chaîne).

Définition 3.3 -relation- *Une relation est un sous-ensemble du produit cartésien de n domaines d'attributs ($n > 0$).*

Une relation est représentée sous la forme d'un tableau à deux dimensions dans lequel les n attributs correspondent aux titres des n colonnes.

Définition 3.4 -schéma de relation- *Un schéma de relation précise le nom de la relation ainsi que la liste des attributs avec leurs domaines.*

Le tableau 3.1 montre un exemple de relation et précise son schéma.

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

TAB. 3.1 – Exemple de relation de schéma *Personne(N° sécu : Entier, Nom : Chaîne, Prénom : Chaîne)*

Définition 3.5 -degré- *Le degré d'une relation est son nombre d'attributs.*

Définition 3.6 -occurrence ou n-uplets ou tuples- *Une occurrence, ou n-uplets, ou tuples, est un élément de l'ensemble figuré par une relation. Autrement dit, une occurrence est une ligne du tableau qui représente la relation.*

Définition 3.7 -cardinalité- *La cardinalité d'une relation est son nombre d'occurrences.*

Définition 3.8 -clé candidate- *Une clé candidate d'une relation est un ensemble minimal des attributs de la relation dont les valeurs identifient à coup sûr une occurrence.*

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

Règle 3.9 *Toute relation a au moins une clé candidate et peut en avoir plusieurs.*

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

Définition 3.10 -clé primaire- *La clé primaire d'une relation est une de ses clés candidates. Pour signaler la clé primaire, ses attributs sont généralement soulignés.*

Définition 3.11 -clé étrangère- *Une clé étrangère dans une relation est formée d'un ou plusieurs attributs qui constituent une clé primaire dans une autre relation.*

Définition 3.12 -schéma relationnel- *Un schéma relationnel est constitué par l'ensemble des schémas de relation.*

Définition 3.13 -base de données relationnelle- *Une base de données relationnelle est constituée par l'ensemble des n-uplets des différentes relations du schéma relationnel.*

3.1.3 Passage du modèle entités-associations au modèle relationnel

Règles de passage

Pour traduire un schéma du modèle entités-associations vers le modèle relationnel, on peut appliquer les règles suivantes :

1. La normalisation devrait toujours être effectuée avant le passage au modèle relationnel (cf. section 2.5.4). Dans les faits, elle est parfois faite *a posteriori* (section 3.2), ce qui impose toujours une surcharge de travail importante.
2. Chaque type-entité donne naissance à une relation. Chaque attribut de ce type-entité devient un attribut de la relation. L'identifiant est conservé en tant que clé de la relation.
3. Chaque type-association dont aucune patte n'a pour cardinalité maximale 1 donne naissance à une relation. Chaque attribut de ce type-association devient un attribut de la relation. L'identifiant, s'il est précisé, est conservé en tant que clé de la relation, sinon cette clé est formée par la concaténation des identifiants des type-entités qui interviennent dans le type-association.
4. Un type-association dont au moins une patte a une cardinalité maximale à 1 (ce type-association devrait être binaire et n'a généralement pas d'attribut) ne devient pas une relation. Il décrit en effet une dépendance fonctionnelle (cf. section 3.2). La relation correspondant au type-entité dont la patte vers le type-association a une cardinalité maximale valant 1, se voit simplement ajouter comme attribut (et donc comme clé étrangère) l'identifiant de l'autre type-entité.

Cas particulier d'un type-assocuation du type 1 **vers** 1

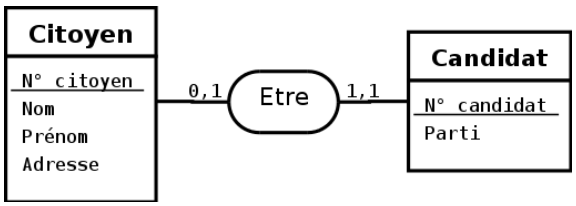


FIG. 3.1 – Reprise de l'exemple de la figure 2.30 d'un type-association *Etre* où toutes les cardinalités maximales sont de 1.

Dans l'exemple de la figure 3.1 toutes les cardinalités maximales du type-association *Etre* sont de 1. L'application des règles de passage du modèle entités-associations au modèle relationnel énoncées ci-dessus nous donnerait :

– Citoyen(Num-Citoyen, *Num-Candidat*, Nom, Prénom, Adresse)
– Candidat(Num-Candidat), *Num-Citoyen*, Parti)

L'attribut *Num-Candidat* dans la relation *Citoyen* est une clé étrangère de la relation *Candidat*. L'attribut *Num-Citoyen* dans la relation *Candidat* est une clé étrangère de la relation *Citoyen*.

Le type-association *Etre* étant du type 1 vers 1, il est entièrement matérialisé dans la relation *Candidat* par l'attribut *Num-Citoyen*. Il est donc inutile de la rematérialiser dans la relation *Citoyen*. L'attribut *Num-Candidat* dans la relation *Citoyen* doit donc être supprimé. D'autre part, dans la relation *Candidat*, l'attribut *Num-Citoyen*, en plus d'être une clé étrangère, constitue une clé candidate. On peut donc se passer de la clé *Num-Candidat*.

Le schéma relationnel adéquat correspondant au modèle entités-associations de la figure 3.1 devient donc :

– Citoyen(Num-Citoyen, Nom, Prénom, Adresse)
– Candidat(Num-Citoyen, Parti)

où *Num-Citoyen*, en plus d'être la clé de la relation *Candidat*, est une clé étrangère de la relation *Citoyen*.

Cas particulier d'un type-entité sans attribut autre que sa clé

Lorsqu'un type-entité ne possède pas d'attribut en dehors de sa clé, il ne faut pas nécessairement en faire une relation.

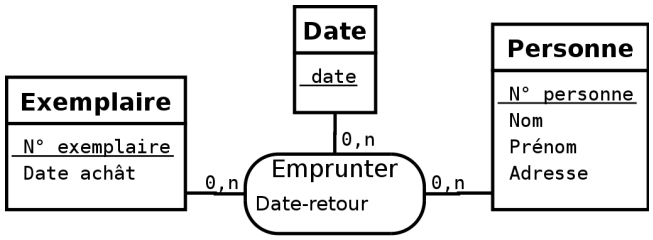


FIG. 3.2 – Ici, le type-entité *Date* ne doit pas se matérialiser par une relation.

Par exemple, le type-entité *Date* de la figure 3.2 ne doit pas se traduire par une relation. Le schéma relationnel adéquat correspondant au modèle entités-associations de la figure 3.2 est donc :

- Exemplaire(Num-Exemplaire, date-achat)
- Personne(Num-Personne, nom, prénom, adresse)
- Emprunter(Num-Exemplaire, Num-Personne, Date, date-retour)

Exemple complet

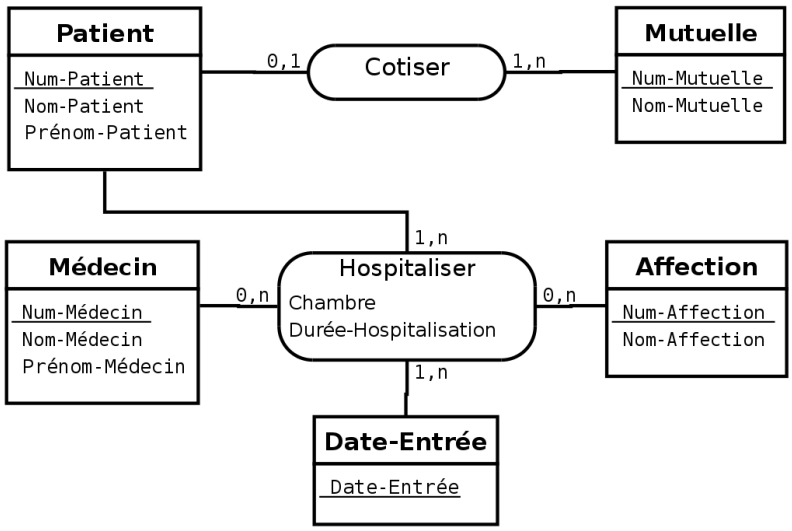


FIG. 3.3 – Exemple très simplifié de modélisation entités-associations

Comme exemple d'application, voici les relations déduites du schéma entités-associations de la figure 3.3 :

- Patient(Num-Patient, Nom-Patient, Num-Mutuelle)
- Mutuelle(Num-Mutuelle, Nom-Mutuelle)
- Médecin(Num-Médecin, Nom-Médecin, Prénom-Médecin)
- Affection(Num-Affection, Nom-Affection)
- Hospitaliser(Num-Patient, Num-Affection, Num-Médecin, Date-Entrée, Chambre, Durée-Hospitalisation)

3.2 Normalisation

3.2.1 Introduction

Les formes normales sont différents stades de qualité qui permettent d'éviter la redondance dans les bases de données relationnelles afin d'éviter ou de limiter : les pertes de données, les incohérences au sein des données, l'effondrement des performances des traitements.

Le processus de normalisation consiste à remplacer une relation donnée par certaines *projections* afin que la *jointure* de ces projections permette de retrouver la relation initiale. En d'autres termes, le processus est réversible (*i.e.* sans perte d'information). Les notions de projection et de jointure seront respectivement définies dans les sections 3.4.3 et 3.4.8.

Il existe une hiérarchie dans les règles de normalisation : une relation en 5^e forme normale est forcément en 4^e forme normale, une relation en 4^e forme normale est forcément en forme normale de Boyce-Codd, etc. Il existe des méthodes systématiques pour normaliser une relation dans chacune des formes normales. Ces algorithmes de décomposition, associés à chacune des formes normales, sortent du cadre de ce cours et ne seront pas abordés.

La normalisation peut être effectuée, et c'est préférable, pendant la phase de conception sur le modèle entités-associations (cf. section 2.5.4). Ce qui a été dit et les exemples qui ont été donnés dans cette section restent transposables au modèle relationnel. Dans le cas où la normalisation est faite en amont, lors de la conception, il n'est pas nécessaire de la recommencer sur le modèle relationnel. On peut tout de même vérifier que les relations obtenues par le passage du modèle entités-associations au modèle relationnel sont toujours en forme normale, mais, sauf erreur, il ne devrait pas y avoir de problème. Il en va tout autrement lorsque l'on ne connaît pas bien, ou maîtrise pas bien, l'origine d'un modèle relationnel. Dans ce cas, vérifier la normalisation des relations, et, le cas échéant, les normaliser, est une phase primordiale. C'est également le cas lorsque le modèle relationnel est le modèle de conception (*i.e.* on ne passe pas par un modèle entités-associations).

Contrairement à ce que nous avions fait dans la section 2.5.4 dans le cadre du modèle entités-associations, nous abordons ici la normalisation en nous appuyant sur les notions de *dépendance fonctionnelle*, *dépendance multivaluée* et *dépendance de jointure*. Il est important de prendre conscience que la dépendance fonctionnelle, la dépendance multivaluée et la dépendance de jointure sont des notions sémantiques. Elles tirent leurs origines dans les contraintes du monde réel. Comme ces contraintes participent à la sémantique de la situation, elles doivent avoir une manifestation dans la base de données. Les dépendances doivent donc être spécifiées dans la définition de la base de données afin que le SGBD puisse les appliquer. Les concepts de normalisation fournissent en fait un moyen indirect de déclarer ces dépendances. Autrement dit, la normalisation d'une base de données est une manifestation observable des dépendances observées dans le monde réel. La dépendance fonctionnelle permet de définir les premières formes normales jusqu'à la forme normale de Boyce-Codd (1FN, 2FN, 3FN et BCNF). La dépendance multivaluée permet de définir la quatrième forme normale (4FN) et la dépendance de jointure la cinquième forme normale (5FN).

3.2.2 Dépendance fonctionnelle (DF)

Définition 3.14 -dépendance fonctionnelle (DF)- *Soit $R(A_1, A_2, \ldots A_n)$ un schéma de relation, et X et Y des sous-ensembles de $A_1, A_2, \ldots A_n$. On dit que X **détermine** Y ou que Y **dépend fonctionnellement** de X si, et seulement si, des valeurs identiques de X impliquent des valeurs identiques de Y. On le note : $X \rightarrow Y$.*

Autrement dit, il existe une dépendance fonctionnelle entre un ensemble d'attributs X et un ensemble d'attributs Y, que l'on note $X \rightarrow Y$, si connaissant une occurrence de X on ne peut lui associer qu'une seule occurrence de Y.

Il est essentiel de noter qu'une dépendance fonctionnelle est une assertion sur toutes les valeurs possibles et non sur les valeurs actuelles : elle caractérise une intention et non une extension de la relation.

Définition 3.15 -dépendance fonctionnelle élémentaire- *Une dépendance fonctionnelle élémentaire est une dépendance fonctionnelle de la forme $X \rightarrow A$, où A est un attribut unique n'appartenant pas à X et où il n'existe pas X' inclus au sens strict dans X* (i.e. $X' \subset X$) *tel que $X' \rightarrow A$.*

Autrement dit, une dépendance fonctionnelle est élémentaire si la cible est un attribut unique et si la source ne comporte pas d'attributs superflus. La question sur l'élémentarité d'une dépendance fonctionnelle ne doit donc se poser que lorsque la partie gauche de la dépendance fonctionnelle comporte plusieurs attributs.

Définition 3.16 -dépendance fonctionnelle directe- *Une dépendance fonctionnelle $X \rightarrow A$ est une dépendance fonctionnelle directe s'il n'existe aucun attribut B tel que l'on puisse avoir $X \rightarrow B$ et $B \rightarrow A$.*

En d'autres termes, cela signifie que la dépendance entre X et A ne peut pas être obtenue par transitivité.

3.2.3 Première et deuxième forme normale

Première forme normale

Définition 3.17 -première forme normale (1FN)- *Une relation est en première forme normale si, et seulement si, tout attribut contient une valeur atomique (non multiples, non composées).*

Par exemple, le pseudo schéma de relation *Personne(num-personne, nom, prénom, rue-et-ville, prénoms-enfants)* n'est pas en première forme normale. Il faut le décomposer en :
- Personne(num-personne, nom, prénom, rue, ville)
- Prénoms-enfants(num-personne, num-prénom)
- Prénoms(num-prénom, prénom)

Remarques sur la première forme normale

La première forme normale impose que chaque ligne d'une relation ait une seule valeur pour chaque colonne (*i.e.* attribut), ce qui est justement la définition d'une table. Donc, une table est nécessairement en première forme normale au sens du modèle relationnel.

Cependant, il faut noter que le modèle relationnel peut être étendu de manière à permettre des colonnes à valeur complexe. On parle alors de *modèle relationnel étendu* (NF^2 pour *Non First Normal Form* en anglais).

Deuxième forme normale

Définition 3.18 -deuxième forme normale (2FN)- *Une relation est en deuxième forme normale si, et seulement si, elle est en première forme normale et si toutes les dépendances fonctionnelles entre la clé et les autres attributs sont élémentaires.*

Autrement dit, une relation est en deuxième forme normale si, et seulement si, elle est en première forme normale et si tout attribut n'appartenant pas à la clé ne dépend pas que d'une partie de la clé.

Une relation peut être en deuxième forme normale par rapport à une de ses clés candidates et ne pas l'être par rapport à une autre. Une relation avec une clé primaire réduite à un seul attribut est, par définition, forcément en deuxième forme normale.

Soit, par exemple, le schéma de relation suivant : CommandeLivre(Num-Commande, Num-Client, Titre, Auteur, Quantité, Prix). Cette relation indique qu'un client (identifié par *Num-Client*) a passé une commande (identifiée par *Num-Commande*) de livre. Elle est bien en première forme normale. Par contre, les attributs *Titre*, *Auteur*, *Quantité* et *Prix* ne dépendent que de *Num-Commande*, et pas de *Num-Client*. Cette relation n'est donc pas en deuxième forme normale. Une solution simple pour la normaliser est de la remplacer par : CommandeLivre(Num-Commande, Num-Client, Titre, Auteur, Quantité, Prix).

3.2.4 Troisième forme normale

Définition 3.19 -troisième forme normale (3FN)- *Une relation est en troisième forme normale si, et seulement si, elle est en deuxième forme normale et si toutes les dépendances fonctionnelles entre la clé et les autres attributs sont élémentaires et directes.*

Autrement dit, une relation est en troisième forme normale si, et seulement si, elle est en deuxième forme normale et si tout attribut n'appartenant pas à la clef ne dépend pas d'un attribut non-clé.

Une relation peut être en troisième forme normale par rapport à une de ses clés candidates et ne pas l'être par rapport à une autre. Une relation en deuxième forme normale avec au plus un attribut qui n'appartient pas à la clé primaire est, par définition, forcément en troisième forme normale.

Soit, par exemple, le schéma de relation suivant : CommandeLivre(Num-Commande, Num-Client, Titre, Auteur, Quantité, Prix). Comme nous l'avons vu plus haut, cette relation est bien en deuxième forme normale. Par contre, les attributs *Auteur* et *Prix* dépendent de l'attribut *Titre*. La relation n'est donc pas en troisième forme normale. Pour la normaliser, il faut la décomposer de la manière suivante :

- CommandeLivre(Num-Commande, Num-Client, Num-Livre, Quantité)
- Livre(Num-Livre, Titre, Auteur, Prix)

Remarques importantes

Soit les schémas de relation suivant :

- Ville(Code-Postal, Nom, Population)
- Personne(Nom, Prénom, Téléphone)

Dans ces relations, on suppose les dépendances fonctionnelles directes suivante :

- Code-Postal → Nom
- Code-Postal → Population
- Nom, Prénom → Téléphone

Dans la section 2.6.2, nous avons dit qu'il est souvent préférable de choisir un identifiant arbitraire de type entier. Cette pratique semble aller à l'encontre de la troisième forme normale. Par exemple, la relation Ville(num-ville, Nom, Code-Postal, Population) n'est pas en troisième forme normale si l'on suppose que les attributs *Nom* et *Population* dépendent toujours de l'attribut *Code-Postal*. Cependant, comme nous l'avons dit dans l'introduction, une dépendance fonctionnelle est la manifestation d'une notion sémantique, pas d'une notion formelle ou absolue. Dans le cas du code postal, nous avons déjà expliqué (cf. note page 33) qu'il n'existe pas de relation systématique entre le code postal et le code du département ou la commune. Ainsi, il n'y a pas de dépendance fonctionnelle entre les attributs *Nom* et *Population* et l'attribut *Code-Postal*. La relation Ville(num-ville, Nom, Code-Postal, Population) est donc bien en troisième forme normale (en France, plusieurs villes portent le même nom). La notion de dépendance fonctionnelle est donc une question d'interprétation faisant appel à la connaissance du système modélisé et au bon sens.

Il en va de même avec le schéma de relation Livre(Num-Livre, Titre, Auteur, Prix). Nous avons ici introduit un identifiant numérique arbitraire *Num-Livre* car l'identifiant naturel *Titre*, qui est une chaîne de caractères complexe, de taille non bornée et au format libre, ne constitue pas un bon identifiant dans la pratique. Pour justifier la troisième forme normale de cette relation, on peut imaginer que plusieurs livres peuvent porter le même titre.

Il faut enfin noter que la normalisation n'est pas une fin en soit et qu'elle ne doit pas nécessairement être systématiquement appliquée (nous y reviendrons section 3.2.7).

3.2.5 Forme normale de BOYCE-CODD

Définition 3.20 -forme normale de BOYCE-CODD (BCNF)- *Une relation est en forme normale de BOYCE-CODD (BCNF) si, et seulement si, elle est en troisième forme normale et si les seules dépendances fonctionnelles élémentaires sont celles dans lesquelles une clé détermine un attribut.*

Cette forme normale permet de renforcer certaines lacunes de la troisième forme normale.

Soit, par exemple, le schéma relationnel décrivant l'enseignement d'une matière donnée à une classe par un enseignant :

- Matière(nom-matière)
- Classe(num-classe)
- Enseignant(nom-enseignant)
- Enseignement(nom-enseignant, num-classe, nom-matière)

Supposons, de plus, qu'une matière n'est enseignée qu'une seule fois dans une classe et que par un seul enseignant, et qu'un enseignant n'enseigne qu'une seule matière. Chacune des relations respecte bien la troisième forme normale. Cependant, dans la relation *Enseignement*, nous avons les dépendances fonctionnelles élémentaires suivantes :

1. nom-matière, num-classe → nom-enseignant
2. nom-enseignant → nom-matière

Il existe donc des dépendances fonctionnelles élémentaires dont la source n'est pas la clé de la relation.

nom-enseignant	**num-classe**	**nom-matière**
George	5	Physique
George	6	Physique
George	7	Physique
George	8	Physique
Michael	5	Mathématiques
Michael	6	Mathématiques
Michael	7	Mathématiques
Michael	8	Mathématiques

TAB. 3.2 – Exemple de relation présentant une redondance due au non respect de la forme normale de BOYCE-CODD.

Le non respect de la forme normale de BOYCE-CODD entraîne une redondance illustrée par la table 3.2 : pour chaque *nom-enseignant* identifiant un enseignant, il faut répéter le *nom-matière* identifiant la matière qu'il enseigne.

Pour normaliser la relation *Enseignement*, il faut la décomposer pour aboutir au schéma relationnel suivant :

- Matière(nom-matière)
- Classe(num-classe)
- Enseignant(nom-enseignant, nom-matière)
- Enseigner(nom-enseignant, num-classe)

Dans la pratique, la plupart des problèmes de conception peuvent être résolus en appliquant les concepts de troisième forme normale et de forme normale de BOYCE-CODD. Les quatrième et cinquième formes normales traitent encore d'autres cas de redondance, mais qui ne sont pas expliqués par des dépendances fonctionnelles.

3.2.6 Pour aller plus loin que le cours : quatrième et cinquième forme normale

Dépendance multivaluée (DM)

Définition 3.21 -dépendance multivaluée (DM)- *Soit $R(A_1, A_2, \ldots A_n)$ un schéma de relation contenant n propriétés, soit X, Y et Z des sous-ensembles de $A_1, A_2, \ldots A_n$ et soit X_i, Y_i et Z_i des instances de ces sous-ensembles* (i.e. *une affectation de valeur à chacune des propriétés de ces sous-ensembles). Il existe une dépendance multivaluée (DM) entre les ensembles de propriétés X, Y lorsque :*

$$(X_1, Y_1, Z_1) \in R \text{ et } (X_1, Y_2, Z_2) \in R \Rightarrow (X_1, Y_1, Z_2) \in R \text{ et } (X_1, Y_2, Z_1) \in R$$

On la note $X \twoheadrightarrow Y$, ce qui se lit X multidétermine Y.

Remarque : $X \twoheadrightarrow Y \Rightarrow X \twoheadrightarrow A_i - (X \cup Y)$.

Comme illustration, supposons une situation où un employé d'un garage est qualifié pour effectuer un certain type d'intervention sur certaines marques de voiture. Cette situation est modélisée par le schéma relationnel suivant :

- Employé(Nom-Employé)

- Intervention(Type-Intervetion)
- Constructeur(Marque)
- Intervenir(Nom-Employé, Type-Intervetion, Marque)

Supposons maintenant qu'un employé qui effectue un ensemble de types d'interventions pour un ensemble de marques de voiture, est capable d'effectuer chacun de ces types d'interventions sur chacune de ces marques de voitures. Dans ce cas, il existe des dépendances multivaluées dans la relation *Intervenir* : *Nom-Employé* $\twoheadrightarrow$ *Type-Intervetion* et *Nom-Employé* $\twoheadrightarrow$ *Marque*.

Quatrième forme normale

Définition 3.22 -quatrième forme normale (4FN)- *Une relation est en quatrième forme normale (4FN) si, et seulement si, elle est en forme normale de BOYCE-CODD et si elle ne possède pas de dépendance multivaluée ou si, $X \twoheadrightarrow Y$ étant la dépendance multivaluée, il existe une propriété A telle que $X \rightarrow A$.*

Nom-Employé	Type-Intervetion	Marque
Tussier	Dépannage	Peugeot
Tussier	Dépannage	Citroën
Martin	Électricité	Citroën
Martin	Électricité	Renault
Martin	Mécanique	Citroën
Martin	Mécanique	Renault
Piquard	Carrosserie	Fiat
Piquard	Carrosserie	Ford
Piquard	Alarme	Fiat
Piquard	Alarme	Ford
Piquard	Électricité	Fiat
Piquard	Électricité	Ford

TAB. 3.3 – Exemple de relation n'étant pas en quatrième forme normale.

Dans la section précédente, nous avons présenté un schéma relationnel qui n'était pas en quatrième forme normale en raison du schéma de relation *Intervenir*. La table 3.3 propose un exemple de relation correspondant à ce schéma de relation. Cette table permet d'observer le phénomène de redondance consécutif au fait que cette table n'est pas en quatrième forme normale. Dans cette table, le nombre de lignes commençant par un nom d'employé donné doit être égale au nombre d'interventions que cet employé peut faire multiplié par le nombre de marques sur lesquelles il peut travailler. Imaginons que l'employé *Piquard* puisse maintenant travailler sur des voitures de la marque *Citroën* (on désire ajouter **une** information dans la base), il faudra alors ajouter trois lignes à la table : une pour chaque type d'intervention (*Carrosserie*, *Alarme* et *Électricité*).

Pour normaliser la relation *Intervenir*, il faut la décomposer pour aboutir au schéma relationnel suivant :

- Employé(Nom-Employé)
- Intervention(Type-Intervetion)
- Constructeur(Marque)
- Etre-capable-de(Nom-Employé, Type-Intervetion)
- Etre-capable-d'intervenir-sur(Nom-Employé, Marque)

Dépendance de jointure (DJ)

Jusqu'ici, nous avons pu résoudre une redondance dans une relation en la remplaçant par deux de ses projections. Il existe cependant des relations qui ne peuvent pas être décomposées sans perte

d'information en deux projections, mais qui peuvent l'être en trois ou plus (ces cas sont assez rares en pratique). C'est ce que permet la normalisation en cinquième forme normale.

Les dépendances de jointures font appel à des notions (*projection* et *jointure*) qui seront définies plus loin (cf. section 3.4).

Définition 3.23 -dépendance de jointure (DJ)- *Soient $X_1, X_2, \ldots, X_n$ des sous-ensembles d'un schéma de relation R. Il y a une dépendance de jointure, notée $*\{X_1, X_2, \ldots, X_n\}$ dans la relation R, si :*

$$R = \Pi_{(X_1)}R \bowtie \Pi_{(X_2)}R \bowtie \ldots \bowtie \Pi_{(X_n)}R$$

Définition 3.24 -dépendance de jointure triviale- *Une dépendance de jointure est triviale si une des parties, X_i, est l'ensemble de toutes les attributs de R.*

Cinquième forme normale (5FN)

Définition 3.25 -cinquième forme normale (5FN)- *Une relation R est en cinquième forme normale (5FN) si, pour toute dépendance de jointure non triviale $*\{X1, X2, \ldots, Xn\}$ dans R, chacun des Xi contient une clé candidate de R.*

En d'autres termes, les seules décompositions qui préservent le contenu sont celles où chacune des tables de la décomposition contient une clé candidate de la table. Il est donc superflu de décomposer de ce point de vue.

Cette forme normale est finale vis-à-vis de la projection et de la jointure : elle garantie qu'une relation en cinquième forme normale ne contient aucune anomalie pouvant être supprimée en effectuant des projections (*i.e.* des décompositions).

Relation Fournisseur		
Num-Fournisseur	**Num-Article**	**Num-Organisme**
$f1$	$a2$	$o1$
$f1$	$a1$	$o2$
$f2$	$a1$	$o1$
$f1$	$a1$	$o1$

TAB. 3.4 – Exemple de relation n'étant pas en cinquième forme normale.

Prenons, comme illustration[2], la relation *Fournisseur* (table 3.4) qui décrit les fournisseurs des organismes de la fonction publique.

La fonction publique a des règles très particulières concernant les fournisseurs pour réduire le potentiel de conflit d'intérêt. Un fournisseur fournit un certain nombre d'articles (par exemple $f1$ fournit $a1$ et $a2$). Le même article peut être fourni par plusieurs fournisseurs (par exemple $a1$ est fourni par $f1$ et $f2$). Un fournisseur peut être attitré à plusieurs organismes (par exemple $f1$ est attitré à $o1$ et $o2$). Un organisme peut avoir plusieurs fournisseurs (par exemple $o1$ est servi par $f1$ et $f2$). Un organisme peut utiliser plusieurs articles (c'est-à-dire que $o1$ utilise $a1$ et $a2$) et un article peut être utilisé par plusieurs organismes (c'est-à-dire que $a1$ est utilisé par $o1$ et $o2$). La règle de la fonction publique est la suivante :

- si un fournisseur fournit un certain article (comme $f1$ fournit $a1$),
- le fournisseur est attitré à l'organisme (comme $f1$ est attitré à $o1$), et
- l'organisme utilise un article (comme $o1$ utilise $a1$),
- alors nécessairement, le fournisseur fournit l'article à l'organisme ($f1$ fournit $a1$ à $o1$).

Le dernier fait est déductible des trois autres.

Cette table contient de la redondance de données parce que certains faits sont répétés. Par exemple, le fait que $f1$ fournit $a1$ est répété à deux reprises, une fois parce qu'il fournit $a1$ à $o1$ et une autre fois parce qu'il fournit $a1$ à $o2$. Le fait que $f1$ est attitré à $o1$ est aussi répété à deux reprises. Il en est de même pour $o1$ qui utilise $a1$.

[2] Exemple tiré de (Godin, 2000a).

La relation *Fournisseur* souffre d'une dépendance de jointure :
**{(Num-Fournisseur, Num-Article), (Num-Fournisseur, Num-Organisme), (Num-Article, Num-Organisme)}*.
Pour résoudre le problème de redondance, il faut décomposer la relation en trois (cf. table 3.5).

Relation FournisseurArticle	
Num-Fournisseur	Num-Article
$f1$	$a2$
$f1$	$a1$
$f2$	$a1$

Relation FournisseurOrganisme	
Num-Fournisseur	Num-Organisme
$f1$	$o1$
$f1$	$o2$
$f2$	$o1$

Relation ArticleOrganisme	
Num-Article	Num-Organisme
$a2$	$o1$
$a1$	$o2$
$a1$	$o1$

TAB. 3.5 – Décomposition de la relation *Fournisseur* (table 3.4) pour obtenir des relations en cinquième forme normale.

Il est important de se convaincre qu'aucune décomposition binaire de cette relation ne préserve le contenu de la relation initiale. Pour cela, il suffit de tenter de joindre deux tables parmi les trois précédentes. Aucune de ces jointures, ne produit la relation *Fournisseur*.

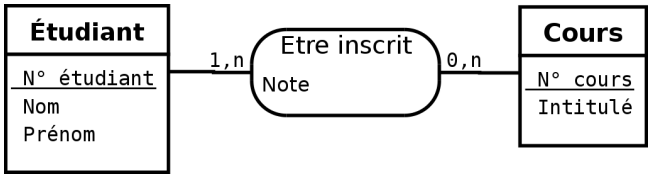
3.2.7 Remarques au sujet de la normalisation

Il existe d'autres formes normales comme la forme normale domaine-clé (FNDC), la forme normale de restriction-union ou la sixième forme normale (6NF).

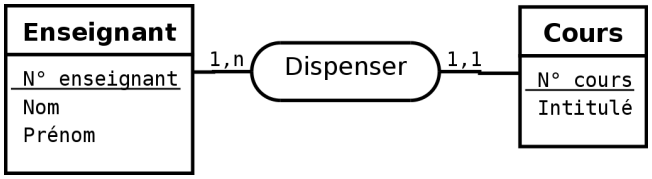
Bien que l'objectif de la normalisation soit d'amener le concepteur à obtenir des relations en forme normale *finale* (*i.e.* en cinquième forme normale), cet objectif ne doit pas être interprété comme une loi. Il peut exister, très occasionnellement, de bonnes raisons pour passer outre les principes de la normalisation. De plus, un schéma en cinquième forme normale n'est pas nécessairement un schéma pleinement satisfaisant. D'autres facteurs sont à considérer dans le processus de conception d'une base de données et l'expérience et l'intuition jouent un rôle important.

3.3 Travaux Dirigés – Modèle relationnel

3.3.1 Passage du modèle entités-associations au modèle relationnel



1. Établissez un schéma relationnel à partir du petit diagramme entités-associations ci-dessus.
2. Quelles sont les clés primaires et les clés étrangères de chaque relation ?
3. Proposez un petit exemple de base de données relationnelle correspondant au schéma relationnel établi précédemment.



4. Combien de schémas de relation doit contenir la traduction en schéma relationnel du petit diagramme entités associations ci-dessus ?
5. Établissez un schéma relationnel à partir du petit diagramme entités associations ci-dessus sans tenir compte de la spécificité de la cardinalité 1-1.
6. Proposez un petit exemple de base de données relationnelle correspondant au schéma relationnel établi précédemment.
7. Expliquez pourquoi deux des relations doivent être fusionnées.
8. Donnez le schéma relationnel correct.
9. Quelles sont les clés primaires et les clés étrangères de chaque relation ?
10. A partir du MCD de la figure 3.4, établir le schéma relationnel.

3.3.2 Normalisation

La pièce

Le schéma de relation *Pièce* permet de décrire des pièces employées dans un atelier de montage :

Pièce (N°pièce, prix-unit, TVA, libellé, catégorie)

Supposons les dépendances fonctionnelles suivantes :
- N°pièce → prix-unit
- N°pièce → TVA
- N°pièce → libellé
- N°pièce → catégorie
- catégorie → TVA

11. Proposez un identifiant pour ce schéma de relation.
12. Normalisez ce schéma de relation jusqu'à la forme normale de Boyce Codd.

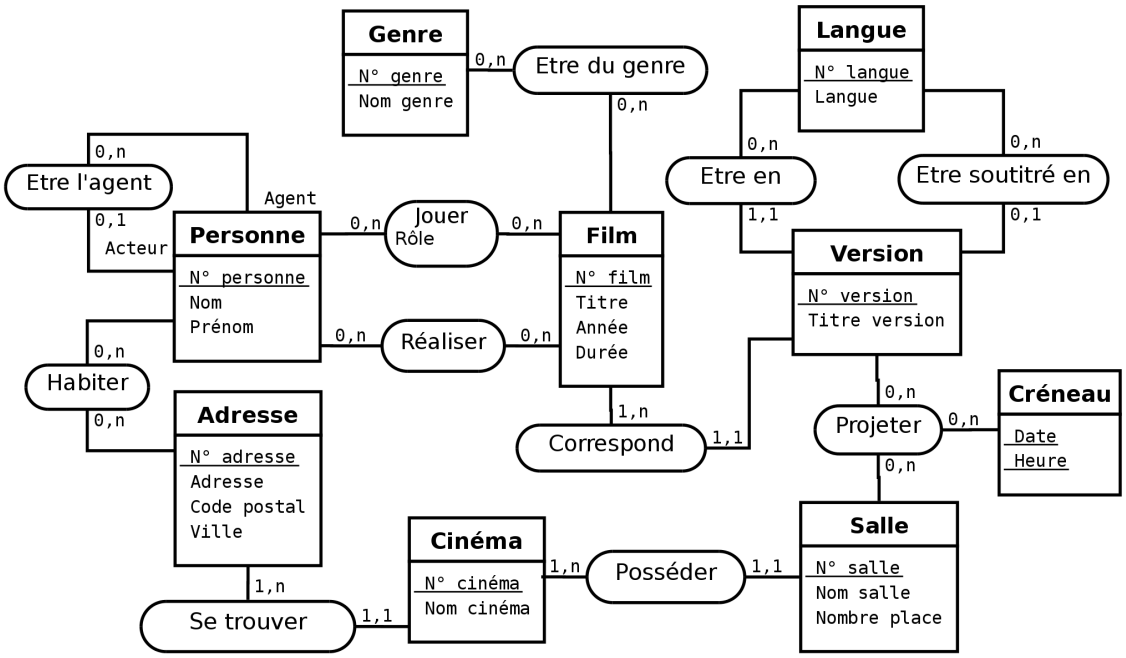


FIG. 3.4 – Exemple de MCD répondant à la question 2.7.1 du TD 2.7

La prime

Le schéma de relation *Prime* donne la liste des primes attribuées au personnel technique en fonction des machines sur lesquelles il travaille :

Prime (N°machine, atelier, N°technicien, montant-prime, nom-technicien)

Supposons les dépendances fonctionnelles suivantes :
- N°machine → atelier
- N°technicien → nom-technicien
- (N°machine, N°technicien) → montant-prime

13. Proposez un identifiant pour ce schéma de relation.
14. Normalisez ce schéma de relation jusqu'à la forme normale de Boyce Codd.

L'école

Soit la relation *Enseignement* qui précise qu'un étudiant a un certain enseignant dans une certaine matière :

Enseignement (nom-étudiant, prénom-étudiant, matière, volume-horaire-matière, nom-enseignant, prénom-enseignant, salaire-enseignant)

15. Identifiez les dépendances fonctionnelles de ce schéma de relation.
16. Normalisez ce schéma de relation jusqu'en troisième forme normale.

Supposons maintenant que les contraintes suivantes s'appliquent :
- Chaque étudiant n'a qu'un enseignant par matière.
- Un enseignant n'enseigne qu'une seule matière, mais une matière peut très bien être enseignée par plusieurs professeurs.

17. En tenant compte de ces nouvelles contraintes, identifiez les dépendances fonctionnelles du schéma de relationnel en troisième forme normale que vous avez obtenu.
18. L'un des schémas de relation n'est pas en forme normale de Boyce Codd, lequel ?
19. A l'aide d'un exemple de relation, illustrez la redondance induite par ce schéma de relation.
20. Normalisez ce schéma de relation en forme normale de Boyce Codd.
21. Cette décomposition résout le problème de redondance, mais n'introduit-elle pas un autre problème ?

3.4 Algèbre relationnelle

3.4.1 Introduction

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

On peut distinguer trois familles d'opérateurs relationnels :

Les opérateurs unaires (Sélection, Projection) : ce sont les opérateurs les plus simples, ils permettent de produire une nouvelle table à partir d'une autre table.

Les opérateurs binaires ensemblistes (Union, Intersection Différence) : ces opérateurs permettent de produire une nouvelle relation à partir de deux relations de même degré et de même domaine.

Les opérateurs binaires ou n-aires (Produit cartésien, Jointure, Division) : ils permettent de produire une nouvelle table à partir de deux ou plusieurs autres tables.

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

3.4.2 Sélection

Définition 3.26 -sélection- *La sélection (parfois appelée restriction) génère une relation regroupant exclusivement toutes les occurrences de la relation R qui satisfont l'expression logique E, on la note* $\sigma_{(E)}R$.

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

$$\text{relation} \times \text{expression logique} \longrightarrow \text{relation}$$

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

Le tableau 3.7 montre un exemple de sélection.

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

TAB. 3.6 – Exemple de relation *Personne*

Numéro	Nom	Prénom
5	Durand	Caroline
12	Dupont	Lisa

TAB. 3.7 – Exemple de sélection sur la relation *Personne* du tableau 3.6 : $\sigma_{(Numéro \geq 5)}Personne$

3.4.3 Projection

Définition 3.27 -projection- *La projection consiste à supprimer les attributs autres que* $A_1, \ldots A_n$ *d'une relation et à éliminer les n-uplets en double apparaissant dans la nouvelle relation ; on la note* $\Pi_{(A_1, \ldots A_n)}R$.

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

$$\text{relation} \times \text{liste d'attributs} \longrightarrow \text{relation}$$

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

Le tableau 3.8 montre un exemple de sélection.

Nom
Durand
Germain
Dupont

TAB. 3.8 – Exemple de projection sur la relation *Personne* du tableau 3.6 : $\Pi_{Nom}Personne$

3.4.4 Union

Définition 3.28 **-union-** *L'union est une opération portant sur deux relations R_1 et R_2 ayant le même schéma et construisant une troisième relation constituée des n-uplets appartenant à chacune des deux relations R_1 et R_2 sans doublon, on la note $R_1 \cup R_2$.*

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

$$\text{relation} \times \text{relation} \longrightarrow \text{relation}$$

Comme nous l'avons déjà dit, R_1 et R_2 doivent avoir les mêmes attributs et si une même occurrence existe dans R_1 et R_2, elle n'apparaît qu'une seule fois dans le résultat de l'union. Le résultat de l'union est une nouvelle relation qui a les mêmes attributs que R_1 et R_2. Si R_1 et R_2 sont vides, la relation qui résulte de l'union est vide. Si R_1 (respectivement R_2) est vide, la relation qui résulte de l'union est identique à R_2 (respectivement R_1).

Le tableau 3.9 montre un exemple d'union.

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

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

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

TAB. 3.9 – Exemple d'union : $R = R_1 \cup R_2$

3.4.5 Intersection

Définition 3.29 **-intersection-** *L'intersection est une opération portant sur deux relations R_1 et R_2 ayant le même schéma et construisant une troisième relation dont les n-uplets sont constitués de ceux appartenant aux deux relations, on la note $R_1 \cap R_2$.*

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

$$\text{relation} \times \text{relation} \longrightarrow \text{relation}$$

Comme nous l'avons déjà dit, R_1 et R_2 doivent avoir les mêmes attributs. Le résultat de l'intersection est une nouvelle relation qui a les mêmes attributs que R_1 et R_2. Si R_1 ou R_2 ou les deux sont vides, la relation qui résulte de l'intersection est vide.

Le tableau 3.10 montre un exemple d'intersection.

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

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

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

TAB. 3.10 – Exemple d'intersection : $R = R_1 \cap R_2$

3.4.6 Différence

Définition 3.30 -différence- *La différence est une opération portant sur deux relations R_1 et R_2 ayant le même schéma et construisant une troisième relation dont les n-uplets sont constitués de ceux ne se trouvant que dans la relation R_1 ; on la note $R_1 - R_2$.*

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

$$\text{relation} \times \text{relation} \longrightarrow \text{relation}$$

Comme nous l'avons déjà dit, R_1 et R_2 doivent avoir les mêmes attributs. Le résultat de la différence est une nouvelle relation qui a les mêmes attributs que R_1 et R_2. Si R_1 est vide, la relation qui résulte de la différence est vide. Si R_2 est vide, la relation qui résulte de la différence est identique à R_1.

Le tableau 3.11 montre un exemple de différence.

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

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

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

TAB. 3.11 – Exemple de différence : $R = R_1 - R_2$

3.4.7 Produit cartésien

Définition 3.31 -produit cartésien- *Le produit cartésien est une opération portant sur deux relations R_1 et R_2 et qui construit une troisième relation regroupant exclusivement toutes les possibilités de combinaison des occurrences des relations R_1 et R_2, on la note $R_1 \times R_2$.*

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

$$\text{relation} \times \text{relation} \longrightarrow \text{relation}$$

Le résultat du produit cartésien est une nouvelle relation qui a tous les attributs de R_1 et tous ceux de R_2. Si R_1 ou R_2 ou les deux sont vides, la relation qui résulte du produit cartésien est vide. Le nombre d'occurrences de la relation qui résulte du produit cartésien est le nombre d'occurrences de R_1 multiplié par le nombre d'occurrences de R_2.

Le tableau 3.12 montre un exemple de produit cartésien.

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

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

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

TAB. 3.12 – Exemple de produit cartésien : $R = Amie \times Cadeau$

3.4.8 Jointure, theta-jointure, equi-jointure, jointure naturelle

Jointure

Définition 3.32 **-jointure-** *La jointure est une opération portant sur deux relations R_1 et R_2 qui construit une troisième relation regroupant exclusivement toutes les possibilités de combinaison des occurrences des relations R_1 et R_2 qui satisfont l'expression logique E. La jointure est notée $R_1 \bowtie_E R_2$.*

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

$$\text{relation} \times \text{relation} \times \text{expression logique} \longrightarrow \text{relation}$$

Si R_1 ou R_2 ou les deux sont vides, la relation qui résulte de la jointure est vide.
En fait, la jointure n'est rien d'autre qu'un produit cartésien suivi d'une sélection :

$$R_1 \bowtie_E R_2 = \sigma_E(R_1 \times R_2)$$

Le tableau 3.13 montre un exemple de jointure.

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

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

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

TAB. 3.13 – Exemple de jointure : $R = Famille \bowtie_{((Age \leq AgeC) \wedge (Prix < 50))} Cadeau$

Theta-jointure

Définition 3.33 **-theta-jointure-** *Une theta-jointure est une jointure dans laquelle l'expression logique E est une simple comparaison entre un attribut A_1 de la relation R_1 et un attribut A_2 de la relation R_2. La theta-jointure est notée $R_1 \bowtie_E R_2$.*

Equi-jointure

Définition 3.34 **-equi-jointure-** *Une equi-jointure est une theta-jointure dans laquelle l'expression logique E est un test d'égalité entre un attribut A_1 de la relation R_1 et un attribut A_2 de la relation R_2. L'equi-jointure est notée $R_1 \bowtie_{A_1,A_2} R_2$.*

Jointure naturelle

Définition 3.35 -jointure naturelle- *Une jointure naturelle est une equi-jointure dans laquelle les attributs des relations R_1 et R_2 portent le même nom A. Dans la relation construite, l'attribut A n'est pas dupliqué mais fusionné en un seul attribut. La jointure naturelle est notée $R_1 \bowtie R_2$.*

Le résultat de la jointure naturelle est une nouvelle relation qui a tous les attributs de R_1 et tous ceux de R_2 sauf A. Il est en fait indifférent d'éliminer l'attribut A de la relation R_1 ou R_2.

Le tableau 3.14 montre un exemple de jointure naturelle.

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

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

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

TAB. 3.14 – Exemple de jointure naturelle : R = *Famille* $\bowtie$ *Cadeau*

3.4.9 Division

Définition 3.36 -division- *La division est une opération portant sur deux relations R_1 et R_2, telles que le schéma de R_2 est strictement inclus dans celui de R_1, qui génère une troisième relation regroupant toutes les parties d'occurrences de la relation R_1 qui sont associées à toutes les occurrences de la relation R_2 ; on la note $R_1 \div R_2$.*

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

$$\text{relation} \times \text{relation} \longrightarrow \text{relation}$$

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

La relation R_2 ne peut pas être vide. Tous les attributs de R_2 doivent être présents dans R_1 et R_1 doit posséder au moins un attribut de plus que R_2 (inclusion stricte). Le résultat de la division est une nouvelle relation qui a tous les attributs de R_1 sans aucun de ceux de R_2. Si R_1 est vide, la relation qui résulte de la division est vide.

Le tableau 3.15 montre un exemple de division.

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

Relation *Etudiant*
Nom
Dubois
Pascal

Relation *R*
Enseignant
Germain
Fidus

TAB. 3.15 – Exemple de division : R = *Enseignement* $\div$ *Etudiant*. La relation R contient donc tous les enseignants de la relation *Enseignement* qui enseignent à tous les étudiants de la relation *Etudiant*.

3.5 Travaux Dirigés – Algèbre relationnelle

Soit le schéma relationnel suivant :
- Individu(Num-Ind, Nom, Prénom)
- Jouer(Num-Ind, Num-Film, Rôle)
- Film(Num-Film Num-Ind, Titre, Genre, Année)
- Projection(Num-Ciné, Num-Film, Date)
- Cinéma(Num-Ciné, Nom, Adresse)

Le tableau 3.16 donne une instance de ce schéma relationnel.

3.5.1 Exercices de compréhension de requêtes

Dans les exercices qui suivent, donnez, sous forme de relation, le résultat des requêtes formulées en algèbre relationnelle.

Sélection, et un peu de logique ...

1. $\sigma_{(Année<1996)}Film$
2. $\sigma_{(Année<2000 \wedge Genre="Drame")}Film$
3. $\sigma_{(Année<1990 \vee Genre="Drame")}Film$
4. $\sigma_{(\neg(Année>2000 \vee Genre="Policier"))}Film$
5. $\sigma_{(\neg(Année>2000))}\sigma_{(Genre="Drame")}Film$

Projection

6. $\Pi_{(Titre,Genre,Année)}Film$
7. $\Pi_{(Genre)}Film$
8. $\Pi_{(Genre)}\sigma_{(Année<2000)}Film$

Union

9. $(\Pi_{(Nom,Prénom)}\sigma_{(Prénom="John")}Individu) \cup (\Pi_{(Nom,Prénom)}\sigma_{(Prénom="Paul")}Individu)$

Intersection

10. $(\Pi_{(Prénom)}Individu) \cap (\Pi_{(Rôle)}Jouer)$

Différence

11. $(\Pi_{(Nom)}\sigma_{(Nom\sim"\hat{}[TW]")}Individu) - (\Pi_{(Nom)}\sigma_{(Prénom="John")}Individu)$

Remarque : $\sim$ est un opérateur de comparaison indiquant que l'élément qui suit n'est pas une chaîne de caractères mais une expression régulière (cf. section 4.5.8).

Produit cartésien

12. $(\Pi_{(Titre,Genre)}\sigma_{(Année\leq 1985)}Film) \times (\Pi_{(Nom)}Cinéma)$

Jointure

13. $\Pi_{(Titre,Nom,Prénom)}(Film \bowtie_{Num-Ind} Individu)$

Relation *Individu*		
Num-Ind	**Nom**	**Prénom**
01	Kidman	Nicole
02	Bettany	Paul
03	Watson	Emily
04	Skarsgard	Stellan
05	Travolta	John
06	L. Jackson	Samuel
07	Willis	Bruce
08	Irons	Jeremy
09	Spader	James
10	Hunter	Holly
11	Arquette	Rosanna
12	Wayne	John
13	von Trier	Lars
14	Tarantino	Quentin
15	Cronenberg	David
16	Mazursky	Paul
17	Jones	Grace
18	Glen	John

Relation *Projection*		
Num-Ciné	**Num-Film**	**Date**
02	05	01/05/2002
02	05	02/05/2002
02	05	03/05/2002
02	04	02/12/1996
01	01	07/05/1996
02	07	09/05/1985
01	04	02/08/1996
04	03	08/04/1994
03	06	02/12/1990
02	02	25/09/1990
03	03	05/11/1994
04	03	06/11/1994
01	06	05/07/1980
02	04	02/09/1996
04	06	01/08/2002
03	06	09/11/1960
01	02	12/03/1988

Relation *Film*				
Num-Film	**Num-Ind**	**Titre**	**Genre**	**Année**
05	13	Dogville	Drame	2002
04	13	Breaking the waves	Drame	1996
03	14	Pulp Fiction	Policier	1994
02	15	Faux-Semblants	Epouvante	1988
01	15	Crash	Drame	1996
06	12	Alamo	Western	1960
07	18	Dangereusement vôtre	Espionnage	1985

Relation *Jouer*		
Num-Ind	**Num-Film**	**Rôle**
01	05	Grace
02	05	Tom Edison
03	04	Bess
04	04	Jan
05	03	Vincent Vega
06	03	Jules Winnfield
07	03	Butch Coolidge
08	02	Beverly & Elliot Mantle
09	01	James Ballard
10	01	Helen Remington
11	01	Gabrielle
04	05	Chuck
16	07	May Day

Relation *Cinéma*		
Num-Ciné	**Nom**	**Adresse**
02	Le Fontenelle	78160 Marly-le-Roi
01	Le Renoir	13100 Aix-en-Provence
03	Gaumont Wilson	31000 Toulouse
04	Espace Ciné	93800 Epinay-sur-Seine

TAB. 3.16 – Exemple d'instance de schéma relationnel

Division

14. $(\Pi_{(Nom,Prénom,Titre)}(Film \bowtie_{Num-Film} Jouer \bowtie_{Num-Ind} Individu))$
$/(\Pi_{(Titre)}(Film \bowtie_{Num-Ind} (\sigma_{(Prénom="Lars")} Individu))$

3.5.2 Trouver la bonne requête

15. Quels sont les titres des films dont le genre est *Drame* ?
16. Quels films sont projetés au cinéma *Le Fontenelle* ?
17. Quels sont les noms et prénoms des réalisateurs ?
18. Quels sont les noms et prénoms des acteurs ?
19. Quels sont les noms et prénoms des acteurs qui sont également réalisateurs ?
20. Quels films (titres) ont été projetés en 2002 ?
21. Donnez le titre des films réalisés par Lars von Trier.
22. Quels sont les réalisateurs qui ont réalisé des films d'épouvante et des films dramatiques ?
23. Quels sont les titres des films où Nicole Kidman a joué un rôle et qui ont été projetés au cinéma *Le Fontenelle* ?
24. Quels sont les acteurs qui n'ont pas joué dans des films dramatiques ?
25. Quels sont les noms et prénoms des individus dont le prénom est à la fois celui d'un acteur et celui d'un réalisateur sans qu'il s'agisse de la même personne ?
26. Quels acteurs a-t-on pu voir au cinéma *Le Fontenelle* depuis l'an 2000 ?
27. Quels sont les films qui ont encore été à l'affiche 5 années après leur sortie ?
28. Quels sont les cinémas qui ont projeté tous les films ?
29. Quels sont les acteurs que l'on a pu voir dans toutes les salles ?

Chapitre 4

Langage SQL

4.1 Introduction

4.1.1 Présentation générale

Introduction

Le langage SQL (*Structured Query Language*) peut être considéré comme le langage d'accès normalisé aux bases de données. Il est aujourd'hui supporté par la plupart des produits commerciaux que ce soit par les systèmes de gestion de bases de données micro tel que *Access* ou par les produits plus professionnels tels que *Oracle*. Il a fait l'objet de plusieurs normes ANSI/ISO dont la plus répandue aujourd'hui est la norme SQL2 qui a été définie en 1992.

Le succès du langage SQL est dû essentiellement à sa simplicité et au fait qu'il s'appuie sur le schéma conceptuel pour énoncer des requêtes en laissant le SGBD responsable de la stratégie d'exécution. Le langage SQL propose un langage de requêtes ensembliste et assertionnel. Néanmoins, le langage SQL ne possède pas la puissance d'un langage de programmation : entrées/sorties, instructions conditionnelles, boucles et affectations. Pour certains traitements il est donc nécessaire de coupler le langage SQL avec un langage de programmation plus complet.

De manière synthétique, on peut dire que SQL est un langage relationnel, il manipule donc des tables (*i.e.* des relations, c'est-à-dire des ensembles) par l'intermédiaire de requêtes qui produisent également des tables.

Historique rapide

- En 1970, E.F. CODD, directeur de recherche du centre IBM de San José, invente le modèle relationnel qui repose sur une algèbre relationnelle. Ce modèle provoque une révolution dans l'approche des bases des données.
- En 1977, création du langage SEQUEL (Structured English Query Language) et mise en place du Système R, prototype de base de données reposant sur la théorie de CODD. SEQUEL continue de s'enrichir pour devenir SQL (Structured Query Language).
- En 1981, la société ORACLE CORP lance la première version de son système de gestion de base de données relationnelle (SGBDR), IBM sort SQL/DS et RTI lance INGRES.
- En 1982, IBM sort SQL/DS pour son environnement VM/CMS et l'ANSI (American National Standard Institute) lance un projet de normalisation d'un langage relationnel.
- En 1983, IBM lance DB2 pour l'environnement MVS.
- En 1986, la sociéte SYBASE lance son SGBDR conçu selon le modèle Client-Serveur.
- La première norme SQL (SQL-1) de l'ISO (International Standard Organisation) apparaît. Il existe désormais plusieurs dizaines de produits proposant le langage SQL et tournant sur des machines allant des micros aux gros systèmes.
- Depuis, les différents produits phares ont évolué, la norme SQL est passée à SQL-2, puis SQL-3. SQL est désormais un langage incontournable pour tout SGBD moderne. Par contre, bien qu'une

norme existe, on assiste à une prolifération de dialectes propres à chaque produit : soit des sous-ensembles de la norme (certaines fonctionnalités n'étant pas implantées), soit des sur-ensembles (ajout de certaines fonctionnalités, propres à chaque produit).

Oracle et *Informix* dominent le marché actuel, *SQL-Server* (de Microsoft) tente de s'imposer dans le monde des PC sous NT. À côté des ces produits, très chers, existent heureusement des systèmes libres et gratuits : *MySQL* et *PostgreSQL* sont les plus connus.

Bien que ces SGBDR n'aient pas la puissance des produits commerciaux, certains s'en approchent de plus en plus. Les différences notables concernent principalement les environnements de développement qui sont de véritables ateliers logiciels sous *Oracle* et qui sont réduits à des interfaces de programmation *C*, *Python*, *Perl* sous *PostgreSQL*. Il en va de même pour les interfaces utilisateurs : il en existe pour *PostgreSQL*, mais ils n'ont certainement pas la puissance de leurs équivalents commerciaux.

Terminologie

Modèle relationnel		
Français	**Anglais**	**Standard SQL**
Relation	Relation	Table
Domaine	Domain	Domaine
Attribut	Attribute	Colonne
n-uplet	tuple	Ligne
Clé primaire	Primary key	Primary key

4.1.2 Catégories d'instructions

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

1. la définition des éléments d'une base de données (tables, colonnes, clefs, index, contraintes, ...),
2. la manipulation des données (insertion, suppression, modification, extraction, ...),
3. la gestion des droits d'accès aux données (acquisition et révocation des droits),
4. la gestion des transactions,
5. et enfin le SQL intégré.

Langage de définition de données

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

Les instructions du LDD sont : CREATE, ALTER, DROP, AUDIT, NOAUDIT, ANALYZE, RENAME, TRUNCATE.

Langage de manipulation de données

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

Les instructions du LMD sont : INSERT, UPDATE, DELETE, SELECT, EXPLAIN, PLAN, LOCK TABLE.

Ces éléments doivent être validés par une transaction pour qu'ils soient pris en compte.

Langage de protections d'accès

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

Les instructions du DCL sont : GRANT, REVOKE.

Langage de contrôle de transaction

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

Les instructions du TCL sont : COMMIT, SAVEPOINT, ROLLBACK, SET TRANSACTION

SQL intégré

Le **SQL intégré** (*Embedded SQL*) permet d'utiliser SQL dans un langage de troisième génération (C, Java, Cobol, etc.) :

- déclaration d'objets ou d'instructions ;
- exécution d'instructions ;
- gestion des variables et des curseurs ;
- traitement des erreurs.

Les instructions du SQL intégré sont : DECLARE, TYPE, DESCRIBE, VAR, CONNECT, PREPARE, EXECUTE, OPEN, FETCH, CLOSE, WHENEVER.

4.1.3 PostgreSQL

Les systèmes traditionnels de gestion de bases de données relationnelles (SGBDR) offrent un modèle de données composé d'une collection de relations contenant des attributs relevant chacun d'un type spécifique. Les systèmes commerciaux gèrent par exemple les nombres décimaux, les entiers, les chaînes de caractères, les monnaies et les dates. Il est communément admis que ce modèle est inadéquat pour les applications de traitement de données de l'avenir car, si le modèle relationnel a remplacé avec succès les modèles précédents en partie grâce à sa « simplicité spartiate », cette dernière complique cependant l'implémentation de certaines applications. PostgreSQL apporte une puissance additionnelle substantielle en incorporant les quatre concepts de base suivants afin que les utilisateurs puissent facilement étendre le système : classes, héritage, types, fonctions. D'autres fonctionnalités accroissent la puissance et la souplesse : contraintes, déclencheurs, règles, intégrité des transactions.

Ces fonctionnalités placent PostgreSQL dans la catégorie des bases de données relationnel-objet. Ne confondez pas cette catégorie avec celle des serveurs d'objets qui ne tolère pas aussi bien les langages traditionnels d'accès aux SGBDR. Ainsi, bien que PostgreSQL possède certaines fonctionnalités orientées objet, il appartient avant tout au monde des SGBDR. C'est essentiellement l'aspect SGBDR de PostgreSQL que nous aborderons dans ce cours.

L'une des principales qualités de PostgreSQL est d'être un logiciel libre, c'est-à-dire gratuit et dont les sources sont disponibles. Il est possible de l'installer sur les systèmes *Unix/Linux* et *Win32*.

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

- d'une partie serveur, c'est-à-dire une application fonctionnant sur la machine hébergeant la base de données (le serveur de bases de données) capable de traiter les requêtes des clients ; il s'agit dans le cas de PostgreSQL d'un programme résident en mémoire appelé *postmaster* ;
- d'une partie client (`psql`) devant être installée sur toutes les machines nécessitant d'accéder au serveur de base de données (un client peut éventuellement fonctionner sur le serveur lui-même).

Les clients (les machines sur lesquelles le client PostgreSQL est installé) peuvent interroger le serveur de bases de données à l'aide de requêtes SQL.

4.2 Définir une base – Langage de définition de données (LDD)

4.2.1 Introduction aux contraintes d'intégrité

Soit le schéma relationnel minimaliste suivant :
- Acteur(Num-Act, Nom, Prénom)
- Jouer(Num-Act, Num-Film)
- Film(Num-Film, Titre, Année)

Contrainte d'intégrité de domaine

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

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

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

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

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

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

Contrainte d'intégrité de référence

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

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

Lors de la suppression d'un n-uplet dans la relation *Acteur*, le SGBD doit vérifier qu'aucun n-uplet de la relation *Jouer* ne fait référence, par l'intermédiaire de l'attribut `Num-Act`, au n-uplet que l'on cherche à supprimer. Le cas échéant, c'est-à-dire si une, ou plusieurs, valeur correspondante de `Num-Act` existe dans *Jouer*, quatre possibilités sont envisageables :
- interdire la suppression ;
- supprimer également les n-uplets concernés dans *Jouer* ;
- avertir l'utilisateur d'une incohérence ;
- mettre les valeurs des attributs concernés à une valeur nulle dans la table *Jouer*, si l'opération est possible (ce qui n'est pas le cas si ces valeurs interviennent dans une clé primaire) ;

4.2.2 Créer une table : `CREATE TABLE`

Introduction

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

Création simple

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

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

Quand on crée une table, il faut définir les contraintes d'intégrité que devront respecter les données que l'on mettra dans la table (cf. section 4.2.3).

Les types de données

Les types de données peuvent être :

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

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

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

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

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

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

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

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

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

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

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

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

Création avec Insertion de données

On peut insérer des données dans une table lors de sa création par la commande suivante :

```
CREATE TABLE nom_table [(nom_col1, nom_col2, ...)] AS SELECT ...
```

On peut ainsi, en un seul ordre SQL créer une table et la remplir avec des données provenant du résultat d'un `SELECT` (cf. section 4.5 et 4.7). Si les types des colonnes ne sont pas spécifiés, ils correspondront à ceux du `SELECT`. Il en va de même pour les noms des colonnes. Le `SELECT` peut contenir des fonctions de groupes mais pas d'`ORDER BY` (cf. section 4.7.2 et 4.5.6) car les lignes d'une table ne peuvent pas être classées.

4.2.3 Contraintes d'intégrité

Syntaxe

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

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

Contraintes de colonne

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

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

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

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

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

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

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

Contraintes de table

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

`PRIMARY KEY (colonne, ...) :` Désigne la concaténation des attributs cités comme clé primaire de la table. Cette contrainte ne peut apparaître qu'une seule fois dans l'instruction.

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

`FOREIGN KEY (colonne, ...) REFERENCES table [(colonne, ...)]`
`[ON DELETE CASCADE | SET NULL] :` Contrainte d'intégrité référentielle pour un ensemble d'attributs de la table en cours de définition. Les valeurs prises par ces attributs doivent exister dans l'ensemble d'attributs spécifié et posséder une contrainte `PRIMARY KEY` ou `UNIQUE` dans la table `table.`

`CHECK (condition)` : Cette contrainte permet d'exprimer une condition qui doit exister entre plusieurs attributs de la ligne.

Les contraintes de tables portent sur plusieurs attributs de la table sur laquelle elles sont définies. Il n'est pas possible de définir une contrainte d'intégrité utilisant des attributs provenant de deux ou plusieurs tables. Ce type de contrainte sera mis en œuvre par l'intermédiaire de déclencheurs de base de données (*trigger*, cf. section **??**).

Complément sur les contraintes

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

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

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

4.2.4 Supprimer une table : `DROP TABLE`

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

La syntaxe est la suivante :

```
DROP TABLE nom_table
```

4.2.5 Modifier une table : `ALTER TABLE`

Ajout ou modification de colonnes

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

Ajout d'une contrainte de table

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

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

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

Renommer une colonne

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

Renommer une table

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

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

4.3.1 Insertion de n-uplets : `INSERT INTO`

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

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

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

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

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

Le `SELECT` (cf. section 4.5 et 4.7) peut contenir n'importe quelle clause sauf un `ORDER BY` (cf. section 4.5.6).

4.3.2 Modification de n-uplets : `UPDATE`

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

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

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

4.3.3 Suppression de n-uplets : `DELETE`

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

La syntaxe est la suivante :

```
DELETE FROM nom_table
WHERE predicat
```

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

4.4 Travaux Pratiques – PostgreSQL : Première base de données

4.4.1 Informations pratiques concernant PostgreSQL

Initialisation et démarrage de PostgreSQL

L'initialisation de PostgreSQL consiste à créer un cluster de bases de données de la manière suivante :

```
/répertoire_des_binaires/initdb -D /répertoire_choisi_pour_la_base
```

Il faut ensuite lancer le serveur PostgreSQL :

```
/répertoire_des_binaires/postmaster -D /répertoire_choisi_pour_la_base
```

Une meilleure solution consiste à lancer le serveur PostgreSQL en tâche de fond et à diriger son flux de sortie vers un fichier (`logfile`) :

```
/répertoire_des_binaires/postmaster -D /répertoire_choisi_pour_la_base  > logfile 2>&1 &
```

La création proprement dite d'une base de données dans le cluster se fait de la manière suivante :

```
createdb nom_de_la_nouvelle_base
```

Nous pouvons enfin utiliser l'interface en ligne de commande de PostgreSQL en démarrant un client :

```
psql nom_de_la_nouvelle_base
```

Remarque concernant SELinux

Attention, il y a des incompatibilités entre PostgreSQL et SELinux. Si vous rencontrez des problèmes, essayez de désactiver temporairement SELinux (`setenforce 0`). Cette solution n'est pas la meilleure. Si elle marche, essayez de corriger le problème plus finement et de manière définitive. Par exemple, sous une installation standard de Fedora Core 3, pour corriger le problème, procédez de la manière suivante :

- Cliquer sur : `Menu principal > Paramètres de système > Niveau de sécurité` ;
- Cliquer sur l'onglet `SELinux` puis développer `SELinux Service Protection` et cocher la case `Disable SELinux protection for postgresql daemon` et valider.

PostgreSQL à l'IUT

À l'IUT, un seul cluster de base de données est créé et disponible pour tous les utilisateurs de PostgreSQL. Une seule base de données est affectée à chaque utilisateur ; son nom étant l'identifiant de l'utilisateur (*i.e.* nom de *login*).

Pour créer la base de données, il faut :

- ouvrir internet *Galeon*,
- puis cliquer sur « *Etat de votre base de données PostgreSQL* »
- et enfin sur « *Créer la base de données* ».

Le démarrage du client se fait de la manière suivante :

```
psql -h nom_serveur -p num_port ma_base identifiant
```

En salle de TP, `nom_serveur` est `aquanux` ; les champs `num_port`, `ma_base` et `identifiant` sont optionnels et inutiles, pour information :

- `num_port` : 5432 ;
- `ma_base` : votre identifiant ;
- `identifiant` : votre identifiant.

Écriture des commandes sous PostgreSQL

Toutes les lignes de commandes SQL doivent se terminer par un « ; » ! Ce n'est, par contre, pas le cas des méta-commandes dont il est question ci-dessous.

Méta-commandes sous PostgreSQL

Méta-commandes	Description
\?	Afficher toutes les méta-commandes
\h	Afficher toutes les commandes SQL
\h *nom_commande*	Aide concernant une commande SQL particulière
\df	Afficher toutes les fonctions postgresql
\cd *nom_repertoire*	Changer de répertoire courant
\! *nom_commande*	Exécuter une commande shell
\i *nom_fichier*	Lire et exécuter un script SQL
\d	Afficher la liste des tables créées
\d *nom_table*	Information concernant une table créée
\copy *nom_table* from *nom_fichier*	Remplissage d'une table à partir d'un fichier texte

4.4.2 Première base de données

1. Créez votre base de données en utilisant internet *Galeon*.
2. Démarrez un client (`psql -h aquanux`) pour vous connecter à PostgreSQL.
3. Tapez \ ? pour afficher la liste des méta-commandes.
4. Tapez `\h CREATE TABLE` pour connaître la syntaxe de la commande SQL de création de table.
5. Créez les tables du schéma relationnel vu en travaux dirigés section 3.5.
   Schéma relationnel :
   – `film (num_film, num_realisateur, titre, genre, annee)`
   – `cinema (num_cinema, nom, adresse)`
   – `individu (num_individu, nom prénom)`
   – `jouer (num_acteur, num_film, role)`
   – `projection (num_cinema, num_film, jour)`
   N'oubliez surtout pas :
   – de choisir correctement le domaine de définition (*i.e.* le type) de chacun des attributs ;
   – de bien préciser la clé primaire de chaque relation ;
   – les contraintes d'intégrité référentielles (*i.e.* les clefs étrangères).
6. Affichez la liste des tables créées (`\d`).
7. Remplissez « à la main », c'est-à-dire en utilisant la commande `INSERT INTO`, la table `cinema` en utilisant le tableau 3.16.
8. Remplissez les tables `jouer`, `film`, `projection` et `individu` à l'aide des fichiers fournis (`jouer.txt`, `film.txt`, `projection.txt` et `individu.txt`) en utilisant la méta-commande adéquate (`\copy` *nom_table* `from` *nom_fichier*).
   Devez-vous respecter un ordre de remplissage des tables ?
   Pourquoi ?
9. Créez un fichier `cinema.txt` permettant de remplir la table `cinema` en respectant le format des fichiers qui vous ont été fournis.
10. Créez un script SQL (`GenBDCine.sql`) permettant de régénérer votre base de données. Ce fichier, composé de trois parties, doit permettre de :
    (a) effacer chacune des tables ;
    (b) créer chacune des tables comme dans l'exercice 5 ;
    (c) remplir chacune des tables.
11. Restaurez votre base de données en utilisant le fichier `GenBDCine.sql`.
12. Vous voulez effacer l'acteur *John Travolta* de la base.
    Quelles opérations sont nécessaires pour mener à bien cet suppression ?
    Réalisez cette suppression.

Remarque – Pour afficher l'ensemble des n-uplets d'une table, vous pouvez utiliser la commande SQL : `SELECT * FROM nom_table`.

4.5 Interroger une base – Langage de manipulation de données (LMD) : SELECT (1re partie)

4.5.1 Introduction à la commande SELECT

Introduction

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

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

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

Syntaxe simplifiée de la commande SELECT

Une requête se présente généralement sous la forme :

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

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

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

```
SELECT * FROM film
```

De manière synthétique, on peut dire que la clause `SELECT` permet de réaliser la *projection*, la clause `FROM` le *produit cartésien* et la clause `WHERE` la *sélection* (cf. section 4.5.2).

Délimiteurs : apostrophes simples et doubles

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

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

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

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

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

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

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

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

Traduction de l'opérateur de projection

L'opérateur de projection $\Pi_{(A_1, \ldots A_n)}(relation)$ se traduit tout simplement en SQL par la requête :

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

`DISTINCT` permet de ne retenir qu'une occurrence de n-uplet dans le cas où une requête produit plusieurs n-uplets identiques (cf. section 4.5.4).

Traduction de l'opérateur de sélection

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

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

De manière simplifiée, un *prédicat* est une expression logique sur des *comparaisons*. Reportez-vous à la section 4.5.7 pour une description plus complète.

Traduction de l'opérateur de produit cartésien

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

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

Nous reviendrons sur le produit cartésien dans les sections 4.5.5 et 4.7.1.

Traduction de l'opérateur d'équi-jointure

L'opérateur d'équi-jointure $relation_1 \bowtie_{A_1,A_2} relation_2$ se traduit en SQL par la requête :

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

Nous reviendrons sur les différents types de jointure dans la section 4.7.1.

4.5.3 Syntaxe générale de la commande SELECT

Voici la syntaxe générale d'une commande SELECT :

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

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

`SELECT` : Cette clause permet de spécifier les attributs que l'on désire voir apparaître dans le résultat de la requête (cf. section 4.5.4).

`FROM` : Cette clause spécifie les tables sur lesquelles porte la requête (cf. section 4.5.5 et 4.7.1).

`WHERE` : Cette clause permet de filtrer les n-uplets en imposant une condition à remplir pour qu'ils soient présents dans le résultat de la requête (cf. section 4.5.7).

`GROUP BY` : Cette clause permet de définir des groupes (*i.e.* sous-ensemble ; cf. section 4.7.2).

`HAVING` : Cette clause permet de spécifier un filtre (condition de regroupement des n-uplets) portant sur les résultats (cf. section 4.7.2).

`UNION`, `INTERSECT` et `EXCEPT` : Cette clause permet d'effectuer des opérations ensemblistes entre plusieurs résultats de requête (*i.e.* entre plusieurs `SELECT`) (cf. section 4.7.3).

`ORDER BY` : Cette clause permet de trier les n-uplets du résultat (cf. section 4.5.6).

4.5.4 La clause SELECT

Introduction

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

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

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

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

L'opérateur *étoile* (*)

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

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

```
SELECT * FROM employee
```

Les opérateurs DISTINCT et ALL

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

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

```
SELECT DISTINCT name FROM employee
```

Les opérations mathématiques de base

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

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

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

L'opérateur AS

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

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

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

L'opérateur de concaténation

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

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

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

4.5.5 La clause FROM (1re partie)

Comportement

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

L'opérateur AS

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

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

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

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

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

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

Sous-requête

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

Par exemple, les deux requêtes suivantes sont équivalentes :

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

Les jointures

Nous traiterons cet aspect de la clause FROM dans la section 4.7.1.

4.5.6 La clause ORDER BY

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

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

expression désigne soit une colonne, soit une opération mathématique de base (nous avons abordé ce type d'opérations dans la section 4.5.4 sur « La clause SELECT ») sur les colonnes.

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

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

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

La clause `ORDER BY` permet de trier le résultat final de la requête, elle est donc la dernière clause de tout ordre SQL et ne doit figurer qu'une seule fois dans le `SELECT`, même s'il existe des requêtes imbriquées ou un jeu de requêtes ensemblistes (cf. section 4.7.3).

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

4.5.7 La clause `WHERE`

Comportement

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

```
WHERE prédicat
```

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

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

Expression simple

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

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

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

Prédicat simple

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

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

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

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

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

Dans ce cas, la chaîne de caractères faisant l'objet du test est à gauche et correspond à une *expression simple* du type chaîne de caractères, il s'agit généralement d'un nom de colonne. L'expression régulière, qui s'écrit entre apostrophe simple, comme une chaîne de caractères, est située à droite de l'opérateur. La section 4.5.8 donne une description détaillée du formalisme des expressions régulières.

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

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

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

Prédicat composé

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

4.5.8 Les expressions régulières

Introduction

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

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

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

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

La syntaxe et le pouvoir expressif des expressions régulières diffèrent pour ces trois opérateurs. Nous ne décrirons ici que la syntaxe du formalisme le plus standard et le plus puissant, celui que l'on retrouve sous *Unix* avec les commandes `egrep`, `sed` et `awk`. Ce formalisme est celui associé à l'opérateur `~`.

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

```
expression='chaine'
```

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

```
expression~'expression_régulière'
```

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

Formalisme

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

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

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

`[^liste_de_caractères]` : est une expression régulière qui décrit les caractères qui ne sont pas dans la liste de caractères.

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

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

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

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

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

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

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

`expreg{n}` : est une expression régulière qui décrit `expreg` `n` fois.

`expreg{n,}` : est une expression régulière qui décrit `expreg` au moins `n` fois.

`expreg{n,m}` : décrit `expreg` au moins `n` fois et au plus `m` fois.

`expreg_1expreg_2` : est une expression régulière qui décrit une chaîne constituée de la concaténation de deux sous-chaînes respectivement décrites par `expreg_1` et `expreg_2`.

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

`(expreg)` : est une expression régulière qui décrit ce que décrit `expreg`.

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

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

Exemples

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

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

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

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

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

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

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

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

4.6 Travaux Pratiques – PostgreSQL : Premières requêtes

Dans les exercices de cette section, l'objectif est de trouver les requêtes SQL permettant de répondre aux problèmes posés. Nous utilisons ici la base de données sur les films (cf. séance de travaux pratiques 4.4).

4.6.1 Premières requêtes

1. Quel est le contenu de la table individu ?
2. Quels sont les prénoms des individus en conservant les doublons ?
3. Quels sont les prénoms des individus en conservant les doublons, mais en les classant par ordre alphabétique ?
4. Quels sont les prénoms des individus sans doublons ?
   Observez le résultat en effectuant un classement alphabétique et sans effectuer de classement.
5. Quels sont les individus dont le prénom est `John` ?
6. Quel est le nom des individus dont le prénom est `John` ?
7. Dressez la liste de toutes les associations possibles entre un individu et un film (il n'y a pas nécessairement de lien entre l'individu et le film qu'on lui associe). Observez le nombre de lignes retournées. Était-il prévisible ?
8. Quels sont les individus qui sont des acteurs ?
9. Dressez la liste de toutes les associations possibles entre un acteur et un film (il n'y a pas nécessairement de lien entre l'acteur et le film qu'on lui associe). Observez le nombre de lignes retournées.
10. Dressez la liste de toutes les interprétations, en précisant le rôle, d'acteur, dont on précisera le nom et le prénom, ayant joué dans des films dont on précisera le titre. Le résultat sera de la forme :

```
 prenom |   nom    |     role     |    titre
--------+----------+--------------+--------------
 Nicole | Kidman   | Grace        | Dogville
 Paul   | Bettany  | Tom Edison   | Dogville
```

11. Même question que la précédente, mais en formattant le résultat de la manière suivante :

```
                              listing
-----------------------------------------------------------------
 Nicole Kidman a joué le rôle de Grace dans le film Dogville
 Paul Bettany a joué le rôle de Tom Edison dans le film Dogville
```

4.6.2 Requêtes déjà résolues en utilisant l'algèbre relationnelle (cf. travaux dirigés section 3.5.2)

12. Quels sont les titres des films dont le genre est *Drame* ?
13. Quels films (titres) ont été projetés en 2002 ?
14. Donnez le titre des films réalisés par Lars von Trier.
15. Quels films sont projetés au cinéma *Le Fontenelle* ?
16. Quels sont les noms et prénoms des réalisateurs ?
17. Quels sont les noms et prénoms des acteurs ?
18. Quels sont les noms et prénoms des acteurs qui sont également réalisateurs ?
    Remarque : vous ne pouvez utiliser le mot clef `INTERSECT` puisque nous ne l'avons pas encore vu.
19. Quels acteurs a-t-on pu voir au cinéma *Le Fontenelle* depuis l'an 2000 ?
20. Quels sont les titres des films où Nicole Kidman a joué un rôle et qui ont été projetés au cinéma *Le Fontenelle* ?

4.6.3 Utilisation des expressions régulières

21. Quels sont les prénoms des individus qui contiennent la lettre `s` ?
22. Même question que la précédente mais sans distinguer les lettres en fonction de la casse.
23. Quels sont les prénoms des individus dont le prénom commence par la lettre `s` sans tenir compte de la casse ?
24. Quels sont les prénoms des individus dont le prénom se termine par la lettre `s` sans tenir compte de la casse ?
25. Quels sont les prénoms des individus dont le prénom contient la lettre `e` sans commencer ou finir par cette lettre et sans tenir compte de la casse ?
26. Quels sont les prénoms des individus qui ne contiennent pas la lettre `e` ?
27. Quels sont les prénoms des individus qui contiennent les lettres `a` et `l` dans un ordre quelconque et sans tenir compte de la casse ?
28. Quels sont les noms des individus qui contiennent la chaîne `an` ou la chaîne `on` ?
    Répondez en utilisant :
    (a) l'opérateur `|` des expressions régulières ;
    (b) les listes de caractères des expressions régulières ;
    (c) l'opérateur `OR` de la clause `WHERE`.
29. Quels sont les titres des films qui contiennent au moins trois `e` ?

4.7 Interroger une base – Langage de manipulation de données (LMD) : SELECT (2e partie)

4.7.1 La clause FROM (2e partie) : les jointures

Recommandation

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

Le produit cartésien

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

$$R_1 \bowtie_E R_2$$

Dans la section 3.4.8, nous avons vu que la jointure n'est rien d'autre qu'un produit cartésien suivi d'une sélection :

$$R_1 \bowtie_E R_2 = \sigma_E(R_1 \times R_2)$$

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

$$R_1 \times R_2 = R_1 \bowtie_{true} R_2$$

Nous avons vu section 4.5.5 que le produit cartésien entre deux tables `table_1` et `table_2` peut s'écrire en SQL :

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

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

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

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

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

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

Syntaxe générale des jointures

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

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

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

`ON` : La clause `ON` correspond à la condition de jointure la plus générale. Le prédicat `predicat` est une expression logique de la même nature que celle de la clause `WHERE` décrite dans la section 4.5.7.

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

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

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

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

`LEFT OUTER JOIN` : Dans un premier temps, une jointure interne (*i.e.* de type `INNER JOIN`) est effectuée. Ensuite, chacune des lignes de la table `table_1` qui ne satisfait pas la condition de jointure avec aucune des lignes de la table `table_2` (*i.e.* les lignes de `table_1` qui n'apparaissent pas dans la table résultat de la jointure interne) est ajoutée à la table résultats. Les attributs correspondant à la table `table_2`, pour cette ligne, sont affectés de la valeur `NULL`. Ainsi, la table résultat contient au moins autant de lignes que la table `table_1`.

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

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

La jointure externe droite peut être obtenue par une jointure externe gauche dans laquelle on inverse l'ordre des tables (et vice-versa). La jointure externe bilatérale peut être obtenue par la combinaison de deux jointures externes unilatérales avec l'opérateur ensembliste `UNION` que nous verrons dans la section 4.7.3.

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

Définition de deux tables pour les exemples qui suivent

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

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

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

La table `realisateur` contient les lignes suivantes :

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

La table `film` contient les lignes suivantes :

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

Exemples de jointures internes

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

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

pour produire le résultat suivant :

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

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

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

Mais la colonne `id_real` aurait été dupliquée :

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

Exemples de jointures externes gauches

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

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

Elle produit le résultat suivant :

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

Naturellement, en écrivant :

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

la colonne `id_real` serait dupliquée :

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

Exemples de jointures externes droites

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

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

Elle produit le résultat suivant :

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

Exemples de jointures externes bilatérales

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

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

Elle produit le résultat suivant :

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

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

Syntaxe

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

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

La clause `GROUP BY`

La commande `GROUP BY` permet de définir des regroupements (*i.e.* des agrégats) qui sont projetés dans la table résultat (un regroupement correspond à une ligne) et d'effectuer des calculs statistiques, définis par les expressions `fonction_agrégation [, ...]`, pour chacun des regroupements. La liste d'expressions `expression_1, [...,] expression_N` correspond généralement à une liste de colonnes `colonne_1, [...,] colonne_N`. La liste de colonnes spécifiée derrière la commande `SELECT` doit être identique à la liste de colonnes de regroupement spécifiée derrière la commande `GROUP BY`. A la place des noms de colonne il est possible de spécifier des opérations mathématiques de base sur les colonnes (comme définies dans la section 4.5.4). Dans ce cas, les regroupements doivent porter sur les mêmes expressions.

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

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

Les fonctions d'agrégation

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

`COUNT( * | [DISTINCT | ALL] expression ) :` Dénombre le nombre de lignes du résultat de la requête. Si `expression` est présent, on ne compte que les lignes pour lesquelles cette expression n'est pas `NULL`.

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

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

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

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

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

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

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

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

Exemples

Reprenons la base de données de la séance de travaux pratiques 4.4 dont le schéma relationnel était :
- `film (num_film, num_realisateur, titre, genre, annee)`
- `cinema (num_cinema, nom, adresse)`
- `individu (num_individu, nom prenom)`
- `jouer (num_acteur, num_film, role)`
- `projection (num_cinema, num_film, jour)`

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

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

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

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

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

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

La clause HAVING

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

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

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

Exemples

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

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

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

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

4.7.3 Opérateurs ensemblistes : UNION, INTERSECT et EXCEPT

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

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

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

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

Dans une requête on ne peut trouver qu'une seule instruction ORDER BY. Si elle est présente, elle doit être placée dans la dernière requête (cf. section 4.5.6). La clause ORDER BY ne peut faire référence qu'aux numéros des colonnes (la première portant le numéro 1), et non pas à leurs noms, car les noms peuvent être différents dans chacune des requêtes sur lesquelles porte le ou les opérateurs ensemblistes.

Les opérateurs UNION et INTERSECT sont commutatifs.

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

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

4.7.4 Traduction des opérateurs d'union, d'intersection, de différence et de division de l'algèbre relationnelle (2^e partie)

Traduction de l'opérateur d'union

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

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

Traduction de l'opérateur d'intersection

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

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

Traduction de l'opérateur de différence

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

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

Traduction de l'opérateur de division

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

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

Cela peut se reformuler par :

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

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

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

On peut donc reformuler le problème de la manière suivante :

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

Ce qui correspond à la requête SQL :

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

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

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

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

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

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

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

4.8 Travaux Pratiques – PostgreSQL : Requêtes avancées

Dans les exercices de cette section, l'objectif est de trouver les requêtes SQL permettant de répondre aux problèmes posés. Nous utilisons la base de données sur le cinéma (cf. séance de travaux pratiques 4.4). **Contrairement à la séance de travaux pratiques 4.6, nous utilisons maintenant la commande `JOIN` pour toutes les jointures des requêtes.**

4.8.1 Prix de `GROUP`

1. Dressez la liste de toutes les interprétations, en précisant le rôle, d'acteur, dont on précisera le nom et le prénom, ayant joué dans des films dont on précisera le titre.
2. On désire connaître le nom et le prénom des acteurs et le nombre de films dans lesquels ils ont joué.
3. On désire connaître le nom et le prénom des acteurs, le nombre de films dans lequel ils ont joué ainsi que l'année du film de leur premier et de leur dernier rôle.
4. On désire connaître le nom et le prénom des acteurs et le nombre de films dans lesquels ils ont joué pour les acteurs ayant joué dans strictement plus d'un film.
5. On désire connaître le nom et le prénom des acteurs et le nombre de drames dans lesquels ils ont joué.

4.8.2 Requêtes déjà résolues en utilisant l'algèbre relationnelle (cf. travaux dirigés section 3.5.2)

6. Quels sont les noms et prénoms des acteurs qui sont également réalisateurs ?
   Remarque : vous devez utiliser le mot clef `INTERSECT` puisque nous l'avons maintenant vu.
7. Quels sont les réalisateurs qui ont réalisé des films d'épouvante et des films dramatiques ?
8. Quels sont les acteurs qui n'ont pas joué dans des films dramatiques ?
9. Quels sont les cinémas qui ont projeté tous les films ?
10. Quels sont les acteurs que l'on a pu voir dans toutes les cinémas ?

4.8.3 `GROUP` toujours !

11. Quel est le nombre de films réalisés par chacun des réalisateurs ?
12. Combien de films à réalisé le réalisateur qui en a le plus réalisés ?
13. Quel sont les réalisateurs (il peut y en avoir un ou plusieurs execo) ayant réalisé le plus de films ? Comment serait-il possible de simplifier cette requête ?

14. Quel est le nombre de films réalisés par les réalisateurs, dont on désire connaître le nom et le prénom, ayant réalisé au moins un film du même genre que l'un des films réalisés par *David Cronenberg* ?
15. On suppose que les têtes d'affiche d'un film sont les acteurs recensés pour ce film dans la base de données. Quel est le nombre de têtes d'affiche et le réalisateur de chacun des films ?
16. En supposant qu'un film coûte 1000000 € plus 200000 € par tête d'affiche, donnez le prix moyen des films réalisés par chacun des réalisateurs.

4.9 Nouveaux objets – Langage de définition de données (LDD)

4.9.1 Séquences (CREATE SEQUENCE) et type SERIAL

Création d'une séquence

Une séquence est en fait une table spéciale contenant une seule ligne. Cet objet est utilisé pour créer une suite de nombres entiers dont l'évolution, généralement croissante, est régie par un certain nombre de paramètres.

Voici la syntaxe de création d'une séquence :

```
CREATE SEQUENCE nom [ INCREMENT [ BY ] incrément ]
    [ MINVALUE valeurmin ]
    [ MAXVALUE valeurmax ]
    [ START [ WITH ] début ]
    [ [ NO ] CYCLE ]
```

La commande `CREATE SEQUENCE` crée un nouveau générateur de nombre. Ceci implique la création et l'initialisation d'une nouvelle table portant le nom `nom`.

`INCREMENT BY` : La clause optionnelle `INCREMENT BY incrément` spécifie la valeur ajoutée à la valeur de la séquence courante pour créer une nouvelle valeur. Une valeur positive créera une séquence ascendante, une négative en créera une descendante. La valeur par défaut est 1.

`MINVALUE` : La clause optionnelle `MINVALUE valeurmin` précise la valeur minimale qu'une séquence peut générer. Si cette clause n'est pas fournie, alors les valeurs par défaut seront utilisées. Les valeurs par défaut sont 1 et $-2^{63} - 1$ pour les séquences respectivement ascendantes et descendantes.

`MAXVALUE` : La clause optionnelle `MAXVALUE valeurmax` précise la valeur maximale pour la séquence. Si cette clause n'est pas fournie, alors les valeurs par défaut seront utilisées. Les valeurs par défaut sont $2^{63} - 1$ et -1 pour les séquences respectivement ascendantes et descendantes.

`START WITH` : La clause optionnelle `START WITH début` précise la valeur d'initialisation de la séquence. La valeur de début par défaut est `valeurmin` pour les séquences ascendantes et `valeurmax` pour les séquences descendantes.

`[ NO ] CYCLE` : L'option `CYCLE` autorise la séquence à recommencer au début lorsque `valeurmax` ou `valeurmin` a été atteinte par une séquence respectivement ascendante ou descendante. Si la limite est atteinte, le prochain nombre généré sera respectivement `valeurmin` ou `valeurmax`. Si `NO CYCLE` est spécifié, tout appel à `nextval` après que la séquence a atteint la valeur minimale renverra une erreur. `NO CYCLE` est le comportement par défaut.

Utilisation d'une séquence

Bien que vous ne pouvez pas mettre à jour directement une séquence, vous pouvez toujours utiliser une requête comme :

```
SELECT * FROM nom_sequence;
```

Après la création d'une séquence, il faut utiliser les fonctions `nextval()`, `currval()` et `setval()` pour la manipuler.

`nextval('nom_sequence')` : incrémente la valeur courante de la séquence `nom_sequence` (excepté la première fois) et retourne cette valeur.

`currval('nom_sequence')` : retourne la valeur courante de la séquence `nom_sequence` ; cette fonction ne peut être appelée que si `nextval()` l'a été au moins une fois.

`setval('nom_sequence', nombre)` : Initialise la valeur courante de la séquence `nom_sequence` à `nombre`.

Vous pouvez appeler ces différentes fonctions de la manière suivante :

```
SELECT nextval('nom_sequence');
SELECT currval('nom_sequence');
SELECT setval('nom_sequence', nombre);
```

Utilisez `DROP SEQUENCE` pour supprimer une séquence.

Type SERIAL

Le type de donnée `SERIAL` n'est pas un vrai type, mais plutôt un raccourci de notation pour décrire une colonne d'identifiants uniques. Ainsi, la commande

```
CREATE TABLE nom_de_table (
    nom_de_colonne SERIAL
);
```

est équivalente à la commande :

```
CREATE SEQUENCE nom_de_sequence;
CREATE TABLE nom_de_table (
    nom_de_colonne integer DEFAULT nextval('nom_de_sequence') NOT NULL
);
```

Ainsi, nous avons créé une colonne d'entiers et fait en sorte que ses valeurs par défaut soient assignées par un générateur de séquence. Une contrainte `NOT NULL` est ajoutée pour s'assurer qu'une valeur nulle ne puisse pas être explicitement insérée. Dans la plupart des cas, on ajoute également une contrainte `UNIQUE` ou `PRIMARY KEY` pour interdire que des doublons soient créés par accident.

Pour insérer la valeur suivante de la séquence dans la colonne de type `SERIAL`, il faut faire en sorte d'utiliser la valeur par défaut de la colonne. Cela peut se faire de deux façons : soit en excluant cette colonne de la liste des colonnes de la commande `INSERT`, ou en utilisant le mot clé `DEFAULT`.

4.9.2 Règles (CREATE RULE)

Description

Le système de règles autorise la définition d'actions alternatives à réaliser sur les insertions, mises à jour ou suppressions dans les tables de la base de données. Concrètement, une règle permet d'exécuter des commandes supplémentaires lorsqu'une commande donnée est exécutée sur une table donnée. Autrement dit, une règle peut remplacer une commande donnée par une autre ou faire qu'une commande ne soit pas exécutée. Les règles sont aussi utilisées pour implémenter les vues de tables (cf. section 4.9.3).

Syntaxe de définition

Voici la syntaxe de création d'une règle :

```
CREATE [ OR REPLACE ] RULE nom AS
ON événement
TO table [ WHERE condition ]
DO [ INSTEAD ] { NOTHING | commande | ( commande ; commande ... ) }
```

`CREATE RULE` : définit une nouvelle règle s'appliquant à une table ou à une vue.

`CREATE OR REPLACE RULE` : définit une nouvelle règle, ou, le cas échéant, remplace une règle existante du même nom pour la même table.

`nom` : désigne le nom d'une règle à créer. Elle doit être distincte du nom de toute autre règle sur la même table. Lorsque plusieurs règles portent sur la même table et le même type d'événement, elles sont appliquées dans l'ordre alphabétique de leur nom.

événement : SELECT, INSERT, UPDATE ou DELETE. Les règles qui sont définies sur INSERT, UPDATE ou DELETE sont appelées des règles de mise à jour. Les règles définies sur SELECT permettent la création de vues (cf. section 4.9.3).

table : Le nom (pouvant être qualifié par le nom du schéma) de la table ou de la vue où s'applique la règle.

condition : Toute expression SQL conditionnelle (*i.e.* de type *boolean*). L'expression de condition ne peut pas référer à une table autre que NEW et OLD et ne peut pas contenir de fonction d'agrégat.

commande : Zone de spécification des commandes réalisant l'action de la règle. Les commandes valides sont SELECT, INSERT, UPDATE, DELETE ou NOTIFY. Le mot-clé NOTHING permet de spécifier que l'on ne veut rien faire.

INSTEAD : Si ce mot-clé est utilisé, la ou les commandes sont exécutées à la place de la requête déclenchante. En l'absence de INSTEAD, la ou les commandes sont exécutées après la requête déclenchante dans le cas ON INSERT (pour permettre aux commandes de voir les lignes insérées) et avant dans le cas ON UPDATE ou ON DELETE (pour permettre aux commandes de voir les lignes à mettre à jour ou à supprimer).

À l'intérieur d'une condition et d'une commande, deux tables spéciales, NEW et OLD, peuvent être utilisées pour se référer à la table sur laquelle porte la règle. NEW est valide dans les règles ON INSERT et ON UPDATE pour désigner la nouvelle ligne en cours d'insertion ou de mise à jour. OLD est valide dans les règles ON UPDATE et ON DELETE pour désigner la ligne existante en cours de modification ou de suppression.

Syntaxe de suppression

```
DROP RULE nom ON relation [ CASCADE | RESTRICT ]
```

DROP RULE : Supprime une règle de réécriture.

nom : Le nom de la règle à supprimer.

relation : Le nom (qualifié ou non du nom du schéma) de la table ou vue où s'applique la règle.

CASCADE : Supprime automatiquement les objets dépendant de la règle.

RESTRICT : Refuse de supprimer la règle si un objet en dépend. Ceci est la valeur par défaut.

4.9.3 Vues (CREATE VIEW)

Description

Les vues sont des tables virtuelles qui « contiennent » le résultat d'une requête SELECT. L'un des intérêts de l'utilisation des vues vient du fait que la vue ne stocke pas les données, mais fait référence à une ou plusieurs tables d'origine à travers une requête SELECT, requête qui est exécutée chaque fois que la vue est référencée. De ce fait, toute modification de données dans les tables d'origine est immédiatement visible dans la vue dès que celle-ci est à nouveau référencée dans une requête.

Les utilisations possibles d'une vue sont multiples :

- Cacher aux utilisateurs certaines colonnes ou certaines lignes en mettant à leur disposition des vues de projection ou de sélection. Ceci permet de fournir un niveau de confidentialité et de sécurité supplémentaire.
- Simplifier l'utilisation de tables comportant de nombreuses colonnes, de nombreuses lignes ou des noms complexes, en créant des vues avec des structures plus simples et des noms plus intelligibles.
- Nommer des requêtes fréquemment utilisées pour simplifier et accélérer l'écriture de requête y faisant référence.

Syntaxe de définition

Voici la syntaxe de création d'une vue :

```
CREATE [ OR REPLACE ] VIEW nom [ ( nom_colonne [, ...] ) ] AS requête
```

CREATE VIEW : définit une nouvelle vue.

CREATE OR REPLACE VIEW : définit une nouvelle vue, ou la remplace si une vue du même nom existe déjà. Vous pouvez seulement remplacer une vue avec une nouvelle requête qui génère un ensemble de colonnes identiques.

nom : Le nom de la vue à créer (qualifié ou non du nom du schéma). Si un nom de schéma (cf. section 4.9.4) est donné (par exemple `CREATE VIEW monschema.mavue ...`), alors la vue est créée dans le schéma donné. Dans les autres cas, elle est créée dans le schéma courant. Le nom de la vue doit être différent du nom des autres vues, tables, séquences ou index du même schéma.

nom_colonne : Une liste optionnelle de noms à utiliser pour les colonnes de la vue. Si elle n'est pas donnée, le nom des colonnes sera déduit de la requête.

requête : Une requête (c'est-à-dire une instruction `SELECT`) qui définit les colonnes et les lignes de la vue.

La norme SQL propose un ensemble important de restrictions pour la modification ou l'insertion ou la modification des données dans les vues. Les systèmes de gestion de base de données ont aussi chacun leur implantation de ce concept et chacun leurs contraintes et restrictions. En particulier, peu d'opérations sont autorisées dès qu'une vue porte sur plusieurs tables ; aucune n'est possible si la vue comporte des opérateurs d'agrégation.

Avec PostgreSQL les vues ne sont que consultables par des instructions `SELECT` (*i.e.* lecture seule). Aucune autre opération n'est possible (insertion, mise à jour ou suppression de lignes). Par contre, la notion de règles permet, avec PostgreSQL, d'implémenter ces fonctionnalités. Cette notion s'avère plus souple et puissante que les restrictions communément appliquées aux SGBD classiques.

Implémentation interne

Avec PostgreSQL, les vues sont implémentées en utilisant le système de règles. En fait, il n'y aucune différence entre

```
CREATE VIEW mavue AS SELECT * FROM matable;
```

et ces deux commandes

```
CREATE TABLE mavue (liste de colonnes identique à celle de matable);

CREATE RULE "_RETURN" AS ON SELECT TO mavue DO INSTEAD
    SELECT * FROM matable;
```

parce que c'est exactement ce que fait la commande `CREATE VIEW` en interne. Ainsi, pour l'analyseur, il n'y a aucune différence entre une table et une vue : il s'agit de relations.

Syntaxe de suppression

```
DROP VIEW nom [, ...] [ CASCADE | RESTRICT ]
```

DROP VIEW : Supprime une vue existante.

nom : Le nom de la vue à supprimer (qualifié ou non du nom du schéma).

CASCADE : Supprime automatiquement les objets qui dépendent de la vue (comme par exemple d'autres vues).

RESTRICT : Refuse de supprimer la vue si un objet en dépend. Ceci est la valeur par défaut.

4.9.4 Schémas (CREATE SCHEMA)

Description

Les schémas sont des espaces dans lesquels sont référencés des éléments (tables, vues, index...). La notion de schéma est très liée à la notion d'utilisateur ou de groupe d'utilisateurs.

Syntaxe de définition

```
CREATE SCHEMA nom_schéma
```

`CREATE SCHEMA` crée un nouveau schéma dans la base de données en cours. Le nom du schéma doit être distinct du nom des différents schémas existants dans la base de données en cours.

Le paramètre `nom_schéma` est le nom du schéma à créer.

Accès aux tables

Lorsqu'une table `nom_table` est dans un schéma `nom_schema`, pour la désigner, il faut faire précéder son nom par le nom du schéma qui la contient de la manière suivante : `nom_schema.nom_table`. C'est ce que l'on appel un nom qualifié.

Le Chemin de Recherche de Schéma

Les noms qualifiés sont pénibles à écrire et il est, de toute façon, préférable de ne pas coder un nom de schéma dans une application. Donc, les tables sont souvent appelées par des noms non qualifiés (*i.e.* nom de la table lui même). Le système détermine quelle table est appelée en suivant un chemin de recherche qui est une liste de schémas à regarder. La première table correspondante est considérée comme la table voulue. S'il n'y a pas de correspondance, une erreur est levée, même si des noms de table correspondants existent dans d'autres schémas dans la base.

Le premier schéma dans le chemin de recherche est appelé le schéma courant. En plus d'être le premier schéma parcouru, il est aussi le schéma dans lequel de nouvelles tables seront créées si la commande `CREATE TABLE` ne précise pas de nom de schéma.

Pour voir le chemin de recherche courant, utilisez la commande suivante :

```
SHOW search_path;
```

Pour ajouter un nouveau schéma `mon_schema` dans le chemin tout en conservant dans ce chemin le schéma par défaut (`public`), nous utilisons la commande :

```
SET search_path TO mon_schema,public;
```

Syntaxe de suppression

La commande `DROP SCHEMA` permet de supprimer des schémas de la base de données. La syntaxe de la commande est la suivante :

```
DROP SCHEMA nom [, ...] [ CASCADE | RESTRICT ]
```

Un schéma peut seulement être supprimé par son propriétaire ou par un super utilisateur.

`nom :` Le nom du schéma

`CASCADE :` Supprime automatiquement les objets (tables, fonctions, etc.) contenus dans le schéma.

`RESTRICT :` Refuse de supprimer le schéma s'il contient un objet. Ceci est la valeur par défaut.

4.10 Travaux Pratiques – PostgreSQL : Manipulation des nouveaux objets

4.10.1 Séquences

1. Créez une séquence `test_sequence` cyclique commençant à 10 de pas d'incrément 2 et de valeur maximum 20.
2. Testez cette séquence (avec la fonction `nextval`) et observez son comportement. Le cycle recommence-t-il à 10 ? Pourquoi ?
3. Testez également les fonctions `currval` et `setval`.
   Effacez la séquence de la table.
4. Modifiez votre fichier `GenBDCine.sql` afin que la colonne `num_individu` de la table `individu` soit du type `serial`. Rechargez votre base.
5. Tentez d'insérer un nouvel individu sans préciser son `num_individu`.
   Quel est le problème ? Comment pouvez-vous y remédier ?

4.10.2 Schéma et vues

6. Créez un schéma `vue`.
7. Dans ce schéma, créez deux vues, l'une correspondant à la liste des acteurs, l'autre à la liste des réalisateurs. Les schémas respectifs de ces relations seront :
   – *acteur(num_acteur, nom, prenom)* ;
   – *realisateur(num_realisateur, nom, prenom)*.

4.10.3 Règles

8. Créez une règle `insertion_acteur` qui insère un individu dans la table `individu` à la place de l'insérer dans la table `vue.acteur` quand on tente de l'insérer dans la table `vue.acteur`.
9. Quel est le problème de cette règle ?
10. Créez une nouvelle ligne dans la table `film`. Il s'agit d'un film fictif :

    ```
    num_film  num_realisateur titre  genre  annee
    0         0               NULL   NULL   NULL
    ```

    Quel problème rencontrez-vous ? Trouvez une solution.
11. Ainsi, quand un nouvel acteur est inséré, il est possible de mettre à jour la table `jouer` en faisant référence à ce film fictif. Corrigez votre règle `insertion_acteur` pour mettre en œuvre cette nouvelle logique. Vérifiez qu'un nouvel acteur « inséré » dans la vue `vue.acteur` apparaisse bien dans cette vue une fois l'opération effectuée.

4.10.4 Toujours des requêtes

Dans les exercices qui suivent, pour répondre, utilisez les vues `vue.acteur` et `vue.realisateur` quand cela permet de simplifier l'écriture des requêtes.

12. Quels sont les individus qui ne sont ni des acteurs, ni des réalisateurs.
13. Quels sont les noms et prénoms des acteurs qui sont également réalisateurs ?
    Remarque : cette requête a déjà été résolue en utilisant l'algèbre relationnelle (cf. travaux dirigés section 3.5.2) et le langage SQL (cf. travaux pratiques 4.6 et 4.8) :
14. Quels sont les noms et prénoms des individus dont le prénom est à la fois celui d'un acteur et celui d'un réalisateur sans qu'il s'agisse de la même personne ? Remarque : cette requête a déjà été résolue en utilisant l'algèbre relationnelle (cf. travaux dirigés section 3.5.2).

4.11 SQL intégré

4.11.1 Introduction

Ce chapitre décrit le pacquage SQL embarqué pour PostgreSQL ECPG. Il est compatible avec les langages `C` et `C++` et a été développé par Linus Tolke et Michael Meskes.

Un programme SQL embarqué est en fait un programme ordinaire, dans notre cas un programme en langage `C`, dans lequel nous insérons des commandes SQL incluses dans des sections spécialement marquées. Ainsi les instructions Embedded SQL commencent par les mots `EXEC SQL` et se terminent par un point-virgule (« ; »). Pour générer l'exécutable, le code source est d'abord traduit par le préprocesseur SQL qui convertit les sections SQL en code source `C` ou `C++`, après quoi il peut être compilé de manière classique.

Le SQL embarqué présente des avantages par rapport à d'autres méthodes pour prendre en compte des commandes SQL dans du code `C`. Par exemple, le passage des informations de et vers les variables du programme `C` est entièrement pris en charge. Ensuite, le code SQL du programme est vérifié syntaxiquement au moment de la précompilation. Enfin, le SQL embarqué en `C` est spécifié dans le standard SQL et supporté par de nombreux systèmes de bases de données SQL. L'implémentation PostgreSQL est conçue pour correspondre à ce standard autant que possible, afin de rendre le code facilement portable vers des SGBD autre que PostgreSQL.

Comme alternative au SQL intégré, on peut citer l'utilisation d'une API (*Application Programming Interface*) permettant au programme de communiquer directement avec le SGBD via des fonctions fournies par l'API. Dans ce cas de figure, il n'y a pas de précompilation à effectuer. Se référer à la documentation PostgreSQL (The PostgreSQL Global Development Group, 2005) pour plus d'information à ce sujet : *Chapitre 27. libpq – Bibliothèque C*.

4.11.2 Connexion au serveur de bases de données

Introduction

Quelque soit le langage utilisé (C, Java, PHP, etc.), pour pouvoir effectuer un traitement sur une base de données, il faut respecter les étapes suivantes :

1. établir une connexion avec la base de données ;
2. récupérer les informations relatives à la connexion ;
3. effectuer les traitements désirés (requêtes ou autres commandes SQL) ;
4. fermer la connexion avec la base de données.

Nous allons voir dans cette section comment ouvrir et fermer une connexion, et nous verrons dans les sections suivantes comment effectuer des traitements.

Ouverture de connexion

La connexion à une base de données se fait en utilisant l'instruction suivante :

```
EXEC SQL CONNECT TO cible [AS nom_connexion] [USER utilisateur];
```

La cible `cible` peut être spécifiée de l'une des façons suivantes :
- `nom_base[@nom_hôte ][:port]` ;
- `tcp:postgresql://nom_hôte [:port ] [/nom_base][? options]` ;
- `unix:postgresql://nom_hôte[: port][/nom_base ][? options]` ;
- une chaîne SQL littérale contenant une des formes ci-dessus ;
- une référence à une variable contenant une des formes ci-dessus ;
- `DEFAULT`.

En pratique, utiliser une chaîne littérale (entre guillemets simples) ou une variable de référence génère moins d'erreurs. La cible de connexion DEFAULT initie une connexion sur la base de données par défaut

avec l'utilisateur par défaut. Aucun nom d'utilisateur ou nom de connexion ne pourrait être spécifié isolément dans ce cas.

Il existe également différentes façons de préciser l'utilisateur `utilisateur` :

- `nom_utilisateur`
- `nom_utilisateur/ mot_de_passe`
- `nom_utilisateur IDENTIFIED BY mot_de_passe`
- `nom_utilisateur USING mot_de_passe`

`nom_utilisateur` et `mot_de_passe` peuvent être un identificateur SQL, une chaîne SQL littérale ou une référence à une variable de type caractère.

`nom_connexion` est utilisé pour gérer plusieurs connexions dans un même programme. Il peut être omis si un programme n'utilise qu'une seule connexion. La dernière connexion ouverte devient la connexion courante, utilisée par défaut lorsqu'une instruction SQL est à exécuter.

Fermeture de connexion

Pour fermer une connexion, utilisez l'instruction suivante :

```
EXEC SQL DISCONNECT [connexion];
```

Le paramètre `connexion` peut prendre l'une des valeurs suivantes :

- `nom_connexion`
- `DEFAULT`
- `CURRENT`
- `ALL`

Si aucun nom de connexion n'est spécifié, c'est la connexion courante qui est fermée. Il est préférable de toujours fermer explicitement chaque connexion ouverte.

4.11.3 Exécuter des commandes SQL

Toute commande SQL, incluse dans des sections spécialement marquées, peut être exécutée à l'intérieur d'une application SQL embarqué. Ces sections se présentent toujours de la manière suivante :

```
EXEC SQL instructions_SQL ;
```

Dans le mode par défaut, les instructions ne sont validées que lorsque `EXEC SQL COMMIT` est exécuté. L'interface SQL embarqué supporte aussi la validation automatique des transactions via l'instruction `EXEC SQL SET AUTOCOMMIT TO ON`. Dans ce cas, chaque commande est automatiquement validée. Ce mode peut être explicitement désactivé en utilisant `EXEC SQL SET AUTOCOMMIT TO OFF`.

Voici un exemple permettant de créer une table :

```
EXEC SQL create table individu ( num_individu integer primary key,
                                 nom varchar(64), prenom varchar(64) );
EXEC SQL COMMIT;
```

4.11.4 Les variables hôtes

Introduction aux variables hôtes

La transmission de données entre le programme `C` et le serveur de base de données est particulièrement simple en SQL embarqué. En effet, il est possible d'utiliser une variable `C`, dans une instruction SQL, simplement en la préfixant par le caractère deux-points (« `:` »). Par exemple, pour insérer une ligne dans la table `individu` on peut écrire :

```
EXEC SQL INSERT INTO individu VALUES (:var_num, 'Poustopol', :var_prenom);
```

Cette instruction fait référence à deux variables `C` nommées `var_num` et `var_prenom` et utilise aussi une chaîne littérale SQL (`'Poustopol'`) pour illustrer que vous n'êtes pas restreint à utiliser un type de données plutôt qu'un autre.

Dans l'environnement SQL, nous appelons les références à des variables `C` des **variables hôtes**.

Déclaration des variables hôtes

Les **variables hôtes** sont des variables de langage C identifiées auprès du préprocesseur SQL. Ainsi, pour être définies, les variables hôtes doivent être placées dans une section de déclaration, comme suit :

```
EXEC SQL BEGIN DECLARE SECTION;
declarations_des_variables_C
EXEC SQL END DECLARE SECTION;
```

Vous pouvez avoir autant de sections de déclaration dans un programme que vous le souhaitez.

Les variables hôtes peuvent remplacer les constantes dans n'importe quelle instruction SQL. Lorsque le serveur de base de données exécute la commande, il utilise la valeur de la variable hôte. Notez toutefois qu'une variable hôte ne peut pas remplacer un nom de table ou de colonne. Comme nous l'avons déjà dit, dans une instruction SQL, le nom de la variable est précédé du signe deux-points (« : ») pour le distinguer d'autres identificateurs admis dans l'instruction.

Les initialisations sur les variables sont admises dans une section de déclaration. Les sections de déclarations sont traitées comme des variables C normales dans le fichier de sortie du précompilateur. Il ne faut donc pas les redéfinir en dehors des sections de déclaration. Les variables qui n'ont pas pour but d'être utilisées dans des commandes SQL peuvent être normalement déclarées en dehors des sections de déclaration.

Les variables en langage C ont leur portée normale au sein du bloc dans lequel elles sont définies. Toutefois, le préprocesseur SQL n'analyse pas le code en langage C. Par conséquent, il ne respecte pas les blocs C. Aussi, pour le préprocesseur SQL, les variables hôtes sont globales : il n'est pas possible que deux de ces variables portent le même nom.

Types des variables hôtes

Seul un nombre limité de types de données du langage C est supporté pour les variables hôtes. En outre, certains types de variable hôte n'ont pas de type correspondant en langage C. Dans ce cas, des macros prédéfinies peuvent être utilisées pour déclarer les variables hôtes. Par exemple, le type prédéfini `VARCHAR` est la structure adéquate pour interfacer des données SQL de type `varchar`. Une déclaration comme

```
VARCHAR var[180];
```

est en fait convertie par le préprocesseur en une structure :

```
struct varchar_var { int len; char arr[180]; } var;
```

Utilisation d'une variable hôte : clause `INTO`

Dans le cas d'une requête de ligne unique, c'est à dire qui n'extrait pas plus d'une ligne de la base de données, les valeurs renvoyées peuvent être stockées directement dans des variables hôtes. Cependant, contrairement au langage C ou C++, le SQL est un langage ensembliste : une requête peut très bien retourner plus d'une ligne. Dans ce cas, il faut faire appel à la notion de curseur que nous abordons dans la section 4.11.7.

Dans le cas d'une requête de ligne unique, une nouvelle clause `INTO` est intercalée entre la clause `SELECT` et la clause `FROM`. La clause `INTO` contient une liste de variables hôtes destinée à recevoir la valeur de chacune des colonnes mentionnées dans la clause `SELECT`. Le nombre de variables hôtes doit être identique au nombre de colonnes de la clause `SELECT`. Les variables hôtes peuvent être accompagnées de variables indicateur afin de prendre en compte les résultats `NULL` (cf. section 4.11.5).

Lors de l'exécution de l'instruction `SELECT`, le serveur de base de données récupère les résultats et les place dans les variables hôtes. Si le résultat de la requête contient plusieurs lignes, le serveur renvoie une erreur. Si la requête n'aboutit pas à la sélection d'une ligne, un avertissement est renvoyé. Les erreurs et les avertissements sont renvoyés dans la structure SQLCA, comme décrit dans la section 4.11.6.

Par exemple, en reprenons la base de données de la séance de travaux pratiques 4.4 et une requête que nous avons déjà rencontrée section 4.7.2 : « nombre de fois que chacun des films a été projeté ». Nous

pouvons récupérer les résultats de cette requête de ligne unique dans des variables hôtes de la manière suivante :

```
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR titre[128];
  int id_film;
  int nb_proj;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT num_film, titre, COUNT(*)
  INTO :id_film, :titre, :nb_proj
  FROM film NATURAL JOIN projection
  GROUP BY num_film, titre;
```

4.11.5 Variables indicateur

Présentation

Les variables indicateur sont des variables en langage C qui fournissent des informations complémentaires pour les opérations de lecture ou d'insertion de données. Il existe plusieurs types d'utilisation pour ces variables.

Valeurs NULL : Pour permettre aux applications de gérer les valeurs NULL.

Troncature de chaînes : Pour permettre aux applications de gérer les cas où les valeurs lues doivent être tronquées pour tenir dans les variables hôtes.

Erreurs de conversion : Pour stocker les informations relatives aux erreurs.

Une variable indicateur est une variable hôte de type int suivant immédiatement une variable hôte normale dans une instruction SQL.

Utilisation de variables indicateur

Dans les données SQL, la valeur NULL représente un attribut inconnu ou une information non applicable. Il ne faut pas confondre la valeur NULL de SQL avec la constante du langage C qui porte le même nom (NULL). Cette dernière représente un pointeur non initialisé, incorrect ou ne pointant pas vers un contenu valide de zone mémoire.

La valeur NULL n'équivaut à aucune autre valeur du type défini pour les colonnes. Ainsi, si une valeur NULL est lue dans la base de données et qu'aucune variable indicateur n'est fournie, une erreur est générée (SQLE_NO_INDICATOR). Pour transmettre des valeurs NULL à la base de données ou en recevoir des résultats NULL, des variables hôtes d'un type particulier sont requises : les variables indicateur.

Par exemple, dans l'exemple précédent, une erreur est générée si, pour une raison quelconque, le titre du film n'existe pas et que sa valeur est NULL. Pour s'affranchir de ce problème, on utilise une variable indicateur de la manière suivante :

```
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR titre[128];
  int id_film;
  int nb_proj;
  int val_ind;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT num_film, titre, COUNT(*)
  INTO :id_film, :titre :val_ind, :nb_proj
  FROM film NATURAL JOIN projection
  GROUP BY num_film, titre;
```

Dans cet exemple, la variable indicateur val_ind vaudra zéro si la valeur retournée n'est pas NULL et elle sera négative si la valeur est NULL. Si la valeur de l'indicateur est positive, cela signifie que la valeur retournée n'est pas NULL mais que la chaîne a été tronquée pour tenir dans la variable hôte.

4.11.6 Gestion des erreurs

Configurer des rappels : instruction `WHENEVER`

L'instruction `WHENEVER` est une méthode simple pour intercepter les erreurs, les avertissements et les conditions exceptionnelles rencontrés par la base de données lors du traitement d'instructions SQL. Elle consiste à configurer une action spécifique à exécuter à chaque fois qu'une condition particulière survient. Cette opération s'effectue de la manière suivante :

```
EXEC SQL WHENEVER condition action;
```

Le paramètre `condition` peut prendre une des valeurs suivantes :

`SQLERROR` : L'action spécifiée est appelée lorsqu'une erreur survient pendant l'exécution d'une instruction SQL.

`SQLWARNING` : L'action spécifiée est appelée lorsqu'un avertissement survient pendant l'exécution d'une instruction SQL.

`NOT FOUND` : L'action spécifiée est appelée lorsqu'une instruction ne récupère ou n'affecte aucune ligne.

Le paramètre `action` peut avoir une des valeurs suivantes :

`CONTINUE` : Signifie effectivement que la condition est ignorée. C'est l'action par défaut.

`SQLPRINT` : Affiche un message sur la sortie standard. Ceci est utile pour des programmes simples ou lors d'un prototypage. Les détails du message ne peuvent pas être configurés.

`STOP` : Appel de exit(1), ce qui terminera le programme.

`BREAK` : Exécute l'instruction `C break`. Cette action est utile dans des boucles ou dans des instructions `switch`.

`GOTO label` et `GO TO label` : Saute au label spécifié (en utilisant une instruction `C goto`).

`CALL nom (args)` et `DO nom (args)` : Appelle les fonctions `C` spécifiées avec les arguments spécifiés.

Le standard SQL ne définit que les actions `CONTINUE` et `GOTO` ou `GO TO`.

L'instruction `WHENEVER` peut être insérée en un endroit quelconque d'un programme SQL embarqué. Cette instruction indique au préprocesseur de générer du code après chaque instruction SQL. L'effet de cette instruction reste actif pour toutes les instructions en SQL embarqué situées entre la ligne de l'instruction `WHENEVER` et l'instruction `WHENEVER` suivante contenant la même condition `condition` d'erreur, ou jusqu'à la fin du fichier source.

Les conditions d'erreur sont fonction du positionnement dans le fichier source de langage `C` et non du moment où l'instruction est exécutée.

Cette instruction est fournie pour vous faciliter le développement de programmes simples. Il est plus rigoureux de contrôler les conditions d'erreur en vérifiant directement le champ `sqlcode` de la zone SQLCA (cf. section suivante). Dans ce cas, l'instruction `WHENEVER` est inutile. En fait, l'instruction `WHENEVER` se contente de demander au préprocesseur de générer un test `if ( SQLCODE )` après chaque instruction SQL.

Zone de communication SQL (SQLCA)

La zone de communication SQL (SQLCA) est une zone de mémoire qui permet, pour chaque demande adressée à la base de données, de communiquer des statistiques et de signaler des erreurs. En consultant la zone SQLCA, vous pouvez tester un code d'erreur spécifique. Un code d'erreur s'affiche dans les champs `sqlcode` et `sqlstate` lorsqu'une requête adressée à la base de données provoque une erreur. Une variable SQLCA globale (`sqlca`) est définie dans la bibliothèque d'interface, elle a la structure suivante :

```
struct {
  char sqlcaid[8];
  long sqlabc;
  long sqlcode;
```

```
  struct {
    int sqlerrml;
    char sqlerrmc[70];
    } sqlerrm;
  char sqlerrp[8];
  long sqlerrd[6];
  char sqlwarn[8];
  char sqlstate[5];
} sqlca;
```

SQLCA couvre à la fois les avertissements et les erreurs. Si plusieurs avertissements ou erreurs surviennent lors de l'exécution d'une instruction, alors `sqlca` ne contient que les informations relatives à la dernière. Si aucune erreur ne survient dans la dernière instruction SQL, `sqlca.sqlcode` vaut 0 et `sqlca.sqlstate` vaut `"00000"`. Si un avertissement ou une erreur a eu lieu, alors sqlca.sqlcode sera négatif et sqlca.sqlstate sera différent de "00000".

Les champs `sqlca.sqlstate` et `sqlca.sqlcode` sont deux schémas différents fournissant des codes d'erreur. Les deux sont spécifiés dans le standard SQL mais `sqlcode` est indiqué comme obsolète dans l'édition de 1992 du standard et a été supprimé dans celle de 1999. Du coup, les nouvelles applications sont fortement encouragées à utiliser `sqlstate`.

4.11.7 Curseurs pour résultats à lignes multiples

Présentation

Lorsque vous exécutez une requête dans une application, le jeu de résultats est constitué d'un certain nombre de lignes. En général, vous ne connaissez pas le nombre de lignes que l'application recevra avant d'exécuter la requête. Les curseurs constituent un moyen de gérer les jeux de résultats d'une requête à lignes multiples.

Les curseurs vous permettent de naviguer dans les résultats d'une requête et d'effectuer des insertions, des mises à jour et des suppressions de données sous-jacentes en tout point d'un jeu de résultats.

Pour gérer un curseur vous devez respecter les étapes suivantes :

1. Déclarer un curseur pour une instruction `SELECT` donnée à l'aide de l'instruction `DECLARE` :

   ```
   EXEC SQL DECLARE nom_curseur CURSOR FOR requête_select ;
   ```

2. Ouvrir le curseur à l'aide de l'instruction `OPEN` :

   ```
   EXEC SQL OPEN nom_curseur ;
   ```

3. Récupérer une par une les lignes du curseur à l'aide de l'instruction `FETCH` :

   ```
   FETCH [ [ NEXT | PRIOR | FIRST | LAST | { ABSOLUTE | RELATIVE } nombre ]
           { FROM | IN } ] nom_curseur
   INTO liste_variables
   ```

   `NEXT` : Récupère la ligne suivante. Ceci est la valeur par défaut.

   `PRIOR` : Récupère la ligne précédente.

   `FIRST` : Récupère la première ligne de la requête (identique à ABSOLUTE 1).

   `LAST` : Récupère la dernière ligne de la requête (identique à ABSOLUTE -1).

   `ABSOLUTE nombre` : Récupère la `nombre`[e] ligne de la requête ou la *abs*(`nombre`)[e] ligne à partir de la fin si nombre est négatif. La position avant la première ligne ou après la dernière si nombre est en-dehors de l'échelle ; en particulier, `ABSOLUTE 0` se positionne avant la première ligne.

   `RELATIVE nombre` : Récupère la `nombre`[e] ligne ou la *abs*(`nombre`)[e] ligne avant si nombre est négatif. `RELATIVE 0` récupère de nouveau la ligne actuelle si elle existe.

   `nom_curseur` : Le nom d'un curseur ouvert.

liste_variables : La liste des variables hôtes destinées à recevoir la valeur de chacun des attributs de la ligne courante. Le nombre de variables hôtes doit être identique au nombre de colonnes de la table résultat.

4. Continuez l'extraction des lignes tant qu'il y en a.
5. Fermer le curseur à l'aide de l'instruction CLOSE :

```
CLOSE nom_curseur
```

Lors de son ouverture, un curseur est placé avant la première ligne. Par défaut, les curseurs sont automatiquement refermés à la fin d'une transaction.

Voici un exemple utilisant la commande `FETCH` :

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;
EXEC SQL OPEN foo;
while (...) {
  EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
  ...
}
```

4.11.8 Précompilation et compilation

Inclusion de fichiers

Pour inclure un fichier externe SQL embarqué dans votre programme, utilisez la commande :

```
EXEC SQL INCLUDE nom_fichier;
```

Cette commande indique au préprocesseur du SQL embarqué de chercher un fichier nommé `nom_fichier.h`, de traiter et de l'inclure dans le fichier `C` généré. Du coup, les instructions SQL embarqué du fichier inclus sont gérées correctement.

En utilisant la directive classique

```
#include <nom_fichier.h>
```

le fichier `nom_fichier.h` ne serait pas sujet au pré-traitement des commandes SQL. Naturellement, vous pouvez continuer à utiliser la directive `#include` pour inclure d'autres fichiers d'en-tête.

Précompilation et compilation

La première étape consiste à traduire les sections SQL embarqué en code source `C`, c'est-à-dire en appels de fonctions de la librairie `libecpg`. Cette étape est assurée par le préprocesseur appelé `ecpg` qui est inclus dans une installation standard de PostgreSQL. Les programmes SQL embarqué sont nommés typiquement avec une extension `.pgc`. Si vous avez un fichier programme nommé `prog.pgc`, vous pouvez le passer au préprocesseur par la simple commande :

```
ecpg prog1.pgc
```

Cette étape permet de créer le fichier `prog.c`. Si vos fichiers en entrée ne suivent pas le modèle de nommage suggéré, vous pouvez spécifier le fichier de sortie explicitement en utilisant l'option `-o`.

Le fichier traité par le préprocesseur peut alors être compilé de façon classique, par exemple :

```
cc -c prog.c
```

Cette étape permet de créer le fichier `prog.o`. Les fichiers sources en `C` générés incluent les fichiers d'entête provenant de l'installation de PostgreSQL. Si vous avez installé PostgreSQL à un emplacement qui n'est pas parcouru par défaut, vous devez ajouter une option comme `-I/usr/local/pgsql/include` sur la ligne de commande de la compilation.

Vous devez enfin lier le programme avec la bibliothèque `libecpg` qui contient les fonctions nécessaires. Ces fonctions récupèrent l'information provenant des arguments, exécutent la commande SQL en utilisant l'interface `libpq` et placent le résultat dans les arguments spécifiés pour la sortie. Pour lier un programme SQL embarqué, vous devez donc inclure la bibliothèque `libecpg` :

```
cc -o monprog prog.o -lecpg
```

De nouveau, vous pourriez avoir besoin d'ajouter une option comme `-L/usr/local/pgsql/lib` sur la ligne de commande.

4.11.9 Exemple complet

Voici un exemple complet qui effectue les opérations suivantes :
- connexion à la base ;
- vérification de la réussite de la connexion ;
- affichage du contenu de la table individu en utilisant un curseur ;
- fermeture de la connexion.

```
#include <stdio.h>
// ____ pour gérer les erreurs
EXEC SQL INCLUDE sqlca;
// ____ Définition des variables hôtes
EXEC SQL BEGIN DECLARE SECTION;
  char var_nom[256];
  char var_prenom[256];
  int  var_num;
EXEC SQL END DECLARE SECTION;

int main(void){
  // ____ Ouverture de la connexion à la base de données
  EXEC SQL CONNECT TO nom_base@aquanux;
  if(sqlca.sqlcode) {
    printf("erreur %s\n",sqlca.sqlerrm.sqlerrmc);
    exit(0);
  }
  printf(" connexion réussie \n");
  // ____ Utilisation d'un curseur pour afficher le contenu de la table individu
  EXEC SQL DECLARE curseur_individu CURSOR FOR
    SELECT num_individu, nom, prenom FROM individu;
  EXEC SQL OPEN curseur_individu;
  // Boucle d'affichage
  while(SQLCODE==0) {
    EXEC SQL FETCH FROM curseur_individu INTO :var_num, :var_nom, :var_prenom;
    printf("L'i-ndividu %d est %s %s\n", var_num, var_prenom, var_nom);
  }
  EXEC SQL CLOSE curseur_individu;
  // ____ Fermeture de connexion
  printf(" Déconnexion \n");
  EXEC SQL DISCONNECT;
  return 0;
}
```

En supposant que ce programme est enregistré dans un fichier nommé `prog.pgc`, l'exécutable est obtenu de la manière suivante :

```
ecpg prog.pgc
cc -c prog.c
cc -o prog prog.o -lecpg
```

Chapitre 5

Corrections

Bibliographie

Akoka, J. & Comyn-Wattiau, I. (2001). *Conception des bases de données relationnelles*. Vuibert informatique.

Articles en ligne sur Developpez.com. (2005a). *LE SQL de A à Z : Le simple (?) SELECT et les fonctions SQL.* (http :// sql.developpez.com/ sqlaz/ select).

Articles en ligne sur Developpez.com. (2005b). *LE SQL de A à Z : Les jointures, ou comment interroger plusieurs tables.* (http :// sql.developpez.com/ sqlaz/ jointures).

Banos, D. & Mouyssinat, M. (1990). *De merise aux bases de données*. Eyrolles.

Bourda, Y. (2005a). *Le langage SQL.* (http :// wwwlsi.supelec.fr/ www/ yb/ poly_bd/ sql/ tdm_sql.html).

Bourda, Y. (2005b). *Systèmes de Gestion de Bases de Données Relationnelles.* (http :// wwwlsi.supelec.fr/ www/ yb/ poly_bd/ poly.html).

Chen, P. (1976, March). The Entity-Relationship Model : Toward a Unified View of Data. *ACM Transactions on Database Systems*, *1*(1), 9–36.

Codd, E. F. (1970, June). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 377–387.

Crescenzo, P. (2005). *Un support de cours magistraux de Bases de données.* (http :// www.crescenzo.nom.fr/ CMBasesDeDonnees).

Date, C. (2000). *Introduction aux bases de données*. Vuibert.

Delisle, P. (2002). *8inf224 - informatique, planification et contrôle.* Support de cours. ((http :// ww-wens.uqac.ca/ ~pdelisle/ 8inf224/))

Dionisi, D. (1993). *L'essentiel sur merise*. Eyrolles.

Encyclopédie Wikipédia. (2005). *Articles en ligne sur Wikipédia.* (http ://fr.wikipedia.org).

Gabillaud, J. (2004). *SQL et algèbre relationnelle - Notions de base*. ENI.

Godin, R. (2000a). *Systèmes de gestion de bases de données* (Vol. 2). Loze-Dion.

Godin, R. (2000b). *Systèmes de gestion de bases de données* (Vol. 1). Loze-Dion.

Gruau, C. (2006). *Conception d'une base de données.* (http :// cyril-gruau.developpez.com/ uml/ tutoriel/ ConceptionBD/).

Guézélou, P. (2006). *ModÉlisation des donnÉes : Approche pour la conception des bases des données.* (http ://philippe.guezelou.free.fr /mcd /mcd.htm).

Hernandez, M. J. & Viescas, J. L. (2001). *Introduction aux requêtes SQL.* Eyrolles.

Kauffman, J., Matsik, B. & Spencer, K. (2001). *Maîtrisez SQL* (Wrox Press, Ed.). CampusPress.

Labbé, C. (2002). *Modéliser les données*. Pratiko.

Marre, D. (1996, January). *Introduction aux systèmes de gestion de bases de données.* Support de cours.

Petrucci, L. (2006). *Base de données.* Présentation projettée et travaux dirigés. (IUT GTR Villetaneuse)

Saglio, J.-M. (2002). *Dominante informatique : Module bases de données.* (http :// www.bd.enst.fr/ dombd.html).

SQL Anywhere Studio. (2005a). *ASA - Guide de programmation : Programmation avec Embedded SQL.* (http :// www.ianywhere.com/ developer/ product_manuals/ sqlanywhere/ 0901/ fr/ html/ dbpgfr9/ 00000171.htm).

SQL Anywhere Studio. (2005b). *ASA - Guide de programmation : Utilisation de SQL dans les applications.* (http :// www.ianywhere.com/ developer/ product_manuals/ sqlanywhere/ 0901/ fr/ html/ dbpgfr9/ 00000018.htm).

Szulman, S. (2005). *Base de données et SGBD.* Présentation projettée.

The PostgreSQL Global Development Group. (2005). *Documentation PostgreSQL 7.4.7.* (http :// traduc.postgresqlfr.org/ pgsql-fr/ index.html).