Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

**MASTER EN SCIENCES INFORMATIQUES**

# Reverse Engineering de Bases de Données Relationnelles SQL

de Ryckel, Nicolas; Soupart, Philippe

*Award date:*
1994

*Awarding institution:*
Universite de Namur

Link to publication

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

**Take down policy**
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

Facultés Universitaires Notre-Dame de la Paix

Institut d'Informatique
Rue Grandgagnage, 21b
B-5000 NAMUR

# Reverse Engineering de Bases de Données Relationnelles SQL

**Tome 1** : Concepts Généraux et Approche Synthétique

*Nicolas de Ryckel*
*Philippe Soupart*

Promoteur : Jean-Luc Hainaut

Mémoire présenté en vue
de l'obtention du grade de
Licencié et Maître en Informatique

Année Académique 1993-1994

# ABSTRACT

Le Reverse Engineering de Bases de Données est une activité dont l'objectif est de retrouver les spécifications techniques et fonctionnelles d'une base de données. Il s'agit de retrouver un schéma conceptuel décrivant les structures de données à partir du texte source de la base de données et des applications qui l'utilisent. La base de données considérée ici est une base de données relationnelle SQL. Cet ouvrage propose des moyens permettant de dégager des heuristiques sur le schéma conceptuel. L'originalité de l'approche proposée se situe au niveau des sources d'informations utilisées. Celles-ci sont très diverses. La source d'information dominante est l'implémentation (via des triggers ou au sein des programmes d'application) des comportements imposés par le concepteur aux données pour gérer les contraintes d'intégrité.

Retrieving technical and functional specifications of a database is the objective of DataBase Reverse Engineering. Actually, we have to retrieve a conceptual schema which describes the data structures from database and application programs source texts. The database considered here is a SQL relationnal database. This work proposes giving means which allow finding heuristics about the conceptual schema. The originality of the proposed approach comes from used information sources. These are various. The main information source is the implementation (via triggers or inside applications programs) of behaviors imposed by the designer to manage integrity constraints.

*Nous remercions vivement le professeur Jean-Luc Hainaut pour ses précieux conseils et son soutien tout au long de la réalisation de ce mémoire.*

*Nous remercions également le professeur Stefano Spaccapietra et l'équipe du Laboratoire de Bases de Données de l'Ecole Polytechnique Fédérale de Lausanne, pour leur accueil et leurs recommandations lors de notre stage. Nous remercions tout particulièrement Martin Andersson pour le temps passé à la lecture de nos textes.*

*Nous sommes également reconnaissants envers l'équipe de développement de DB-MAIN, en particulier Jean Henrard, pour leur aide concernant la partie programmation de notre travail.*

*Merci enfin à nos familles et nos amis pour leur soutien moral.*

# AVERTISSEMENT

Ce mémoire est divisé en deux tomes. Cette division se justifie par le fait que le résultat de notre travail est constitué en grande partie de développements techniques dont la lecture peut sembler longue et fastidieuse.

Le présent document consitue le premier tome de notre mémoire. Il reprend les concepts généraux utiles à la lecture du mémoire, et une présentation assez synthétique de notre travail. Il est donc destiné au lecteur qui désire connaître le résultat de nos recherches sans devoir entrer dans des considérations techniques.

Le lecteur intéressé par ces développements techniques peut consulter le tome 2.

# TABLE DES MATIERES

# TOME 1

**INTRODUCTION** .......... **1**

## PARTIE I : CONCEPTS GENERAUX

**CHAPITRE 1. MODELES SOURCE ET CIBLE** .......... **8**

1.1 LE MODELE RELATIONNEL .......... 8
1.2 LE LANGAGE SQL .......... 11
*1.2.1 Définition des structures de données* .......... *12*
*1.2.2 Consultation et extraction de données* .......... *12*
*1.2.3 Les vues* .......... *13*
*1.2.4 Modification de données* .......... *14*
*1.2.5 Les triggers* .......... *14*
*1.2.6 Les checks* .......... *15*
*1.2.7 Interface avec les programmes d'application* .......... *16*
1.3 LE MODELE ENTITE/ASSOCIATION .......... 17

**CHAPITRE 2. METHODE DE CONCEPTION DE BASES DE DONNEES** .......... **20**

2.1 LE NIVEAU CONCEPTUEL .......... 20
2.2 LE NIVEAU LOGIQUE .......... 20
*2.2.1 But* .......... *20*
*2.2.2 Passage du schéma conceptuel au schéma logique* .......... *22*
*2.2.3 Transformations* .......... *24*
2.3 LE NIVEAU PHYSIQUE .......... 25

**CHAPITRE 3. METHODE DE REVERSE ENGINEERING DE BASES DE DONNEES** .......... **27**

## PARTIE II : RECHERCHES PROPOSEES

**CHAPITRE 4. LES SOURCES D'INFORMATION** .......... **32**

4.1 PRESENTATION GENERALE .......... 32
*4.1.1 Les déclarations de tables* .......... *32*
*4.1.2 Les noms des tables et des attributs* .......... *32*
*4.1.3 Les vues* .......... *32*
*4.1.4 Les checks* .......... *32*

*4.1.5 Les comportements imposés aux données : les triggers et les requêtes* ..... *33*
*4.1.6 Les consultations des données* ..... *33*
*4.1.7 L'extension de la base de données* ..... *33*
4.2 MOYENS D'ANALYSE ..... 34
4.3 TABLEAU COMPARATIF ..... 35

**CHAPITRE 5. EXTRACTION DES STRUCTURES DE DONNEES ..... 36**

5.1 LES TYPES D'ENTITES ET LEURS ATTRIBUTS ..... 36
5.2 LES IDENTIFIANTS ..... 38
5.3 LES ATTRIBUTS FACULTATIFS ..... 40
5.4 LES DEPENDANCES FONCTIONNELLES ..... 41
5.5 LES CONTRAINTES REFERENTIELLES ..... 42
*5.5.1 Les contraintes d'inclusion* ..... *42*
*5.5.2 Les contraintes d'égalité* ..... *44*

**CHAPITRE 6. CONCEPTUALISATION DES STRUCTURES DE DONNEES ..... 45**

6.1 NETTOYAGE ET RENOMMAGE DU SCHEMA ..... 45
*6.1.1 Nettoyage* ..... *45*
*6.1.2 Renommage* ..... *46*
6.2 DE-TRADUCTION DU SCHEMA CONFORME ..... 47
*6.2.1 Les types d'associations one-to-many.* ..... *47*
6.2.1.1 Transformation d'un type d'associations one-to-many ..... 47
6.2.1.2 Recherche d'un type d'associations one-to-many ..... 48
*6.2.2 Les identifiants dont l'un des composants est un rôle* ..... *49*
*6.2.3 Les types d'entités manquants* ..... *50*
6.3 DE-OPTIMISATION D'UN SCHEMA RELATIONNEL ..... 51
*6.3.1 La redondance de dénormalisation* ..... *51*
*6.3.2 La redondance structurelle* ..... *54*
6.3.2.1 Les attributs dérivables ..... 54
6.3.2.2 La redondance entre types d'associations ..... 56
*6.3.3 Transformations de restructuration* ..... *57*
6.3.3.1 Le partitionnement horizontal ..... 58
6.3.3.2 La fusion horizontale ..... 59
6.3.3.3 Le partitionnement vertical ..... 59
6.3.3.4 La fusion verticale ..... 60
6.4 EXPRESSION DU SCHEMA DANS UN MODELE DE PLUS HAUT NIVEAU ..... 62
*6.4.1 Les types d'associations complexes* ..... *62*
6.4.1.1 Les types d'associations many-to-many ..... 62
6.4.1.2 Les types d'associations contenant un attribut ..... 64
6.4.1.3 Les types d'associations de degré supérieur à 2 ..... 64
*6.4.2 Les attributs particuliers* ..... *65*
6.4.2.1 Les attributs facultatifs ..... 65
6.4.2.2 Les attributs décomposables ..... 65
6.4.2.3 Les attributs multivalués ..... 66
*6.4.3 Les contraintes d'intégrité sur les rôles* ..... *67*
6.4.3.1 Les contraintes d'inclusion ..... 67
6.4.3.2 Les contraintes d'égalité ..... 69
6.4.3.3 Les contraintes d'exclusion ..... 69
6.4.3.4 Les contraintes d'existence ..... 71
*6.4.4 Les relations IS-A* ..... *72*
6.4.4.1 Transformation en types d'associations ..... 73
6.4.4.2 Représentation des types d'entités spécifiques ..... 75
6.4.4.3 Représentation des types d'entités génériques ..... 78

**CONCLUSION** .......... **81**

**REFERENCES** .......... **84**

# TOME 2

**CHAPITRE 1. ETAT DE L'ART**

**CHAPITRE 2. LES UNITES DE COMPORTEMENT**

**CHAPITRE 3. EXTRACTION DES STRUCTURES DE DONNEES**

**CHAPITRE 4. CONCEPTUALISATION DES STRUCTURES DE DONNEES**

**CHAPITRE 5. PROPOSITION D'UN OUTIL POUR L'ANALYSE DE TEXTES**

**ANNEXE 1. LE MODELE RELATIONNEL**

**ANNEXE 2. LA REQUETE DE SELECTION SQL**

**ANNEXE 3. LE MODELE ENTITE/ASSOCIATION**

**ANNEXE 4. REDONDANCE ENTRE TYPES D'ASSOCIATIONS MANY-TO-MANY**

# LE REVERSE ENGINEERING

Le Reverse Engineering (RE) est une activité s'inscrivant dans le cadre du software engineering. Il s'agit de la contrepartie du forward engineering.

Le forward engineering traite de la conception d'un programme, depuis ses spécifications jusqu'à son implémentation. Il existe actuellement de nombreuses méthodes de conception de programmes. Cependant, elles ne sont pas toujours (bien) utilisées. C'est particulièrement le cas pour les anciennes applications. Une des conséquences de l'absence ou de la mauvaise utilisation de méthodes de conception est un manque de documentation sur l'application. Ce manque de documentation peut aussi être dû à d'autres raisons comme la perte de celle-ci ou l'absence de mise à jour des spécifications lors d'une modification du programme. Cela pose des problèmes lorsqu'on veut faire évoluer ces applications. Une application est en effet normalement amenée à évoluer, que ce soit pour y ajouter des fonctionnalités, pour en augmenter les performances, pour corriger des erreurs, ou encore pour passer d'un Système de Gestion de Bases de Données (SGBD) à un autre. Tous ces changements nécessitent une documentation correcte sur l'application.

C'est dans ce cadre que ce situe le RE. En effet, faire du RE sur un programme (ou une partie de programme) consiste à retrouver sa documentation technique et fonctionnelle, en se basant principalement sur son texte source.

# LE REVERSE ENGINEERING DE BASES DE DONNEES

Le RE n'est pas une tâche aisée, surtout dans le cas d'applications construites sans méthode de conception. En effet, le code source de ces applications est souvent compliqué et obscur. La complexité du problème peut être diminuée dans le cas des applications orientées données, en considérant qu'on peut faire du RE de fichiers ou de Bases de Données (BD) (presque) indépendamment des parties procédurales.

En effet :

- la distance sémantique entre les spécifications conceptuelles et l'implémentation physique est plus faible pour les données que pour les parties procédurales ;
- la structure des données est généralement plus stable que les parties procédurales ;
- la structure sémantique des données est indépendante des parties procédurales ;

- il est plus facile de faire du RE des parties procédurales lorsqu'on a retrouvé la structure conceptuelle des données.

Il paraît donc logique de conceptualiser d'abord les structures de données, avant de s'intéresser aux parties procédurales. Bien que le RE des structures de données soit une tâche complexe, les concepts et techniques existants actuellement rendent cette tâche réaliste.

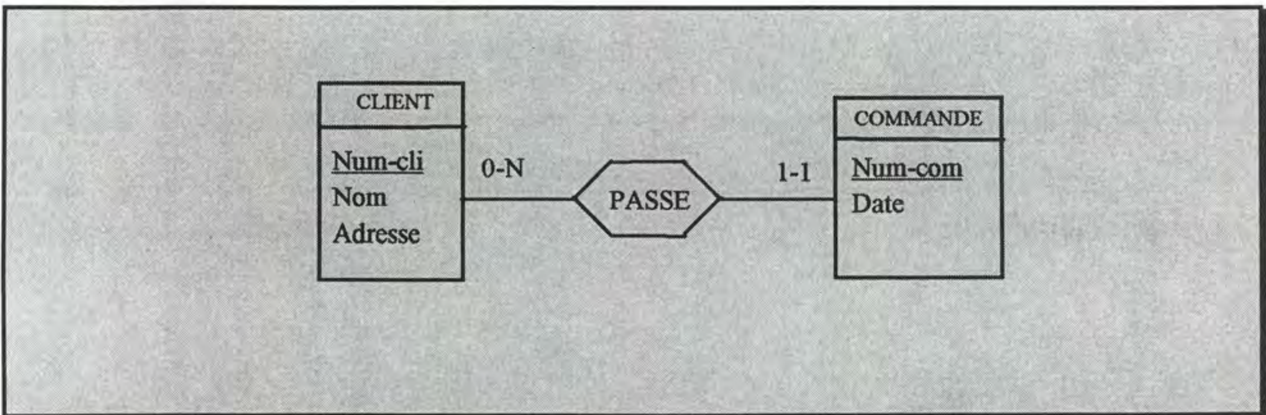
On appelle schéma conceptuel les spécifications conceptuelles d'une BD. Le RE d'une BD consiste donc à retrouver un schéma conceptuel possible à partir du texte de sa déclaration et du code source des applications qui l'utilisent.

L'exemple suivant, provenant de [PHE93] permet d'avoir une première idée de ce qu'est le reverse engineering. Considérons deux tables, `CLIENT` et `COMMANDE`, dont les déclarations sont les suivantes :

```
create table CLIENT (    num-cli .. not null,
                         nom ..,
                         adresse .. ,
                         primary key (num-cli)     )

create table COMMANDE (  num-com .. not null,
                         num-cli .. not null,
                         date ..,
                         primary key (num-com),
                         foreign key (num-cli)
                         references CLIENT         )
```

Le reverse engineering de ces déclarations de tables donnera le schéma conceptuel suivant :



Cet exemple est très simple. En effet, toutes les structures de données sont complètement exprimées dans le langage du SGBD. Dans la pratique, il en va tout autrement. Les structures conceptuelles sont généralement plus difficiles à retrouver.

# LE REVERSE ENGINEERING DE BASES DE DONNEES RELATIONNELLES

C'est dans le cadre du RE de BD relationnelles de type SQL que se situe notre mémoire.

Comme son nom l'indique, un SGBD relationnel est construit sur base du modèle relationnel. Ce modèle a été proposé par Codd en 1970 [COD70]. Il est actuellement considéré comme le modèle standard pour les SGBD commerciaux. Le langage SQL (Structured Query Language) est un langage de manipulation de BD relationnelles. Il est actuellement présent dans la plupart des SGBD relationnels. Dans notre travail, le modèle relationnel est le modèle source.

Un des modèles les plus utilisés pour décrire le schéma conceptuel d'une base de données est le modèle Entité/Association (E/A) [CHE76], [BOD89]. Ce modèle a une puissance d'expression sémantique plus grande que celle du modèle relationnel. Dans notre travail, le modèle E/A est le modèle cible.

Une première analyse du problème du RE de BD relationnelles a été réalisée dans le mémoire de Jean-François Bellem et Xavier Deflorenne [BEL93][1]. Notre mémoire s'inscrit dans la continuité de ce travail. Comme nous le verrons plus loin, l'originalité de notre travail se situe au niveau des sources d'information.

Le RE de BD relationnelles n'est pas une tâche facile, principalement pour deux raisons. La première est qu'il n'existe pas de correspondance biunivoque entre le modèle relationnel et le modèle E/A, c-à-d que pour un schéma relationnel donné, il existe N schémas E/A possibles, et vice versa. Il y a donc une **correspondance multiple** entre les deux modèles. La deuxième raison est liée aux **possibilités offertes** par les SGBD relationnels, qui ne permettent pas ou n'obligent pas la déclaration de certaines contraintes d'intégrité[2] (citons par exemple les dépendances fonctionnelles, et, dans les anciens SGBD, la déclaration des clés étrangères). Ces deux difficultés ont fortement influencé la façon dont nous avons travaillé.

### La correspondance multiple

La correspondance multiple entre le modèle relationnel et le modèle E/A amène à faire des choix tout au long du processus de RE. La nécessité de faire des choix a deux conséquences.

---

[1]Ce mémoire traitait aussi du RE de BD CODASYL.

[2]Une contrainte d'intégrité est une propriété que doivent respecter les données.

D'une part, le RE doit être vu comme une tâche interactive. En effet, les choix ne peuvent pas être totalement automatisés : ils ne font pas seulement appel à des connaissances formelles sur les BD (modélisation et conception de BD, connaissances techniques des SGBD, etc.), mais aussi à des connaissances sur les habitudes de programmation, sur le domaine des applications, etc. Un outil informatique de RE doit donc être interactif[3]. Dans notre travail, nous proposons des recherches permettant de découvrir des indices qui doivent alors être analysés par l'utilisateur.

D'autre part, le processus de RE doit être divisé en étapes, afin de permettre de faire les choix au fur et à mesure de la recherche des structures conceptuelles. La méthode de RE sur laquelle nous nous basons est celle développée à l'Institut d'Informatique [HAI93a], [PHE93]. Elle se compose de deux phases principales : l'extraction des structures de données et la conceptualisation des structures de données. La première phase donne, par une analyse du code source des schémas et des applications, un modèle des structures physiques de données conforme au modèle du SGBD utilisé (dans notre cas, le modèle relationnel). La deuxième phase aboutit à un schéma conceptuel clair, normalisé et naturel, par une série de transformations sur le schéma obtenu suite à la phase d'extraction. Nous pensons que cette découpe permet de réduire le problème de la multiplicité de traductions.

**Possibilités offertes pour la gestion des contraintes d'intégrité**

La façon la plus logique de déclarer une contrainte d'intégrité est de le faire explicitement dans le schéma de la base de données. Cependant, les SGBD ne permettent pas de déclarer toutes les contraintes. De plus, on n'est pas obligé de les déclarer.

Pour retrouver toutes les contraintes d'intégrité, il faut donc considérer d'autres sources d'information que la déclaration du schéma de la BD. Nous avons considéré comme sources d'information supplémentaires les noms des tables et de leurs attributs, la déclaration des vues, des checks, et des triggers, les requêtes des programmes d'application et le contenu de la BD (extension de la BD).

# BREF APERÇU DE L'ETAT DE L'ART

Bien que le RE de BD n'aie pas encore suscité un grand intérêt dans la communauté scientifique des BD, il existe cependant quelques travaux intéressants.

Citons par exemple :

- RE de fichiers standards : [NIL85] , [DAV85] , [PHE93] ;
- RE de BD IMS : [NAV88] , [WIN90] ;

[3]DB-MAIN, l'atelier de développement et de maintenance de bases de données développé à l'Institut, est un outil interactif.

- RE de BD CODASYL : [BAT92] , [BEL93] ;
- RE de BD relationnelles : [DAV88] , [NAV88] , [JOH89] , [FON92] , [PRE93] , [CAS93], [AND94].

On peut faire divers reproches à la plupart de ces travaux. Citons par exemple :

- la traduction est généralement directe (pas de prise en compte de représentations non classiques) ;
- les optimisations ne sont pas toujours prises en compte ;
- les dépendances fonctionnelles et d'inclusion sont souvent supposées connues ;
- certaines méthodes font l'hypothèse que les noms ont été choisis de manière rationnelle (par exemple, une clé étrangère et l'identifiant référencé ont le même nom) ;
- les identifiants sont parfois supposés connus.

Nous évoquerons en détail ces travaux dans le tome 2, où nous donnerons un état de l'art plus complet, avec des comparaisons des différentes méthodes auxquelles nous nous sommes intéressés.

# ARCHITECTURE DU TRAVAIL

Ce mémoire est divisé en deux tomes.

Le premier tome est destiné au lecteur qui désire avoir une idée de notre travail, sans s'intéresser à tous les aspects techniques, repris dans le deuxième tome. Cette division se justifie par le fait que le travail que nous avons réalisé comporte beaucoup de développements exhaustifs (par exemple, nous citons beaucoup de façons de gérer les contraintes d'intégrité). Le deuxième tome, même s'il n'a pas tout à fait la même structure que le tome 1, peut être vu comme un développement très détaillé de celui-ci. Il ne reprendra pas tous les points du tome 1. En effet, nous ne faisons pas de développements techniques pour toutes les recherches que nous proposons. Il y aura cependant quelques fois des petites répétitions dans le tome 2, afin de ne pas obliger le lecteur à lire les deux tomes "en parallèle".

Le tome 1 se compose de deux parties. La partie 1 reprend les concepts généraux utiles à une bonne compréhension de notre travail. On y trouvera une brève définition du modèle relationnel, du langage SQL et du modèle Entité/Association, une méthode de conception de bases de données, et la méthode de reverse engineering sur laquelle nous nous sommes basés. La partie 2 contient les aspects essentiels de nos recherches. Nous y ferons de nombreuses références au tome 2 pour des développements plus complets.

Le tome 2 reprend un état de l'art plus complet, les développements techniques des recherches que nous proposons, et les annexes.

# PARTIE I
# CONCEPTS GENERAUX

Cette partie reprend les concepts utiles à une bonne compréhension de notre travail. On y trouvera une définition des principaux concepts du modèle relationnel, du langage SQL et du modèle E/A. On y verra aussi une méthode de conception de bases de données et une méthode de reverse engineering.

# CHAPITRE 1
# MODELES SOURCE ET CIBLE

Nous allons examiner ici le modèle relationnel, le langage SQL et le modèle Entité/Association. Nous les présenterons brièvement en nous basant sur des exemples. On trouvera dans les annexes un développement plus détaillé du modèle relationnel (Annexe 1), de la commande SQL `SELECT` (Annexe 2) et du modèle Entité/Association (Annexe 3).

## 1.1 LE MODELE RELATIONNEL

Cette partie est inspirée de [HAI89b] et [BEL93].

Le modèle relationnel est proposé pour la première fois par Codd en 1970 [COD70]. Il s'agit d'un modèle original de représentation de l'information puisqu'il se base sur le concept mathématique de relation. Ainsi, il permet de représenter la réalité de façon simple et formelle.

Ce modèle a suscité un intérêt sans cesse croissant, non seulement dans la communauté scientifique, mais également dans le monde des entreprises puisque les SGBD qui s'en sont inspirés (c-à-d les **SGBD relationnels**) sont parmi les plus utilisés aujourd'hui.

On peut diviser ce modèle en 3 parties : une partie structurelle décrivant les **concepts de base** du modèle, une partie composée des **contraintes d'intégrité** imposée aux données, et une partie de manipulations reprenant les différents **opérateurs** applicables aux données et à leurs structures.

Les concepts de base sont les concepts de relation, d'attribut, de domaine, de clé et de clé étrangère. Les principales contraintes d'intégrité sont les identifiants, les dépendances fonctionnelles et les contraintes d'inclusion. Parmi les opérateurs les plus fréquemment utilisés sont la projection et la jointure.

Nous donnerons dans le tome 2 une définition formelle de tous ces concepts. Nous nous contentons pour l'instant de les illustrer en nous basant sur l'exemple de la figure 1.1.

Cet exemple décrit un schéma relationnel composé de la **relation** `VEND`. Cette relation associe deux **attributs** : `MAGASIN` et `PRODUIT`. Les **valeurs** de `MAGASIN` sont `(SARMA,COLRUYT,GB)` et les **valeurs** de `PRODUIT` sont `(SUCRE,FARINE, CONSERVES)`. Une **ligne** est composée de valeurs (une valeur pour chaque attribut de la re-

lation). On ne trouvera jamais deux lignes identiques, car le modèle relationnel est un modèle ensembliste[4]. Le contenu des lignes d'une relation est appelé **extension** de cette relation.

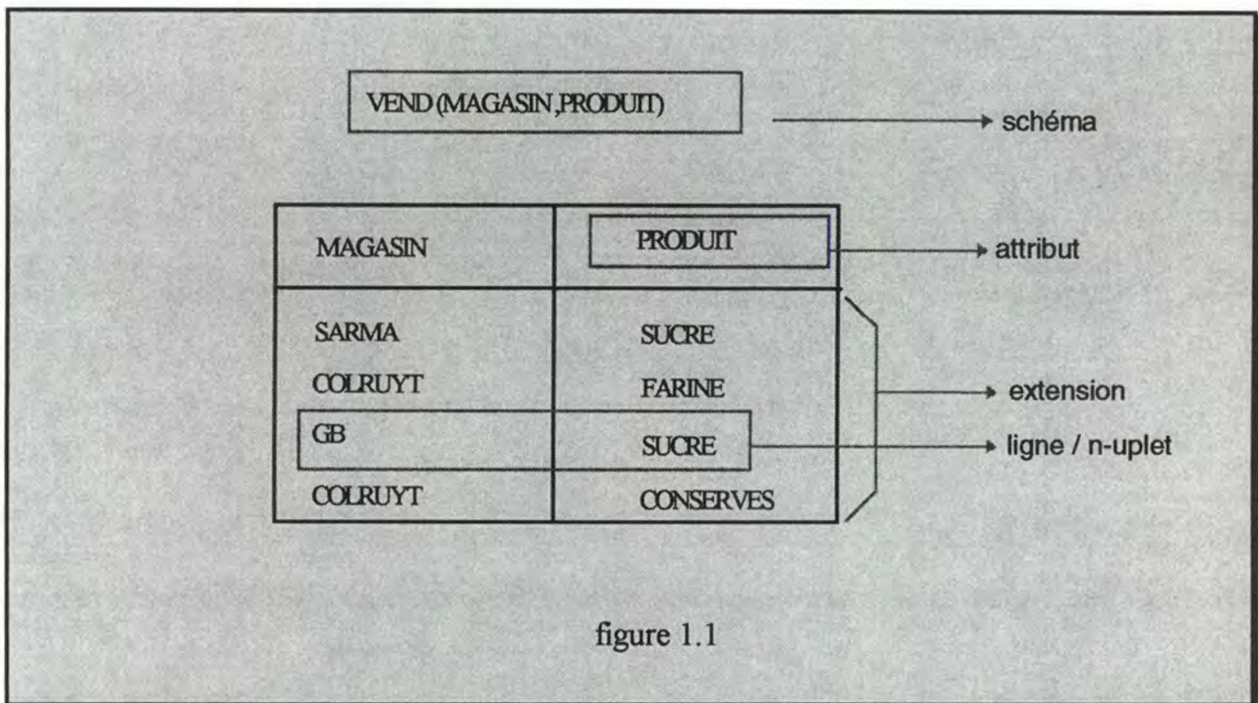


figure 1.1

Ce schéma est composé d'une seule relation. En général, un schéma sera composé de plusieurs relations. On pourrait par exemple avoir le schéma de la figure 1.2.

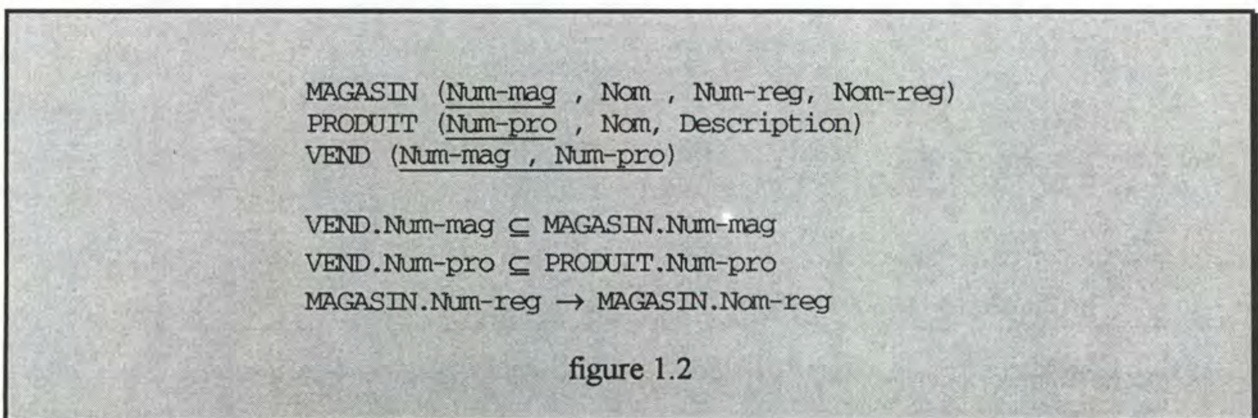


figure 1.2

Dans ce schéma on trouve 3 **relations**: VEND , PRODUIT, MAGASIN.

[4]Un schéma relationnel est donc un ensemble de relations, une relation est un ensemble de lignes, et une ligne est un ensemble de valeurs d'attributs.

« Un **identifiant** d'une relation est un attribut ou une liste d'attributs telle qu'il ne peut se trouver dans cette relation deux lignes qui possèdent les mêmes valeurs pour les attributs de cette liste » [HAI89b][5]. Les identifiants de chaque relation sont soulignés. Si une relation possède plusieurs identifiants *stricts*[6], un de ceux-ci est désigné comme identifiant *primaire*. Les autres sont les identifiants *candidats*.

On voit qu'il y a une **contrainte d'inclusion** entre VEND.Num-mag et MAGASIN.Num-mag. Cela signifie que chaque valeur de VEND.Num-mag trouve son équivalent dans l'ensemble des valeurs de MAGASIN.Num-mag. On peut dire qu'une valeur de VEND.Num-mag permet de 'désigner' une ligne de MAGASIN (la ligne désignée étant la ligne qui a comme valeur d'identifiant la valeur de VEND.Num-mag). Lorsque le côté droit d'une contrainte d'inclusion est un identifiant, on appelle celle-ci une **contrainte référentielle**. L'attribut Num-mag de VEND est appelé **clé étrangère**. Il en va de même pour l'attribut Num-pro de VEND. Notons qu'une contrainte référentielle peut aussi prendre la forme d'une **contrainte d'égalité** (correspondant à deux contraintes d'inclusion réciproques).

Il y a une **Dépendance Fonctionnelle** (DF, notée →) de MAGASIN.Num-reg vers MAGASIN.Nom-reg. Cela signifie qu'à une valeur donnée de Num-reg est toujours associée la même valeur de Nom-Reg (en pratique, cela signifie qu'à un numéro de région correspond toujours le même nom de région).

La **projection** d'une relation est la relation obtenue en ne conservant de la relation initiale que les valeurs de certains attributs. Par exemple PRODUIT[Nom] est la projection de la relation PRODUIT sur l'attribut Nom On utilisera aussi la notation PRODUIT.Nom

La **jointure** de deux relations permet de construire une relation en couplant les n-uplets de deux relations qui ont une même valeur pour un ou plusieurs attributs compatibles. Notations : VEND(Num-pro) * PRODUIT(Num-pro) ou de façon plus concise VEND * PRODUIT (Num-pro).

[5]Comme on ne trouve jamais deux lignes identiques pour une relation, tous les attributs d'une relation constituent un identifiant pour celle-ci.

[6]Un identifiant strict est un identifiant tel que si on lui retire un attribut, il ne constitue plus un identifiant. Les identifiants stricts sont les seuls identifiants intéressants.

# 1.2 LE LANGAGE SQL

Cette partie est inspirée de [HAI92].

SQL (Structured Query Language) est un langage de requêtes proposé par la majorité des SGBD relationnels. Il a été proposé pour la première fois en 1973 par une équipe de chercheurs d'IBM.

SQL propose une série d'instructions (appelées requêtes) permettant la définition des structures des données, la consultation des données et la mise à jour de celles-ci (ajout, suppression, modification). Il offre aussi la possibilité de déclarer des *vues*, des *checks* et des *triggers*. On peut exécuter ces instructions interactivement ou à l'intérieur des programmes d'application. Avant d'examiner ces instructions, nous allons préciser certains termes :

- en SQL une relation est appelée une table, un attribut une colonne, et un identifiant une clé ;
- on appelle texte DDL (Data Description Language) le texte reprenant l'ensemble des requêtes SQL définissant une BD (tables, vues, triggers ,checks, etc.) ;
- toute base de données SQL comporte une collection de tables dont le contenu décrit les structures de cette base. Ces tables constituent le catalogue du système et sont appelées tables système. Il est possible de retrouver le texte DDL de la BD à partir de ce catalogue.

Les exemples dont nous nous servirons pour expliquer les principales instructions de SQL se basent sur le schéma relationnel de la figure 1.3.

```
CLIENT (Num-cli , Nom , Adresse, Num-tel, Debit-credit , Num-cat)
CATEGORIE-CLIENT (Num-cat, Region, Salaire-moyen)
CLIENT.Num-cat ⊆ CATEGORIE-CLIENT.Num-cat
```

figure 1.3

Ce schéma est composé d'une table `CLIENT`, contenant le numéro du client, son nom, son adresse, son solde débiteur/créditeur, son numéro de catégorie et son numéro de téléphone. Le numéro de catégorie référence la table `CATEGORIE-CLIENT`, qui représente les catégories de clients. Dans une catégorie de clients, on trouve le numéro de catégorie, la région et le salaire moyen des clients.

### 1.2.1 Définition des structures de données

Considérons la création de la table CLIENT :

```
CREATE TABLE CLIENT (Num-cli INTEGER NOT NULL,
                     Nom CHAR (30) NOT NULL,
                     Adresse CHAR (60),
                     Num-tel CHAR(20) NOT NULL,
                     Debit-credit FLOAT NOT NULL,
                     Num-cat INTEGER NOT NULL,

                     PRIMARY KEY (Num-cli),
                     UNIQUE(Nom, Adresse),
                     FOREIGN KEY(num-cat) REFERENCES CATEGORIE-CLIENT
```

La commande de création d'une table est CREATE TABLE, suivie du nom de la table avec en arguments les colonnes de celle-ci et leur type. Un identifiant primaire est déclaré par la clause PRIMARY KEY, un identifiant candidat par la clause UNIQUE. Notons que dans les anciens SGBD, cette possibilité de déclaration des identifiants n'existait pas. Il fallait alors déclarer un "UNIQUE INDEX" (un index est un mécanisme permettant d'accéder plus rapidement aux données).

```
CREATE UNIQUE INDEX x-cli
ON CLIENT (Num-cli)
```

La déclaration d'une contrainte référentielle peut être faite dans les SGBD récents grâce à la clause FOREIGN KEY. Si on n'a pas cette possibilité, il faut gérer cette contrainte autrement (par exemple par trigger ou par check, comme nous le verrons plus loin).

Dans le modèle relationnel de CODD tout attribut est obligatoire. Les SGBD SQL permettent de simuler les attributs facultatifs à l'aide d'une valeur particulière : la valeur NULL (représentant une valeur d'attribut inconnue ou absente). Par défaut toute colonne est facultative. Le caractère obligatoire d'une colonne est déclaré par la clause NOT NULL.

### 1.2.2 Consultation et extraction de données

La commande SELECT permet d'extraire des données de l'ensemble des tables afin de les présenter à l'utilisateur final ou à un programme d'application qui en a demandé l'exécution. Nous allons présenter les grands principes de cette commande.

Le SELECT produit un résultat qui est une table. Il est composé de trois parties :

- la clause SELECT donne les valeurs constituant chaque ligne du résultat ;

- la clause `FROM` fournit les tables mise en cause par l'extraction ;
- la clause `WHERE` précise les conditions de sélection que doivent satisfaire les lignes du résultat.

Il est en outre possible d'imbriquer une sous-requête dans une condition de sélection.

Exemple de requête:

```
SELECT Nom
FROM CLIENT
WHERE Num-cat IN (SELECT Num-cat
                  FROM CATEGORIE-CLIENT
                  WHERE REGION = 'BRABANT')
```

Cette commande sélectionne le nom des clients habitant la région du Brabant (`Region = 'BRABANT'`).

## 1.2.3 Les vues

Une vue est une table virtuelle, construite à partir de tables existantes ou d'autres vues. Le contenu d'une vue n'est pas stocké physiquement, seule sa définition l'est. Une vue peut être utilisée comme une table 'normale' pour l'extraction de données. Dans certains cas, on peut modifier les données à partir d'une vue (le SGBD répercute alors les modifications sur les tables réelles).

On utilise les vues pour adapter la présentation des données aux besoins de l'utilisateur, pour restreindre l'accès aux données à l'utilisateur, ou pour protéger l'utilisateur d'une modification de la structure de la BD en créant des vues qui offrent à l'utilisateur la même perception des données qu'avant cette modification.

Pour déclarer une vue, on utilise la commande `CREATE VIEW` suivie du nom des colonnes virtuelles et d'une requête définissant les données reprises dans la vue.

```
CREATE VIEW CLIENT-LUX (Num-cli, Nom-cli, Adresse, Debit-credit)
AS SELECT Num-cli, Nom-cli, Adresse, Debit-credit
FROM CLIENT CLI, CATEGORIE-CLIENT CAT
WHERE CAT.Num-cat = CLI.Num-cat AND CAT.REGION = 'LUXEMBOURG'
```

Cette vue reprend le numéro, le nom, l'adresse et le solde débiteur/créditeur des clients de la région du Luxembourg.

## 1.2.4 Modification de données

Pour insérer une ligne on utilise l'instruction INSERT...VALUES.

```
INSERT INTO CLIENT VALUES (191,'DUPONT', '3, rue de Bruxelles, Namur',
                           4532, '081-67-54-34', 7)
```

Il est également possible d'insérer des données provenant d'une ou plusieurs tables par l'instruction INSERT INTO nom-table suivi d'une requête d'extraction.

Pour supprimer une ligne on utilise l'instruction DELETE. Les lignes à supprimer sont désignées de la même façon que pour la condition de sélection de l'instruction SELECT.

```
DELETE FROM CLIENT
WHERE Num-cli = 191
```

Pour modifier certaines lignes qui vérifient une condition, on utilise l'instruction UPDATE...SET.

```
UPDATE CLIENT
SET Debit-credit = Debit-credit * 0.05
WHERE Debit-credit >= 50000
```

Cette instruction ajoute 5% d'intérêts aux clients dont le solde est supérieur à 50000.

## 1.2.5 Les triggers

Un trigger est une construction particulière qui provoque le déclenchement automatique d'une action dès qu'une opération (appelée opération de déclenchement) INSERT, DELETE, ou UPDATE est effectuée sur une table donnée (appelée table de déclenchement). Grâce aux triggers il est possible de définir des actions telles que :

- la vérification d'une contrainte à laquelle doit satisfaire la BD au moment du déclenchement. Nous appellerons ce genre de trigger un trigger de vérification ;
- une modification de la BD. Nous appellerons ce genre de trigger un trigger d'action.

Il n'y a actuellement pas beaucoup de standardisation au niveau des triggers. Chaque SGBD commercial a sa terminologie, sa syntaxe, et ses possibilités. Dans la suite nous avons utilisé les triggers SYBASE pour leur simplicité et leur souplesse. En voici les principales caractéristiques :

- l'action d'un trigger SYBASE est toujours exécutée après l'opération de déclenchement (contrairement aux triggers RDB où le concepteur a le choix entre une exécution avant ou après l'opération de déclenchement) ;
- l'action d'un trigger SYBASE n'est exécutée qu'une seule fois lors du déclenchement d'un trigger (en RDB, l'action d'un trigger peut être executée une fois par ligne faisant l'objet de l'opération de déclenchement) ;
- l'action peut comporter une ou plusieurs instructions `IF THEN ELSE` (l'argument du `IF` étant un prédicat). Ces instructions peuvent être imbriquées ;
- les lignes mises en cause lors de l'opération de déclenchement du trigger peuvent être accédées grâce à des tables système. Aux anciennes valeurs (lors d'un `DELETE` ou d'un `UPDATE`) correspond la table système `deleted` et aux nouvelles valeur (lors d'un `INSERT` ou d'un `UPDATE`) correspond la table système `inserted`.

Voici un exemple de trigger Sybase.

```
CREATE TRIGGER supprimer-client
ON CATEGORIE-CLIENT FOR DELETE
AS
BEGIN
   DELETE FROM deleted , CLIENT
   WHERE deleted.Num-cat = CLIENT.Num-cat)
END
```

Ce trigger est déclenché à chaque opération de suppression d'une catégorie de clients. Il provoque la suppression des clients de cette catégorie.

## *1.2.6 Les checks*

Un check permet de s'assurer qu'à tout moment, la base de données respecte une certaine contrainte.

Voici un exemple un check vérifiant la contrainte d'inclusion entre `CLIENT.Num-cat` et `CATEGORIE-CLIENT.Num-cat` :

```
CREATE TABLE CLIENT
(...,
CONSTRAINT reference-cat
CHECK (Num-cat IN (SELECT Num-cat
                   FROM CATEGORIE-CLIENT))
```

## 1.2.7 Interface avec les programmes d'application

On peut intégrer n'importe quelle commande SQL dans un programme d'application. Un protocole spécifique existe pour les langages de la troisième génération.

Voici les principales composantes de ce protocole.

En COBOL, pour insérer une commande, il suffit de l'entourer des déclarations `EXEC SQL` et `END-EXEC`.

On peut utiliser des variables du programme d'application dans une commande SQL en les faisant précéder du caractère ':'.

Voyons maintenant le problème de l'extraction de données.

- Lorsqu'on est certain qu'une requête ne donnera toujours qu'une seule ligne, on peut utiliser l'instruction `SELECT... INTO... FROM...WHERE` qui est identique au `SELECT` habituel mais dans laquelle la clause `INTO` permet de spécifier les variables du programme d'application qui recevrons le résultat.
- Dans le cas contraire, le nombre de lignes est indéterminé à l'avance. Il faut donc les traiter en séquence. Il faut tout d'abord définir un curseur par la commande `DECLARE CURSOR nom-curseur FOR` suivie de la requête qui nous intéresse. Le résultat de l'exécution de cette requête est déclenché par l'instruction `OPEN nom-curseur`. Ce résultat peut être considéré comme un fichier séquentiel. Pour extraire les lignes du résultat les unes après les autres on utilise la commande `FETCH nom-curseur INTO` suivie des variables du programme d'application qui vont recevoir le résultat. Une fois le travail terminé, on ferme le curseur par `CLOSE nom-curseur`.

A tout moment on peut utiliser l'indicateur de diagnostic `SQLCODE` qui contient une valeur permettant de connaître la manière dont s'est déroulée une commande. Par exemple, `SQLCODE = 0` signifie que tous s'est déroulé correctement et `SQLCODE = 100` signifie qu'aucune ligne n'a été trouvée.

# 1.3 LE MODELE ENTITE/ASSOCIATION

Le modèle Entité/Association (E/A) a été proposé par Chen en 1976 [CHE76]. Le modèle E/A que nous utilisons est quelque peu différent. Il est constitué de la plupart des concepts présents dans [BOD89] et de certains concepts présents dans [HAI89a]. Ce modèle permet de représenter les structures conceptuelles sous-jacentes aux données, indépendamment de leur représentation physique et de leur utilisation. Il permet de représenter la réalité sous forme d'objets qui ont des propriétés et qui sont associés les uns aux autres. On trouve dans le modèle E/A que nous utilisons les concepts d'**entités**, d'**attributs**, d'**associations** et de **spécialisation/généralisation**. On y trouve aussi des contraintes d'intégrité, qui sont des règles auxquelles les données doivent satisfaire.

Nous donnerons une définition précise et assez formelle des concepts du modèle E/A dans le tome 2. Pour l'instant, nous préférons donner une brève description des principaux concepts, en nous basant sur l'exemple de la figure 1.4.

Une entité représente un objet du monde réel. Des entités similaires sont regroupées en un **type d'entités (T.E.)**. Un T.E. comprend 0 ou plusieurs attributs. Dans l'exemple, `CLIENT`, `FOURNISSEUR`, `PARTICULIER` et `ENTREPRISE` sont des T.E.

Un **attribut** représente une caractéristique propre du T.E. auquel il appartient. Un attribut est *obligatoire* s'il a une valeur pour chaque occurrence de son T.E. Autrement, il est *facultatif* .Un attribut est *monovalué* si au plus une valeur de cet attribut peut être associée a une entité. Si plusieurs valeurs d'un attribut peuvent être associées à une entité, il est *multivalué*. Un attribut est *décomposable* si ses valeurs peuvent être décomposées en éléments significatifs. Sinon, il est *élémentaire*.

Une **association** relie des entités. Les associations d'un même type sont regroupées en **types d'association (T.A.)**. Un T.A. relie donc des types d'entités. Si le T.A. relie deux T.E., on parle de T.A. *binaire*, s'il en relie trois on parle de T.A. *ternaire*, etc. Le nombre de T.E. reliés à un T.A. est le *degré* de ce T.A.. Si le T.A. relie un T.E. à lui-même, on parle de T.E. *récursif* ou *cyclique*. Les T.A. peuvent contenir des attributs.

Une entité reliée à une ou plusieurs autres entités via une association joue un ou plusieurs **rôle(s)** dans cette association. Les **connectivités** d'un rôle permettent de définir le nombre de rôles que jouent les entités. Il y a une connectivité minimale et une connectivité maximale sur chaque rôle. La connectivité minimale du rôle d'un T.E. permet de définir le nombre minimum de rôles que doivent jouer les entités de ce type . Les valeurs les plus fréquemment utilisées sont 0 et 1. La connectivité maximale permet de définir le nombre maximum de rôles que peuvent jouer les entités d'un certain type. Les valeurs les plus fréquentes sont 1 et N (N représentant un nombre quelconque de fois).

Pour les T.A. binaires, on utilise un vocabulaire simplifié : (i représente soit 0 soit 1)

- T.A. one-to-many : l'un des deux rôles est de connectivité i-1 ;
- T.A. one-to-one : les deux rôles sont de connectivité i-1 ;
- T.A. many-to-many : aucun rôle n'est de connectivité i-1.

Un T.E. peut avoir un ou plusieurs **identifiants**. Un identifiant permet de reconnaître univoquement une entité d'un certain type. Il est constitué d'un ou de plusieurs attributs et/ou d'un ou de plusieurs rôles[7]. Dans le cas d'un identifiant constitué d'attributs et de rôles on parle d'identifiant hybride.

Un T.A. est généralement identifié par l'ensemble de ses rôles. Dans certains cas, des attributs du T.A. peuvent faire partie de son identifiant.

On peut trouver des relations de **spécialisation/généralisation**, encore appelées relations **IS-A** entre des types d'entités. Dans l'exemple, on trouve deux relations IS-A, entre `CLIENT` et `PARTICULIER` et entre `CLIENT` et `ENTREPRISE`. Cela signifie qu'un particulier (de même qu'une entreprise) est aussi un client. Les types d'entités `ENTREPRISE` et `PARTICULIER` ont les mêmes caractéristiques (attributs, rôles, etc.) que le type d'entités `CLIENT` (mécanisme d'héritage), et ils peuvent avoir leurs propres caractéristiques (leurs propres attributs, rôles, etc.). Les deux contraintes les plus intéressantes qui peuvent porter sur une relation IS-A sont les contraintes d'**exclusion** (à une instance du sur-type ne peut correspondre qu'une instance d'un seul des sous-types) et de **couvrance** (à une instance du sur-type doit correspondre au moins une instance de l'un des sous-types). Lorsqu'on a à la fois la couvrance et l'exclusion, on dit qu'on a une structure de **partition**.

Des **contraintes d'intégrité** peuvent être associées à ces constructions. Mentionnons les contraintes d'inclusion, d'exclusion, d'égalité entre attributs et/ou rôles, les contraintes d'existance et les dépendances fonctionnelles (voir infra).

[7]Un type d'entités peut être identifié par les rôles assumés par des types d'entités dans les types d'associations auxquels il participe.

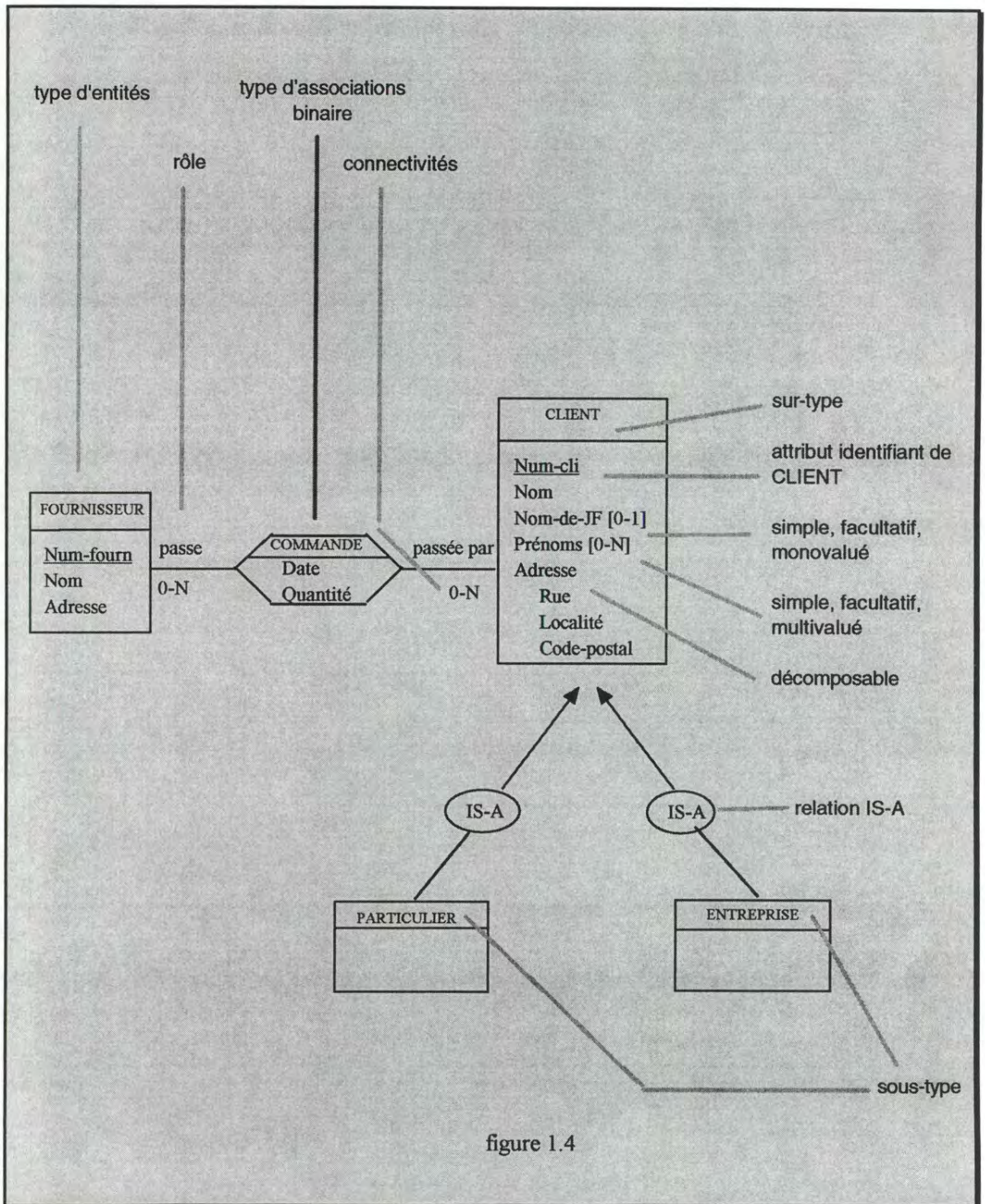



figure 1.4

# CHAPITRE 2
# METHODE DE CONCEPTION DE BASES DE DONNEES

Une bonne compréhension du processus de reverse engineering passe nécessairement par une bonne compréhension du processus de conception d'une base de données. La méthode de conception que nous allons brièvement présenter ici est celle développée dans [HAI92]. Elle se compose de trois processus, appellés niveaux : le niveau conceptuel, logique et physique.

La figure 2.1 donne un aperçu de la méthode de conception.

## 2.1 LE NIVEAU CONCEPTUEL

Le niveau conceptuel aboutit à un schéma conceptuel (sous forme Entité/Association) du système d'information que l'on construit. Ce schéma conceptuel sera utilisé pour définir le schéma de la BD. Etant donné que le but du reverse engineering de BD est d'obtenir ce schéma conceptuel, il ne nous semble pas utile de voir en détail de quelle façon on aboutit à ce schéma lors du processus de conception. Mentionnons en seulement les principales étapes.

- découpe du réel perçu en sous-systèmes et construction par enrichissement progressif d'un schéma conceptuel pour chaque sous-système ;
- normalisation de ces schémas ;
- intégration des schémas.

## 2.2 LE NIVEAU LOGIQUE

Nous allons d'abord voir le but du niveau logique : obtenir un schéma logique. Ensuite, nous verrons les principales étapes permettant de passer d'un schéma conceptuel à un schéma logique. Enfin, nous aborderons le concept de transformation.

### *2.2.1 But*

Le but du niveau logique est d'obtenir un **schéma optimisé et conforme au modèle de la base de données visée** (dans notre cas, le modèle relationnel). Il faut bien entendu que l'on passe du schéma conceptuel au schéma logique sans perte de sémantique (c-à-d que le schéma logique

doit exprimer la même chose que le schéma conceptuel). On utilisera le modèle E/A pour représenter le schéma logique.

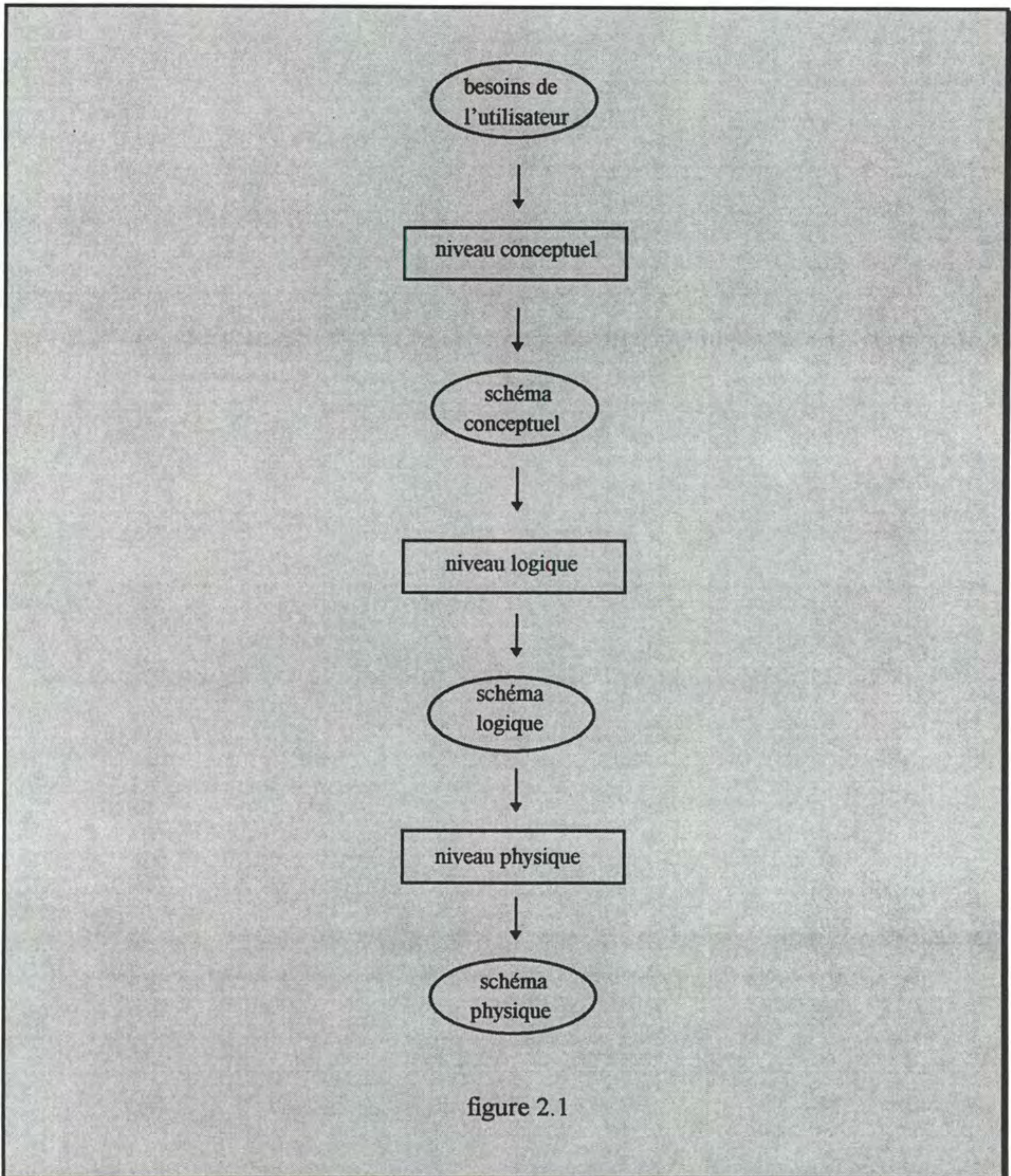



figure 2.1

**« schéma optimisé »**

Lors de la conception d'une base de données, on peut être amené à optimiser le schéma relationnel, pour deux raisons : diminuer la taille des structures de données et diminuer le temps d'accès[8].

**« schéma conforme au modèle de la base de données visée »**

Par "schéma conforme au modèle de la base de données visée", nous voulons dire que le schéma doit se conformer aux possibilités offertes par ce modèle. Il y aura donc des restrictions par rapport au schéma conceptuel (le modèle relationnel a en effet une puissance d'expression sémantique plus faible que celle du modèle E/A).

Un schéma conforme au modèle relationnel contient des types d'entités, des attributs, des identifiants, des contraintes référentielles. Voici les principales restrictions par rapport au schéma E/A :

- il n'existe pas de types d'associations ;
- tout type d'entité possède au moins un attribut ;
- tout attribut est monovalué ;
- tout attribut est élémentaire ;
- les contraintes d'intégrité reconnues sont les identifiants et les contraintes référentielles.

## *2.2.2 Passage du schéma conceptuel au schéma logique*

Maintenant que nous savons à quoi le niveau logique aboutit, nous allons voir de quelle façon on y aboutit. On divise le processus de conception logique en deux sous-processus : la conception logique indépendante du modèle cible, et la conception logique dépendante du modèle cible. Le schéma de la figure 2.2 permet d'avoir un aperçu de l'articulation de ces deux sous-processus.

La **conception logique indépendante** est commune à tous les modèles cibles et donne un schéma conceptuel **simplifié** et **partiellement optimisé**. La simplification du schéma conceptuel consiste à éliminer les structures de niveau d'abstraction trop élevé (types d'associations plus que binaires, relations IS-A, etc.). L'optimisation se fait sur base d'informations sur la

[8]Le schéma conceptuel ne doit pas prendre en compte les optimisations : en effet, il décrit la sémantique de la base de données, indépendamment des implémentations possibles.

charge de la base de données. La phase de conception logique aboutit à un schéma intermédiaire.

La **conception logique dépendante** a pour but de produire, à partir de ce schéma intermédiaire, un schéma conforme au modèle du SGBD visé. Nous avons défini ci-dessus les caractéristiques d'un schéma E/A conforme au modèle relationnel. L'optimisation de ce schéma se fera en fonction de divers critères de performance.

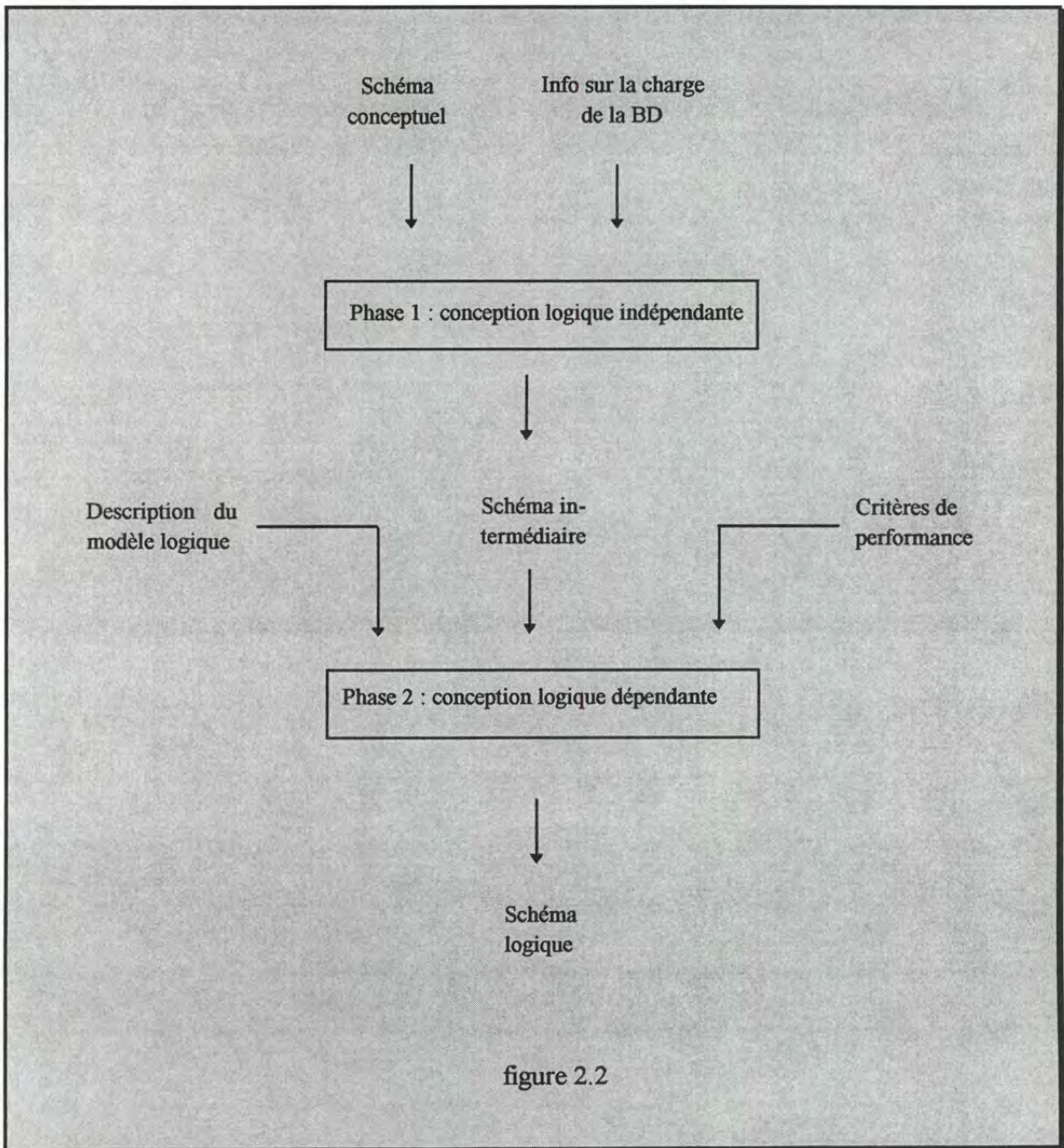


figure 2.2

Ces deux sous-processus de conception sont réalisés à l'aide de **transformations**. La conception logique revient en fait à appliquer des transformations sur le schéma de départ pour l'optimiser et pour le rendre conforme au modèle cible. Nous allons maintenant définir brièvement le concept de transformation.

### 2.2.3 Transformations

Le concept de transformation est extrêmement important pour une bonne compréhension de ce travail. En effet, une bonne partie du processus de reverse engineering se compose de l'application des transformations inverses de celles utilisées au cours du processus de conception. Une description des principales transformations utilisées au cours du processus de conception aurait sa place ici. Cependant, pour faciliter la lecture de ce travail, nous avons préféré les présenter au fur et à mesure des recherches des structures résultantes de leur application. Nous allons nous contenter ici de définir les concepts de transformation, de transformation réversible et de transformation inverse, avant de donner un exemple de transformation. Pour plus d'informations sur les transformations, on peut consulter [HAI91] et [HAI93b].

Faire une **transformation**, c'est remplacer un ensemble d'objets C d'un schéma S par un ensemble d'objets C', conduisant au schéma S'. Lorsque S et S' sont sémantiquements équivalents on a une **transformation réversible** (aussi appelée transformation à sémantique constante). Pour chaque transformation réversible, il existe la **transformation inverse**, permettant de passer de S' à S.

Une transformation peut générer de nouvelles contraintes d'intégrité, ou transformer des contraintes existantes. Ces contraintes devront être gérées par le SGBD ou par les programmes d'application.

**Exemple de transformation**

Dans la transformation de la figure 2.3, on voit que l'on remplace le type d'associations entre `COMMANDE` et `CLIENT` par l'attribut de référence `Num-cli` dans commande, et de la contrainte d'inclusion stipulant que toutes les valeurs de Num-`cli` dans `commande` doivent se retrouver dans `client`. Cette transformation est très souvent utilisée dans la phase de conception logique dépendante lorsque le SGBD visé est de type relationnel (en effet, en relationnel, il n'y a pas de types d'associations). Cette transformation est réversible : partant du schéma supérieur, il existe une transformation permettant de retrouver le schéma inférieur.

On remarque que la disparition du type d'associations `PASSE` a généré une contrainte d'inclusion. Les valeurs de `Num-cli` de `COMMANDE` doivent se retrouver dans les valeurs de `Num-cli` de `CLIENT`. Une telle contrainte est appelée contrainte référentielle. Cette contrainte peut être gérée directement par le SGBD (déclaration de `FOREIGN KEY` pour `COMMANDE.Num-cli`, utilisation de checks, de triggers) ou dans le programme d'application, par des actions ou des vérifications adéquates.

## 2.3 LE NIVEAU PHYSIQUE

Le niveau physique consiste à passer du schéma logique à un schéma physique qui est une description de l'implémentation de la base de données, c-à-d une description des paramètres physiques des structures de données et de leur usage. On obtient donc le texte DDL de description de la base de données.

On peut ajouter au niveau physique l'écriture des routines des programmes d'application qui gèrent certaines contraintes d'intégrité.

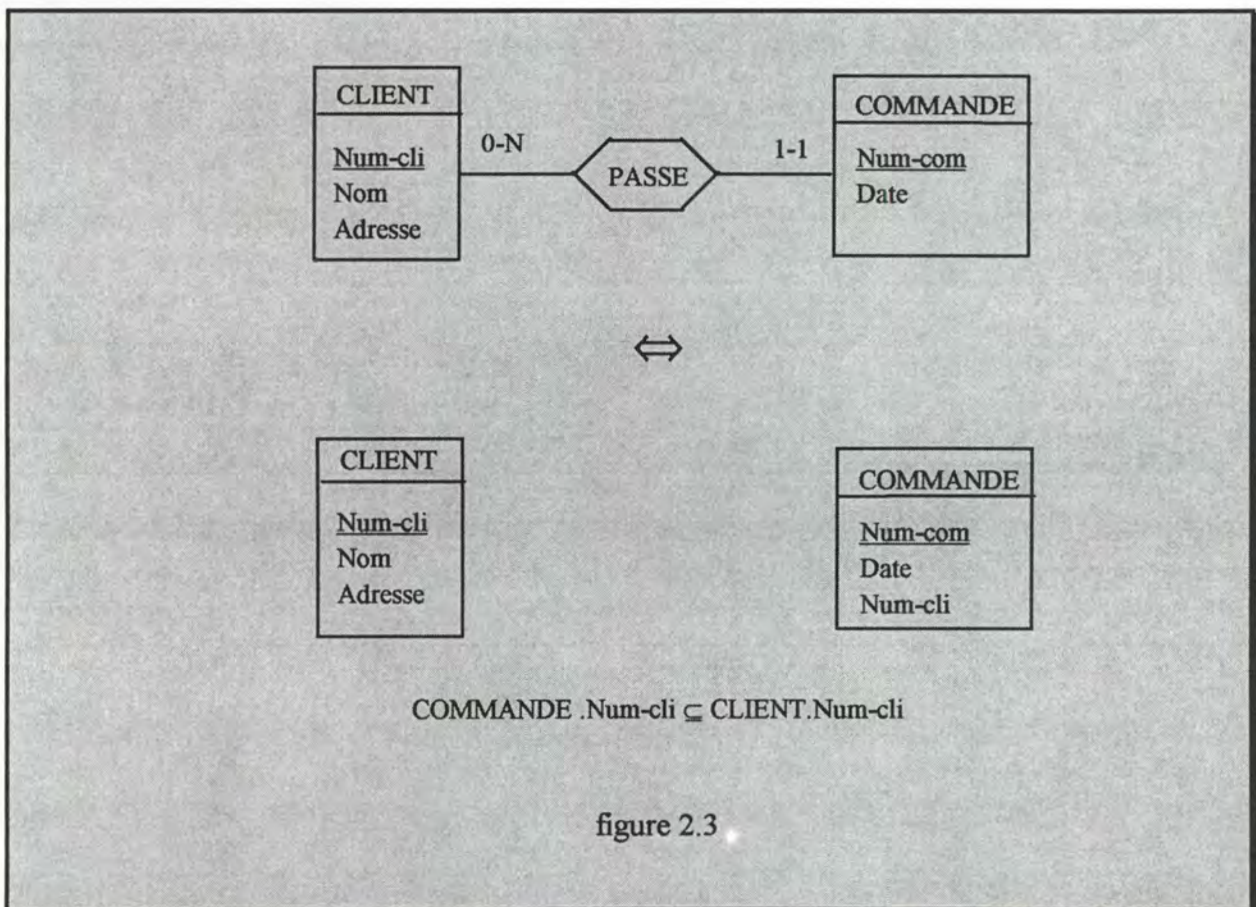



figure 2.3

Lorsqu'on a le schéma logique, on peut très facilement le transformer en un schéma de la base de données. L'**interprétation d'un schéma conforme au modèle relationnel** est la suivante :

| | | |
|---|---|---|
| type d'entités | → | table |
| attribut | → | colonne |
| identifiant | → | identifiant primaire (`primary key`) ou identifiant candidat (`unique`) |
| attributs de référence | → | clé étrangère (`foreign key`) |

# CHAPITRE 3
# METHODE DE REVERSE ENGINEERING DE BASES DE DONNEES

La méthode sur laquelle nous nous basons est schématisée à la figure 3.1. On peut en trouver une description précise dans [HAI93a]. Cette méthode est divisée en deux grandes étapes : l'extraction des structures de données et la conceptualisation des structures de données. Cette méthode est générique, en ce sens qu'elle convient pour le reverse engineering de tout Système de Gestion de Données (SGD). Etant donné que le SGD qui nous intéresse est un SGBD relationnel, la description que nous en ferons sera adaptée à ce type de SGBD.

Au départ, on a des textes sources. Il s'agit de la description de la base de données[9] (déclarations de tables, de vues, de checks et de triggers) et du code source des applications qui utilisent cette base de données[10].

La **phase d'extraction** fournit une description complète des structures de données conforme au modèle relationnel. La phase d'extraction des données est l'inverse du niveau physique de la méthode de conception d'une base de données que nous venons de décrire.

La **phase de conceptualisation** essaie de rendre explicite la sémantique du schéma physique. Cette phase consiste principalement à retirer les constructions techniques (par exemple, les structures d'optimisation) et à *de-traduire* les structures de données du SGD. A la fin de la phase de conceptualisation on a donc un schéma conceptuel de la base de données.

Notons que les sources d'informations sur lesquelles nous nous basons seront aussi utilisées dans la phase de conceptualisation. En effet, dans la phase d'extraction on retrouve uniquement des structures relationnelles. Au cours de la phase de conceptualisation, on retrouve des structures de plus haut niveau, et à ce moment là on peut retourner dans nos sources d'information pour de nouvelles recherches.

---

[9]Cette description peut se retrouver dans le texte DDL ou dans les tables système.

[10]On s'intéresse surtout dans le code source aux requêtes sur la base de données.

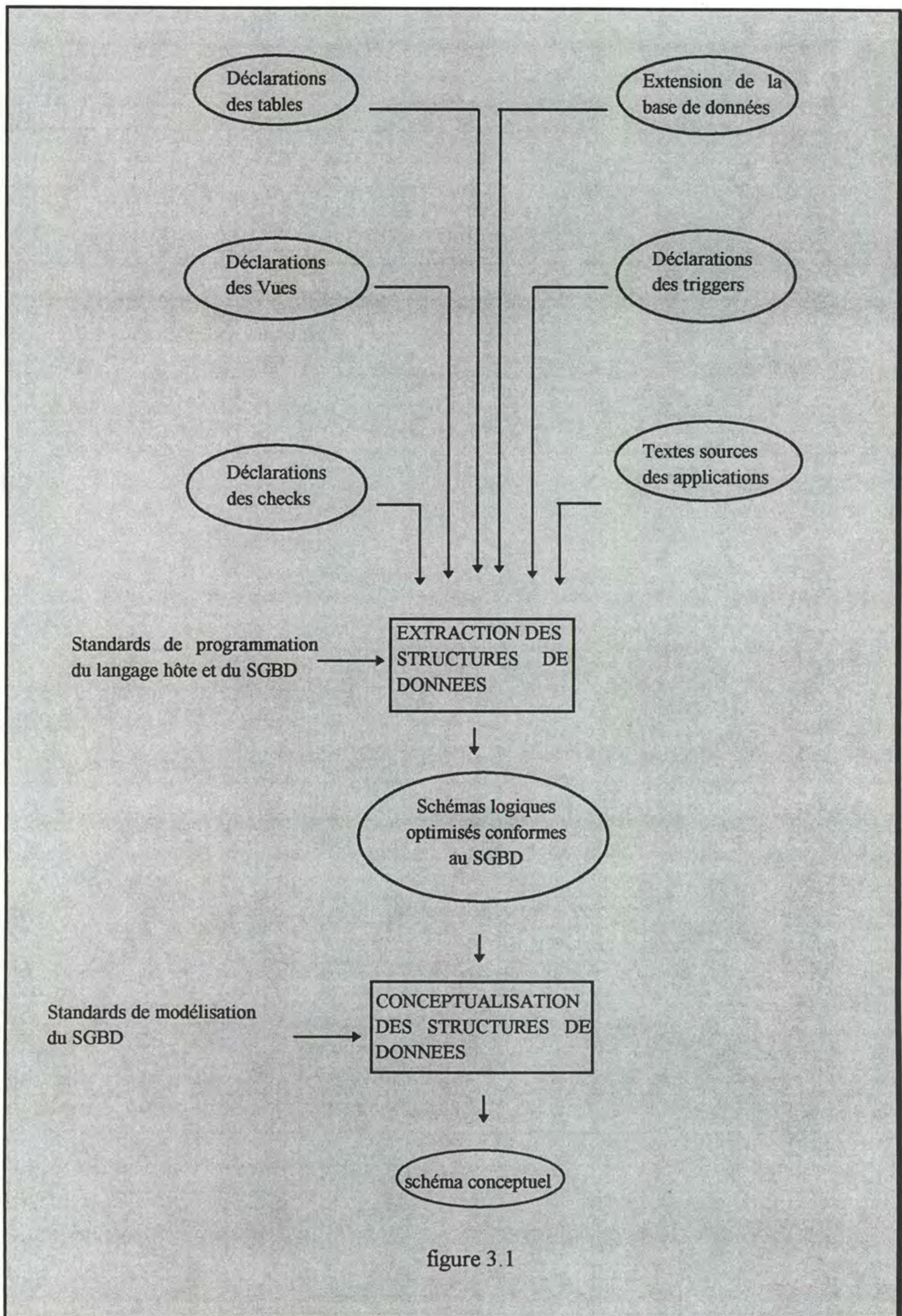



figure 3.1

## *Extraction des structures de données*

Dans cette phase, on analyse le texte source des schémas et programmes, afin de retirer les structures de données du schéma optimisé.

Le problème est double :

1. Retrouver les structures de données du SGBD.

   On trouve une description du schéma conforme dans le texte DDL ou dans le dictionnaire de données.

2. Retrouver les spécifications éliminées.

   Ce processus est basé principalement sur l'analyse des parties procédurales (parties de programmes ou triggers). Ces parties procédurales servent à la vérification de contraintes d'intégrité et au calcul de données dérivées.

Lorsque ces deux problèmes ont été résolus, on obtient des vues sur des structures de données. Il y alors deux stratégies possibles : soit intégrer les vues pour obtenir un schéma global que l'on conceptualise, soit conceptualiser chaque vue et intégrer ensuite.

## *Conceptualisation des structures de données*

Le but de cette phase est d'éliminer les constructions techniques du schéma conforme, de réduire les constructions dépendantes du SGBD, d'éliminer les redondances de données, de rendre les structures cachées explicites. On obtient ainsi un schéma conceptuel clair, normalisé et naturel. Cette phase se compose de 5 étapes. Elles seront présentées dans un ordre logique (du 'moins conceptuel' au 'plus conceptuel'). Cependant, certaines étapes peuvent se recouvrir partiellement. De plus, certaines étapes peuvent/doivent être réalisées en paralèlle et des retours en arrière sont possibles.

Les étapes sont les suivantes.

**Etape 1 : Renommage et nettoyage du schéma.**

On retire tout ce qui ne concerne pas les données elles-mêmes, et il se peut aussi que l'on renomme les structures de données (par exemple, à cause des mots réservés, des conventions de syntaxe, etc.).

**Etape 2 : Dé-traduction du schéma conforme**

La production d'un schéma conforme à un SGBD implique la traduction de structures de données qui ne sont pas conformes au SGBD. La détection de telles structures transformées et leur remplacement par leur origine conceptuelle donne un schéma de plus haut niveau.

Le principal problème est qu'il n'y a pas de correspondance biunivoque entre les structures conceptuelles et leur traduction conforme au SGBD.

**Etape 3 : Elimination de constructions d'optimisation[11].**

On recherche ici les redondances de dénormalisation, les redondances structurelles et les structures résultantes de l'application de transformations de restructuration (fusion horizontale ou verticale et partionnement horizontal ou vertical). Ces concepts seront expliqués lorsque nous proposerons des moyens de retrouver des constructions d'optimisation.

**Etape 4 : Expression du schéma dans un modèle de plus haut niveau**

On rend le schéma obtenu par les précédentes étapes plus lisible en le restructurant. Par exemple, on remplace des structures binaires par des structures n-aires, on exprime les structures de généralisation/spécialisation, etc.

**Etape 5 : Intégration de schémas**

On fusionne les vues obtenues lors de la conceptualisation. L'intégration de schémas est aussi utilisée quand on fait du reverse engineering de plus d'une base de données, ou d'une base de données hétérogène.

Cette étape ne sera pas traitée dans ce mémoire. On peut trouver des moyens d'intégrer des schémas dans [PHE93] et [SPA].

---

[11]Dans la méthode originale, on distinguait les optimisations indépendantes et dépendantes du SGBD, chacune faisant l'objet d'une étape. Etant donné que les transformations d'optimisation que nous considérons sont applicables pour les deux types d'optimisation, nous avons jugé opportun de ne considérer qu'une seule étape d'optimisation.

# PARTIE II
# RECHERCHES PROPOSEES

Nous venons de voir que la méthode sur laquelle nous nous basons se compose d'une phase d'extraction dans laquelle on tente d'obtenir un schéma conforme au modèle relationnel et d'une phase de conceptualisation dans laquelle on tente d'obtenir un schéma clair et naturel par des restructurations et des enrichissements progressifs du schéma relationnel.

Rappelons aussi que les sources d'informations à notre disposition sont les suivantes : les textes DDL (reprenant les déclarations des tables, des index, des vues, des checks, et des triggers), les textes sources des programmes d'application et l'extension de la BD.

A chaque étape nous allons expliquer comment retrouver des indices sur l'existence d'une spécification possible du schéma cible à partir des sources d'information. Ces indices peuvent porter sur l'existence probable de telle ou telle contrainte d'intégrité. Ils peuvent également porter sur les traces de l'utilisation d'une transformation particulière lors de la conception.

Il est important de noter qu'il n'est généralement pas possible de déterminer avec certitude la présence d'une contrainte. Il en va de même pour la détection de l'application d'une certaine transformation. Dès lors, lorsque des indices ont été trouvés, l'avis de la personne qui fait le RE (nous appelerons cette personne par la suite l'*utilisateur* de la méthode de RE) est important.

La structure de cette deuxième partie est la suivante. Nous commencons par une présentation des sources d'information.

Ensuite, nous passons à la phase d'extraction des structures de données. Dans cette phase, nous proposons des recherches des types d'entités et de leurs attributs, des identifiants, des attributs facultatifs, des dépendances fonctionnelles et des contraintes référentielles.

Enfin, nous abordons la phase de conceptualisation des structures de données. Nous suivrons les 4 premières étapes définies dans la méthode générale.

# CHAPITRE 4
# LES SOURCES D'INFORMATION

## 4.1 PRESENTATION GENERALE

### *4.1.1 Les déclarations de tables*

La source d'information la plus fiable et la plus facile à analyser est la déclaration des tables. Celle-ci peut être consultée dans le texte DDL, ou dans les tables système. La déclaration des tables reprend, outre la définition des attributs et de leurs types, la déclaration des index, des valeurs obligatoires (clause `NOT NULL`) des identifiants (clause `PRIMARY KEY`) et des clés étrangères (clause `FOREIGN KEY`). Une analyse de la déclaration des tables a déjà été réalisée par Jean-François Bellem et Xavier Deflorenne [BEL93]. Nous mentionnerons donc ce type de recherche, mais sans entrer dans les détails.

### *4.1.2 Les noms des tables et des attributs*

Les noms des tables et des attributs constituent une autre source d'information. Le concepteur ne choisit généralement pas les noms des tables et des attributs au hasard. Ces noms peuvent donc constituer une source d'information qui, si elle n'est pas très fiable, est assez facile à analyser. Des recherches sur les noms ont été réalisées dans [BEL93]. Nous ne les développerons donc pas en détail.

### *4.1.3 Les vues*

Les vues constituent une bonne source d'information. En effet, elles peuvent refléter la façon dont le concepteur percoit les données, et dès lors elles fournissent de bons indices pour retrouver le schéma conceptuel.

### *4.1.4 Les checks*

Pour rappel, un check vérifie que la base de données respecte à tout moment une certaine contrainte. Une même contrainte peut être implémentée par check de diverses façons. Nous ne verrons que quelques façons de définir des checks (celles qui nous semblent être les plus susceptibles d'être utilisées).

### 4.1.5 Les comportements imposés aux données : les triggers et les requêtes

Pour gérer les contraintes d'intégrité, le concepteur peut, lors d'un ajout, d'une suppression ou d'une modification dans une table, effectuer certaines opérations sur d'autres tables et/ou attributs. Ces opérations sont soit des vérifications, soit des actions (ajout, modification ou suppressions dans des tables). Nous appelerons comportements imposés aux données ce genre d'opérations.

On peut imposer des comportements aux données au moyen de triggers ou de configurations de requêtes dans le programme d'application. Ces triggers et requêtes effectuent les mêmes opérations logiques. Nous avons donc jugé opportun de les exprimer sous une forme plus abstraite, appellée Unité de Comportement (UC). Nous développons les UC de gestion des différentes contraintes et leurs implémentations dans le tome 2. Dans le tome 1, nous avons préféré étudier les façons d'imposer des comportements aux données de façon moins formelle que dans le tome 2.

Les triggers et les requêtes constituent la source d'information pilier de nos recherches.

### 4.1.6 Les consultations des données

La manière dont on consulte les données dans le programme d'application peut refléter certaines contraintes. Par exemple, les jointures des requêtes de consultation peuvent donner des indices sur la présence de contraintes référentielles. Les consultations constituent une source d'information intéressante.

### 4.1.7 L'extension de la base de données

La recherche dans l'extension consiste à analyser le contenu des tables.

Une telle recherche peut parfois être coûteuse en temps. Une façon de remédier à ce problème est d'utiliser un échantillon suffisament représentatif de l'extension. On pourrait aussi restreindre la recherche dans l'extension en l'utilisant seulement pour confirmer certaines hypothèses.

La recherche dans l'extension est assez fiable. Il y a cependant un risque que l'état de la base de données à un moment donné reflète des contraintes qui n'existent pas. C'est notamment le cas lorsque le nombre de lignes des tables est peu élevé.

# 4.2 MOYENS D'ANALYSE

Nous allons maintenant considérer les moyens par lesquels on peut analyser les sources d'information que nous venons de présenter.

- Pour les **déclarations des tables**, on peut se référer au travail de Jean-François Bellem et Xavier Deflorenne qui ont mis au point un algorithme d'analyse de ces déclarations.
- Les **noms** peuvent être retrouvés dans le texte DDL ou dans les tables systèmes. On peut utiliser un outil permettant de retrouver des noms à l'aide de *masques* (comme par exemple les symboles * et ? utilisés en DOS).
- Un outil d'analyse de textes peut aider l'utilisateur à rechercher des configurations particulières de **vues, checks, triggers et requêtes.** En effet, les textes sources dans lesquels on trouvera ces configurations sont généralement très longs et difficiles à lire :
  - La recherche dans les triggers n'est pas aisée, pour plusieurs raisons. D'abord, il y a beaucoup de façon d'écrire un trigger (selon les SGBD les triggers sont différents, et dans un SGBD particulier, il y a encore plusieurs possibilités). Ensuite, un seul trigger peut gérer plusieurs contraintes.

    Il en va de même pour les checks.
  - La recherche dans les requêtes n'est non plus une tâche facile. En effet, Il est possible que l'utilisateur mette entre des requêtes implémentant une contrainte certains traitements supplémentaires (par exemple, pour la gestion des erreurs ou de la valeur nulle). De plus les routines faisant appel à SQL seront plus que probablement modularisées. On peut donc retrouver des requêtes faisant partie d'une même configuration dans des procédures différentes[12].

Nous proposons dans le tome 2 (cf chapitre 5) un atelier d'analyse de texte. Cet atelier comprend des outils d'édition et d'exécution de *patterns* et de *scripts*.

Un *pattern* est une structure décrivant une partie de texte de façon "générique". Certains composants du pattern peuvent être fixés, d'autres pas. On pourrait par exemple vouloir chercher dans un texte tous les débuts de phrase de la forme "La ville de *nom-ville*", *nom-ville* représentant un nom quelconque de ville. Pour cela, on peut définir un pattern composé de la chaîne fixe de caractères "La ville de", suivie d'un symbole représentant une chaîne de caractères non fixée.

Un *script* est un algorithme décrivant les recherches à effectuer sur le texte source. Un script permet d'enchaîner les recherches de différents patterns. Cela augmente les possibilités de recherche.

---

[12] Pour remédier (partiellement) à ce problème, on peut utiliser un outil permettant de suivre pas à pas l'exécution d'un programme (TRACE).

Remarquons qu'un tel outil ne permet pas de tout retrouver, il ne s'agit que d'un support.

- Des requêtes peuvent être utilisées pour effectuer la recherche dans l'**extension**. Nous proposons par la suite quelques algorithmes de recherche.

## 4.3 TABLEAU COMPARATIF

Voici un tableau comparatif des principales caractéristiques des recherches dans les sources d'information que nous venons d'évoquer.

| | Facilité d'analyse | Niveau de certitude |
|---|---|---|
| Déclarations des tables | **** | **** |
| Noms | *** | ** |
| Vues | ** | * |
| Checks | ** | **** |
| Comportements : triggers | ** | *** |
| Comportements : requêtes | * | ** |
| Consultations | ** | ** |
| Extension | * (1) | *** |

Signification des symbloles

| | |
|---|---|
| * | : mauvais |
| ** | : moyen |
| *** | : bon |
| **** | : très bon |

(1) pour la recherche dans l'extension, la difficulté d'analyse vient du temps nécessaire à la recherche. De plus, nous avons considéré une recherche complète dans toute l'extension. Si on restreignait les recherches (utilisation d'un échantillon ou confirmation d'hypothèses), on pourrait mettre trois étoiles (mais le niveau de certitude serait alors plus faible) .

# CHAPITRE 5

# EXTRACTION DES STRUCTURES DE DONNEES

Le but de l'extraction des structures de données est d'obtenir un schéma de la base de données conforme au modèle relationnel. Ce schéma conforme sera exprimé sous la forme E/A. Par facilité de dénomination, nous utiliserons encore la terminologie relationnelle et SQL. En effet, le lien entre un schéma relationnel et un schéma conforme au relationnel est direct.

On recherche au cours de la phase d'extraction les types d'entités et leurs attributs, les identifiants, les attributs facultatifs, les dépendances fonctionnelles et les contraintes référentielles.

## 5.1 LES TYPES D'ENTITES ET LEURS ATTRIBUTS

Pour chaque table déclarée dans le SGBD, on crée un type d'entités qui a le même nom. Les colonnes de la table deviennent les attributs du type d'entités[13]. Cette étape est donc directe et sans ambiguïté.

La figure 5.1 montre la traduction d'une table `CLIENT` en type d'entités.

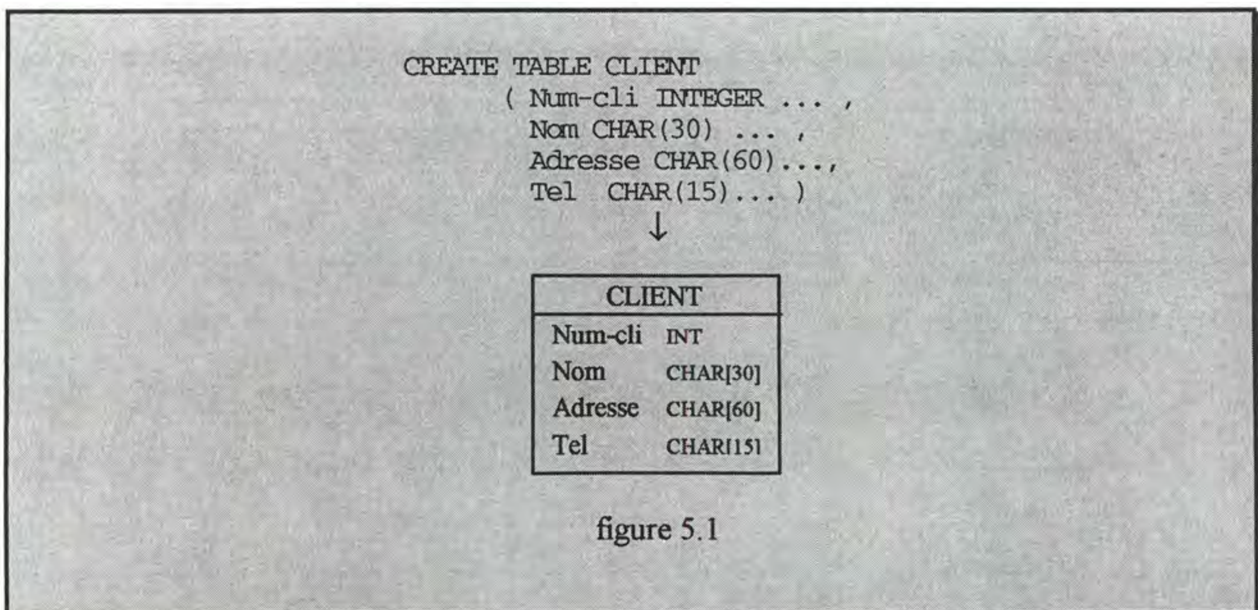



figure 5.1

[13]Nous utiliserons souvent le terme attribut pour désigner une colonne d'une table.

Remarque : dans la suite du travail, nous ne mentionnerons plus les types des attributs.

# 5.2 LES IDENTIFIANTS

Nous allons développer ici les moyens de recherche des identifiants d'une table. Nous considérons que chaque table est analysée séparément[14].

Le moyen le plus sûr de retrouver un identifiant est de se baser sur la déclaration des index uniques, et des colonnes clés primaires ou uniques. Néanmoins, le concepteur d'une BD peut ne pas avoir déclaré (tous) les identifiants. C'est pourquoi nous avons besoin d'autres indices pour pouvoir les retrouver. Passons-les en revue brièvement.

1. Un identifiant est souvent indexé.
2. Souvent l'attribut identifiant porte le même nom que la table, ou une abréviation de ce nom. Il se peut aussi qu'il soit préfixé ou postfixé par "id", "num", etc.
3. Lorsqu'un identifiant est composé de plusieurs attributs, il est possible que le concepteur ait (pour des raisons de facilité de dénomination) créé une vue dans laquelle les attributs identifiants sont perçus comme un attribut unique (à l'aide d'une concaténation).
4. Un trigger peut vérifier qu'une insertion ou une modification ne fait pas apparaître de doublon.
5. Lorsqu'un attribut identifiant est de type entier, un trigger peut affecter automatiquement une valeur à cet attribut lors d'une insertion. Un tel trigger ira chercher la plus grande valeur de la colonne identifiante et lui ajoutera 1.
6. Dans les programmes d'application, il se peut qu'avant une insertion ou une modification, on vérifie que la valeur de l'attribut identifiant inséré ou modifié ne se trouve pas déjà dans la table.
7. Dans les programmes d'application lorsqu'un identifiant est un attribut entier, il se peut parfois (de la même façon qu'au point 5) que sa valeur soit déterminée juste avant l'insertion en prenant la plus grande valeur de la colonne et en lui ajoutant 1.
8. Dans le programme d'application, lorsqu'une instruction `SELECT` est réalisée sans curseur c'est qu'on est certain d'obtenir une seule ligne. Donc si l'on trouve une requête du type `SELECT...FROM CLIENT WHERE Num-cli = :var-cli` et que cette requête n'est jamais accompagnée d'un curseur, c'est qu'on est certain à chaque fois de n'obtenir qu'une seule ligne. On peut donc faire l'hypothèse que l'attribut `Num-cli` de `CLIENT` est identifiant.
9. Dans l'extension de la BD les valeurs d'un identifiant sont toujours distinctes. Une démarche possible est donc d'examiner toutes les combinaisons d'attributs à l'aide de l'extension et de

---

[14]La recherche des contraintes référentielles (cf. 5.5), très liée à la recherche des identifiants, peut nous permettre de retrouver encore quelques indices sur la présence de ceux-ci. Cependant l'analyse de ces contraintes ne peut être faite en considérant chaque table séparément.

voir lesquelles sont vraissemblablement identifiantes. Cette méthode est très coûteuse en temps et peut mener à retrouver des identifiants qui n'en sont pas. C'est pourquoi l'extension ne devrait être utilisée que pour confirmer la présence d'un identifiant.

## 5.3 LES ATTRIBUTS FACULTATIFS

En relationnel, aucun attribut n'est facultatif. On peut toutefois simuler la présence d'un attribut facultatif en utilisant une valeur particulière[15] (nous l'appellerons valeur nulle). Cette valeur n'a d'autre signification que de représenter une valeur d'attribut absente.

SQL propose la valeur `NULL` pour représenter une valeur nulle[16]. On peut retrouver des traces de l'utilisation de la valeur `NULL` dans la déclaration des tables. Par défaut tout attribut peut prendre la valeur `NULL` excepté ceux qui ont été déclarés explictement `NOT NULL`. Ce n'est pourtant pas une bonne méthode pour distinguer les attributs obligatoires des attributs facultatifs car un concepteur peut ne pas déclarer les attributs obligatoires `NOT NULL`. Cependant la probabilité qu'il s'agisse bien d'un attribut facultatif augmente si la valeur `NULL` est utilisée dans des requêtes ou des triggers utilisant l'attribut en question, ou si elle est présente dans l'extension de la base de données.

L'utilisation de la valeur `NULL` peut être évitée. Le concepteur peut en effet utiliser une valeur conventionnelle[17]. L'utilisation d'une telle valeur est assez difficile à détecter. En effet, on ne sait pas ce que sera cette valeur, puisque tout est géré par le concepteur. Il peut s'agir de 0 pour le prix d'un produit, ou '*****' pour un libellé, etc. Dans le tome 2, nous donnons quelques moyens de retrouver les valeurs conventionnelles.

---

[15]On peut aussi transformer les attributs facultatifs en types d'entités, comme nous le verrons plus loin (cf 6.4.2).

[16]Notons qu'en SQL, la valeur NULL ne sert pas qu'à représenter une valeur absente. Elle sert aussi à représenter une valeur inconnue pour un attribut.

[17]L'utilisation d'une valeur conventionnelle est cependant assez rare.

# 5.4 LES DEPENDANCES FONCTIONNELLES

Une dépendance fonctionnelle (DF) est une contrainte qui spécifie qu'une valeur d'un attribut X (ou d'un ensemble d'attributs X) détermine de façon univoque la valeur d'un autre attribut Y (ou d'un ensemble d'attributs Y). X est appelé déterminant, Y déterminé. On peut déduire de cette définition qu'il y a une DF de l'identifiant d'une table vers tous les autres attributs de la table[18].

Dans le tome 2, nous présentons un algorithme de recherche des DF dans l'extension. Cette recherche est coûteuse en temps mais peut être intéressante étant donné le peu de moyens à notre disposition pour retrouver des DF.

Une DF peut être gérée dans les programmes d'application ou à l'aide de triggers :

1. Un trigger peut vérifier lors d'une modification ou d'une insertion que toutes les lignes qui ont le même déterminant que la nouvelle ligne ont le même déterminé que la nouvelle ligne. Cette vérification peut aussi être faite au sein du programme d'application juste avant l'insertion ou la modification.
2. Un trigger peut lui-même aller rechercher la valeur du déterminé lors d'une insertion ou d'une modification si la valeur du déterminant inséré ce trouve déjà dans la table. Les programmes d'application peuvent aussi aller rechercher la valeur du déterminé.
3. Dans le programme d'application on peut retrouver une requête du type :

```
EXEC-SQL SELECT DISTINCT a2 FROM A WHERE a1 = :var-a1 END EXEC
```

Si une telle requête n'est pas accompagnée de la définition d'un curseur, alors c'est que le concepteur est certain de ne retrouver qu'une seule valeur de `a2` en donnant une valeur de `a1`, et donc c'est que `a1` est déterminant et `a2` déterminé.

Dans le tome 2, nous proposons également une recherche dans les vues.

[18] Les dépendances fonctionnelles pourraient donc servir à une recherche des identifiants. On trouve dans [HAI89b] une recherche des identifiants basée sur les dépendances fonctionnelles.

# 5.5 LES CONTRAINTES REFERENTIELLES

Les contraintes référentielles sont des contraintes d'intégrité permettant d'implémenter un lien relationnel par clé étrangère. De telles contraintes seront transformées en types d'associations one-to-many.

Nous allons distinguer la recherche des contraintes d'inclusion de la recherche des contraintes d'égalité

## *5.5.1 Les contraintes d'inclusion*

Les développements qui suivent sont basés sur le schéma de la figure 5.2.

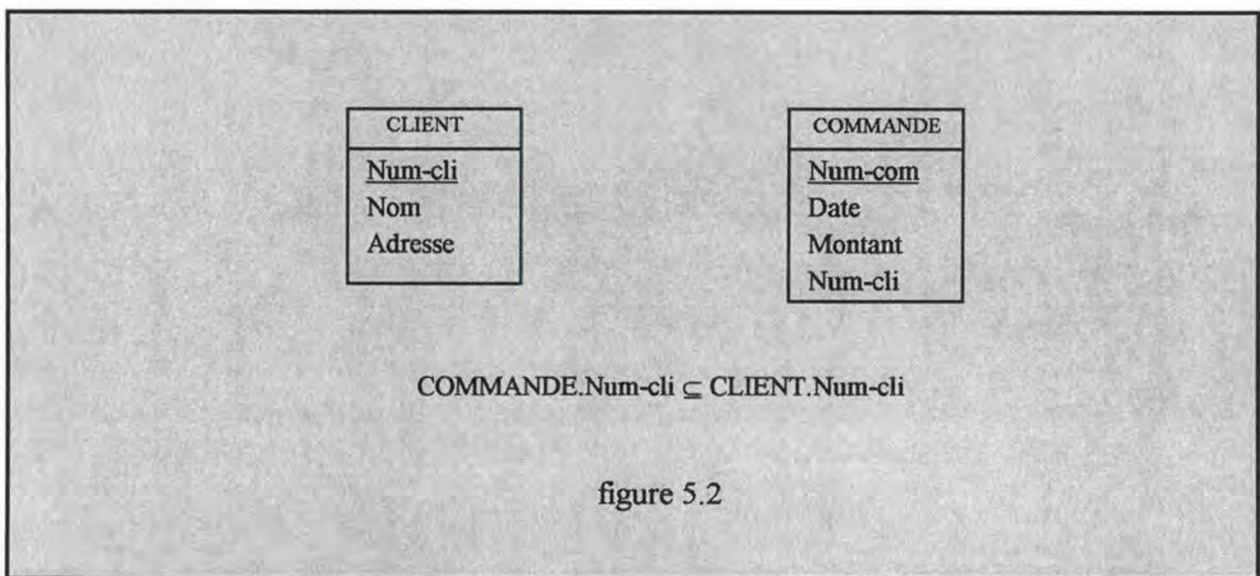


figure 5.2

La contrainte d'intégrité de ce schéma est appelée contrainte d'inclusion et signifie que toute valeur de l'attribut `COMMANDE.Num-cli` (excepté la valeur nulle) doit se retrouver dans une valeur de l'attribut `CLIENT.Num-cli`. Comme celui-ci est identifiant, `COMMANDE.Num-cli` est une clé étrangère. Une clé étrangère peut être facultative ou obligatoire.

La clé étrangère peut être déclarée explicitement par une clause de la définition de la table `COMMANDE`.

```
FOREIGN KEY (Ref-cli) REFERENCES CLIENT
```

La recherche de telles clauses est un moyen sûr pour retrouver les contraintes référentielles.

Néanmoins le concepteur n'est pas obligé d'utiliser cette déclaration. De plus, dans les anciens SGBD elle n'existe pas. C'est pourquoi nous avons besoin d'autres indices pour retrouver les contraintes référentielles. Examinons-les brièvement.

1. Les clés étrangères sont souvent indexées.
2. Souvent une clé étrangère porte le même nom que l'identifiant qu'elle référence, ou que la table à laquelle appartient cet identifiant. Elle peut aussi être préfixée ou postfixée par 'FK', 'REF', etc.
3. Lorsqu'une clé étrangère est composée de plusieurs attributs, il est possible que le concepteur ait (pour des raisons de facilité de dénomination) créé une vue dans laquelle les attributs composants de la clé étrangère sont perçus comme un attribut unique (à l'aide d'une concaténation).
4. On pourrait avoir un check attaché à la table COMMANDE qui vérifierait la contrainte reférentielle. Il serait de la forme :

   ```
   CHECK (Num-cli IN (SELECT Num-cli FROM CLIENT))
   ```

5. Lors d'un ajout d'une ligne dans COMMANDE (où la valeur de Num-cli est non nulle), il est vraisemblable que l'on vérifie que la valeur de la clé étrangère Num-cli se retrouve bien dans CLIENT.Num-cli. Cette vérification peut se faire automatiquement par trigger mais également dans le programme d'application. Il en va de même lors d'une modification d'une ligne de COMMANDE.
6. Lors d'une modification d'une valeur de CLIENT.Num-cli, comme les valeurs de COMMANDE.Num-cli sont inclues dans CLIENT.Num-cli, il faut effectuer l'une des opérations suivantes :
   - vérifier qu'aucun COMMANDE.Num-cli ne référençait l'ancienne valeur de CLIENT.Num-cli ;
   - remplacer les valeurs de COMMANDE.Num-cli référençant l'ancienne valeur de CLIENT.Num-cli par la nouvelle valeur de CLIENT.Num-cli ;
   - remplacer les valeurs de COMMANDE.Ref-cli référençant l'ancienne valeur de CLIENT.Num-cli par la valeur nulle;
   - supprimer les lignes de COMMANDE dont l'attribut Num-cli référençait l'ancienne valeur de CLIENT.Num-cli.

   Tout ceci peut être réalisé par triggers ou dans les programmes d'application.
7. Si l'on veut supprimer une ligne de CLIENT, il y a une valeur de CLIENT.Num-cli en moins. Dès lors, il faut :
   - soit vérifier qu'il n'y avait pas de valeur de COMMANDE.Num-cli qui référençait la ligne supprimée ;

- soit affecter une valeur nulle aux valeurs de COMMANDE.Num-cli référençant la ligne supprimée ;
- soit supprimer les lignes de COMMANDE qui référençaient la ligne de CLIENT supprimée.

Ceci peut être réalisé par trigger ou dans les programmes d'application.

8. On peut réaliser une recherche des contraintes d'inclusion en examinant l'extension. Cette méthode est, à l'instar de la recherche des DF dans l'extension, extrêmement coûteuse puisqu'il faut essayer toutes les combinaisons possibles d'attributs. On peut malgré tout l'améliorer en ne considérant par exemple pour le côté droit que des identifiants, ou en considérant que les attributs des deux côtés de la contrainte doivent être du même type.
9. Les jointures ont souvent comme arguments une clé étrangère et l'identifiant qu'elle référence. Dès lors, en analysant les requêtes qui font une jointure entre deux ou plusieurs tables, on peut encore trouver des indices sur la présence d'une contrainte d'inclusion (et aussi sur la présence d'un identifiant). On cherchera par exemple des requêtes du genre

```
SELECT Nom
FROM CLIENT , COMMANDE
WHERE (CLIENT.Num-cli = COMMANDE.Num-cli) AND
      (COMMANDE.Montant > 1000)
```

Le même type d'analyse peut être fait en repérant dans le programme d'application des jointures *implicites*. On a une jointure implicite lorsque le contenu d'une variable provenant d'une requête sur une table est réutilisé en tant qu'argument du WHERE dans une requête sur une autre table.

## 5.5.2 Les contraintes d'égalité

Définir une contrainte d'égalité revient en fait à définir deux contraintes d'inclusion réciproques. On peut donc retrouver des contraintes d'égalité à partir des contraintes d'inclusion. Il existe quelques moyens spécifiques de recherche dans les triggers et les requêtes, que nous développons dans le tome 2.

# CHAPITRE 6
# CONCEPTUALISATION DES STRUCTURES DE DONNEES

Nous allons reprendre les 4 premières étapes définies dans la méthode générale. Nous verrons d'abord l'étape de **nettoyage et renommage du schéma**. Ensuite, nous verrons la **détraduction du schéma conforme**. Dans cette étape, nous rechercherons les types d'associations one-to-many, les identifiants composés partiellement ou complètement de rôles et les types d'entités manquants . Nous passerons alors à la **dé-optimisation**. Nous verrons la redondance de dénormalisation, la redondance structurelle, et les transformations de restructuration). La dernière étape est **l'expression du schéma dans un modèle de plus haut niveau**. On y recherchera les types d'associations complexes, des attributs particuliers, les contraintes d'intégrité sur les rôles, et les relations IS-A.

## 6.1 NETTOYAGE ET RENOMMAGE DU SCHEMA

Dans certains cas, au début de la conceptualisation, il faut opérer un nettoyage et un renommage du schéma.

### *6.1.1 Nettoyage*

Le schéma d'une base de données peut contenir des constructions qui n'appartiennent pas "vraiment" à la BD. Il s'agit des structures de données transitoires, des variables d'état des programmes ou des parties n'ayant plus d'utilité.

On trouve dans [BEL93] la notion de "tables parasites". Il s'agit de tables servant à

- conserver des paramètres de champs d'acquisition ;
- conserver des paramètres de présentation des données à l'écran ;
- contenir des informations sur le format d'impression des rapports ;
- contenir des parties de code ou des requêtes interprétées dynamiquement ;
- faire des historiques.

Pour retrouver de telles tables il suffit dans la plupart des cas de s'intéresser a leurs noms et à ceux de leurs attributs (ces noms sont souvent évocateurs). Dans certains cas une connaissance appronfondie des structures des programmes et du domaine d'application est nécessaire.

Ce genre de tables doit bien entendu être écarté du processus de RE.

### 6.1.2 Renommage

Les noms doivent parfois être modifiés pour diverses raisons : conventions de syntaxe du SGBD, mots réservés du SGBD, noms dans plusieurs langues, incohérences dans les conventions de noms, etc.

# 6.2 DE-TRADUCTION DU SCHEMA CONFORME

Au cours de cette étape on recherchera les types d'associations one-to-many, les identifiants composés partiellement ou complètement de rôles et les types d'entités manquants.

## *6.2.1 Les types d'associations one-to-many.*

Nous allons développer ici à partir d'un exemple la démarche classique de transformation d'un type d'associations one-to-many et le moyen de la retrouver à partir d'un schéma relationnel. Dans le tome 2, nous évoquons quelques transformations non-classiques.

### 6.2.1.1 Transformation d'un type d'associations one-to-many

Considérons le schéma E/A de la figure 6.1.

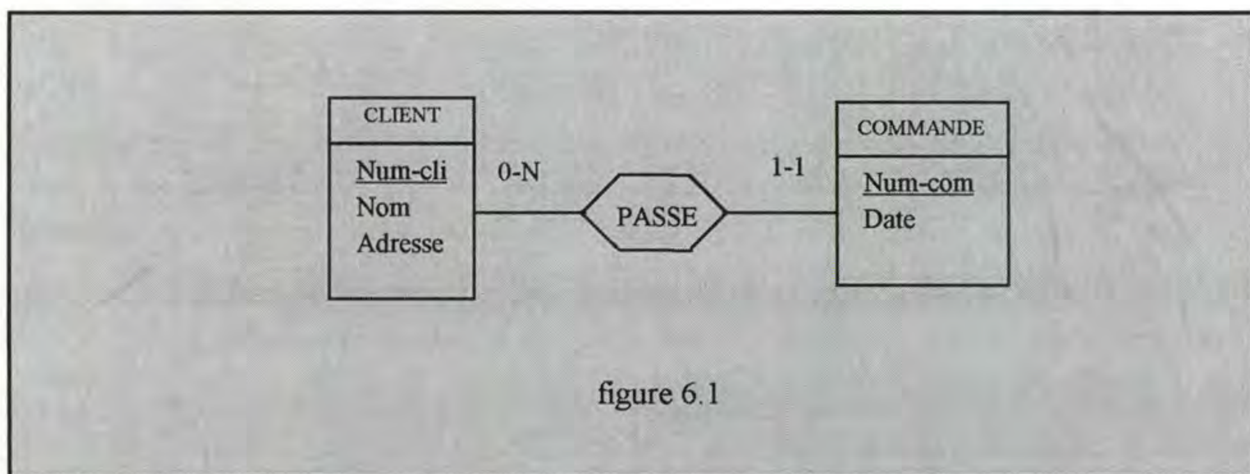



figure 6.1

On voit sur ce schéma qu'il y a une association one-to-many entre COMMANDE et CLIENT. Le modèle relationnel n'admet pas les types d'associations. Pour obtenir un schéma conforme au modèle relationnel, il faut donc supprimer le type d'associations PASSE. Pour ce faire, on ajoute au type d'entités COMMANDE un attribut, Num-cli, qui référence l'attribut Num-cli des CLIENTS reliés à COMMANDE. Cette transformation a pour résultat le schéma de la figure 6.2.

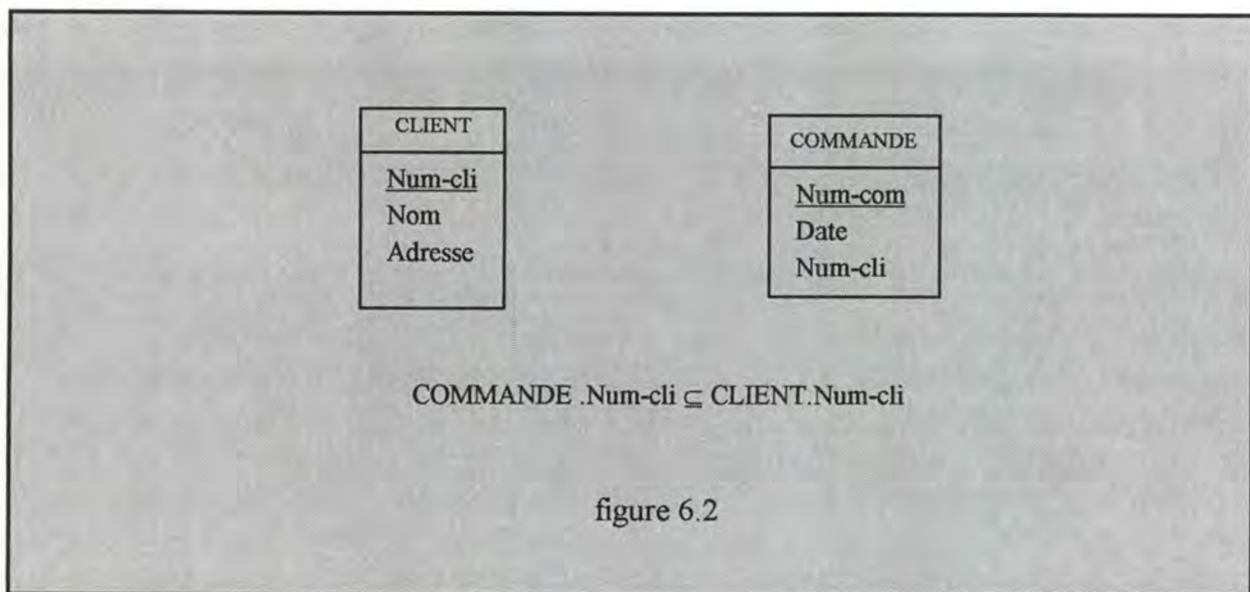



figure 6.2

On constate qu'il y a maintenant une contrainte d'inclusion entre COMMANDE.Num-cli et CLIENT.Num-cli. Cela signifie que toutes les valeurs de COMMANDE.Num-cli doivent se retrouver dans l'ensemble des valeurs de CLIENT.Num-cli. L'attribut COMMANDE.Num-cli est la clé étrangère, et la contrainte d'inclusion est une contrainte référentielle. De cette façon, pour une valeur donnée de COMMANDE.Num-cli, on peut accéder à une entité CLIENT ayant cette même valeur pour Num-cli. Inversément, pour une valeur donnée de CLIENT.Num-cli, on peut accéder aux entités COMMANDE ayant cette même valeur pour Num-cli. On a donc bien encore un *lien one-to-many*.

Remarques

- Si la connectivité minimale du rôle joué par CLIENT était 1, on aurait eu une contrainte d'égalité à la place de la contrainte d'inclusion (COMMANDE.Num-cli = CLIENT.Num-cli).
- Si la connectivité minimale du rôle joué par COMMANDE était 0, Num-cli de COMMANDE aurait été facultatif.
- Si la connectivté maximale du rôle joué par CLIENT avait été 1, Num-cli de COMMANDE aurait été identifiant.

### 6.2.1.2 Recherche d'un type d'associations one-to-many

L'objectif de cette recherche est de retrouver le schéma de la figure 6.1 à partir du schéma de la figure 6.2. On appliquera donc la transformation inverse de celle developpée ci-dessus.

La contrainte d'inclusion nous indique que Num-cli de COMMANDE est une clé étrangère référencant Num-cli de CLIENT. On peut donc enlever la clé étrangère et la remplacer par un type d'associations one-to-many entre COMMANDE et CLIENT.

On aura les connectivités suivantes :

- la connectivité maximale du côté de COMMANDE est 1 ;
- la connectivité maximale du côté de CLIENT est N, car Num-cli de COMMANDE ;
- la connectivité minimale du côté de COMMANDE est 1 car Num-cli est obligatoire dans COMMANDE (autrement dit, chaque occurence de COMMANDE est reliée à une occurence de CLIENT, puisque chaque occurence de COMMANDE a une valeur non nulle pour Num-cli). Cette connectivité aurait été 0 si Num-cli de COMMANDE avait été facultatif ;
- la connectivité minimale du côté de CLIENT est 0, car on a une contrainte référentielle qui est une contrainte d'inclusion (autrement dit, toutes les valeurs de Num-cli de CLIENT ne sont pas forçément reprises dans l'ensemble des valeurs de Num-cli de COMMANDE, donc il peut y avoir des clients n'ayant pas passé de commande). Cette connectivité aurait été 1 si on avait eu une contrainte d'égalité.

## 6.2.2 Les identifiants dont l'un des composants est un rôle

Considérons le schéma conceptuel de la figure 6.3.

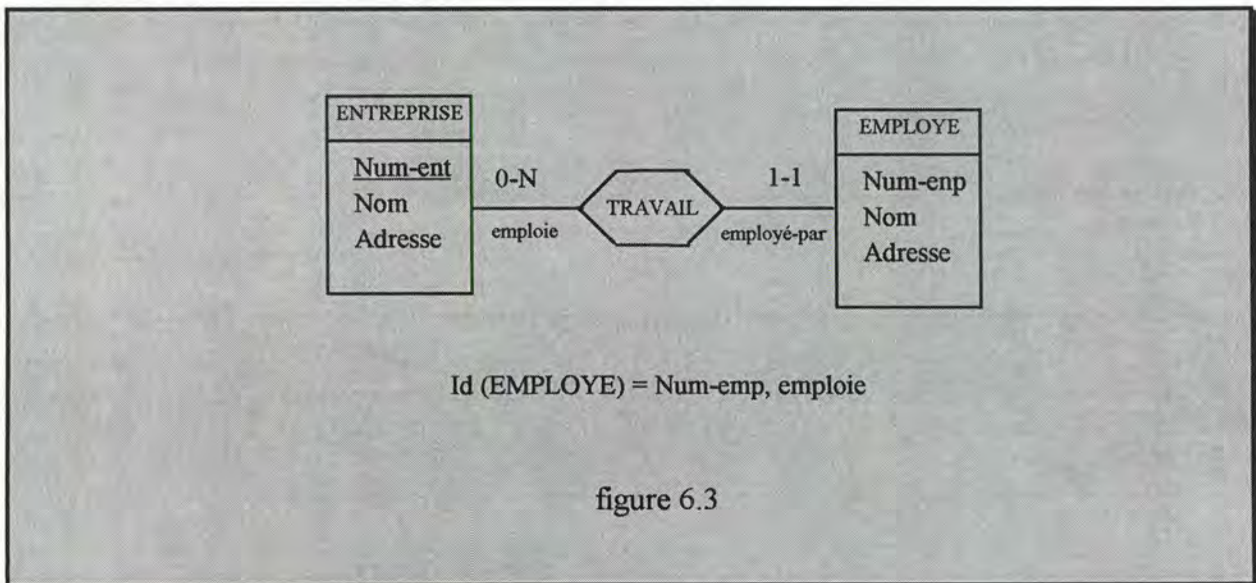



figure 6.3

On voit que EMPLOYE a un identifiant hybride composé de l'attribut Num-emp et du rôle emploie. La traduction de ce schéma en structures conformes au relationnel est donnée à la figure 6.4.

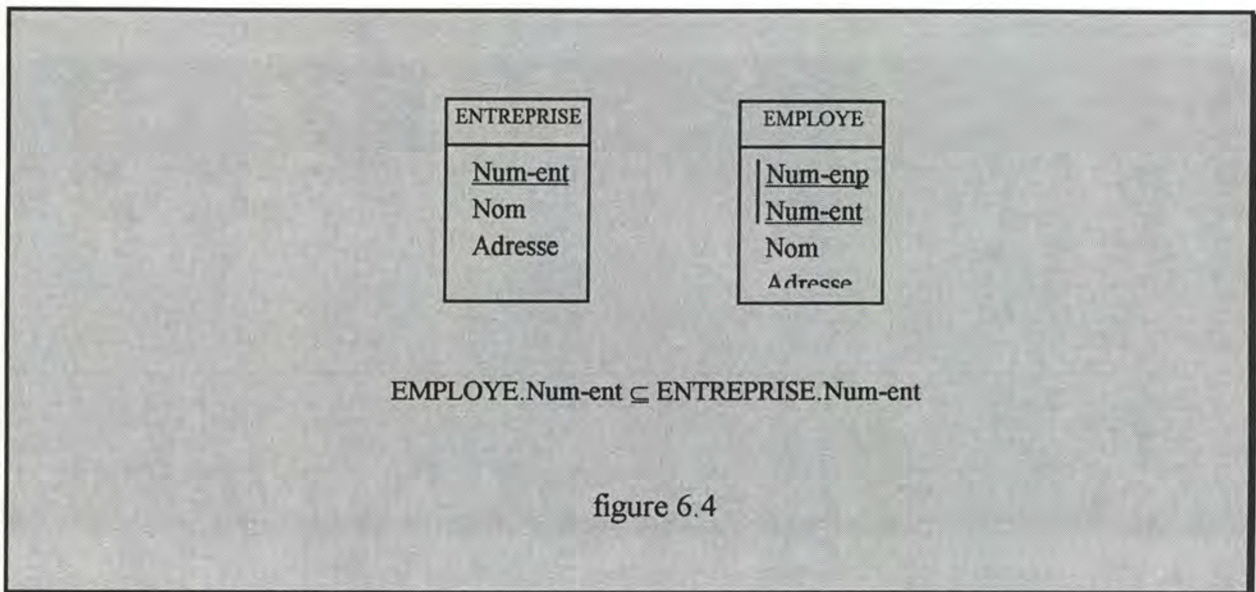



figure 6.4

Lors de la recherche des types d'associations one-to-many, on remplacera l'attribut clé étrangère EMPLOYE.Num-ent par un type d'associations one-to-many entre ENTREPRISE et EMPLOYE. Num-ent ne sera donc pas présent dans EMPLOYE. Il faudra alors que le rôle joué par ENTREPRISE soit composant de l'identifiant de EMPLOYE.

Donc, chaque fois qu'un des composants de l'identifiant d'une table est une clé étrangère, on retrouvera ce composant en tant que rôle composant de l'identifiant du type d'entités résultant de la dé-traduction de la table.

### 6.2.3 Les types d'entités manquants

Un *type d'entités manquant* [CAS93] est un type d'entités qui a été modélisé comme un attribut ou un groupe d'attributs dans une table du schéma relationnel. Pour le retrouver, on se base sur les contraintes d'inclusion. Nous développerons les types d'entités manquants dans le tome 2.

# 6.3 DE-OPTIMISATION D'UN SCHEMA RELATIONNEL

Lors de la conception d'une base de données, on peut être amené à optimiser le schéma relationnel, pour deux raisons : diminuer la taille des structures de données et diminuer le temps d'accès à ces données. Le schéma conceptuel ne doit pas prendre en compte les optimisations : en effet, il décrit la sémantique de la base de données, indépendamment des implémentations possibles.

Il y a 3 techniques principales d'optimisation d'un schéma : l'introduction **de redondances de dénormalisation**, l'introduction de **redondances structurelles**, et les **transformations de restructuration**. Nous allons développer ces techniques en montrant d'abord comment on les utilise, et nous proposerons ensuite des moyens pour les retrouver.

## *6.3.1 La redondance de dénormalisation*

Le concepteur d'une base de données peut décider de regrouper les attributs de deux types d'entités de façons à en obtenir un seul. La dénormalisation permet d'accéder aux deux types d'entités d'origine en un seul accès logique. Cependant, cela introduit de la redondance. On aura un schéma dénormalisé.

La dénormalisation peut aussi être due à d'autres raisons que l'optimisation (par exemple une mauvaise interprétation du réel percu).

Le schéma de la figure 6.5 donne un exemple de l'utilisation d'une transformation permettant d'aboutir à un schéma dénormalisé. On voit que le type d'entités `EMPL` contient une dépendance fonctionelle. Cette DF permet de détecter la présence d'une structure dénormalisée.

EMPLOYE
Num-emp
Nom-emp

1-1 DEPEND 1-N

DEPARTEMENT
Nom-dep
Localisation

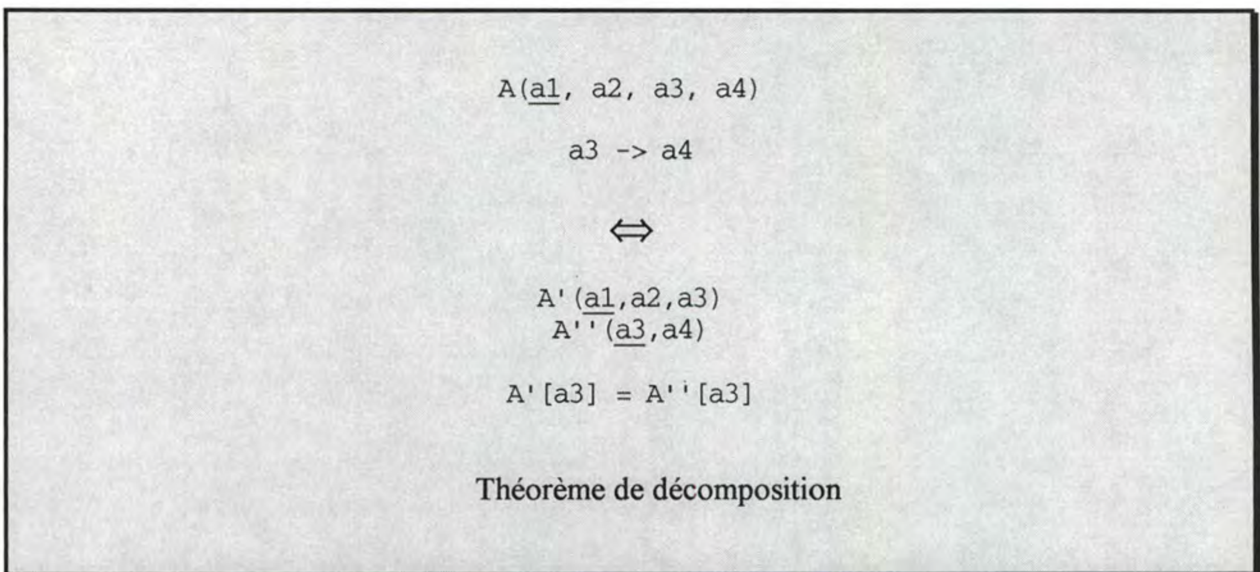
⇔

EMPL
Num-emp
Nom-emp
Nom-depart
Localisation

Nom-depart → Localisation

figure 6.5

La transformation permettant de dénormaliser un schéma et son inverse permettant de normaliser un schéma se basent sur le théorème de décomposition :




Théorème de décomposition

Nous allons donner maintenant un algorithme classique de normalisation [HAI89b] permettant de mettre un schéma sous 3ème Forme Normale (3NF).

**Définition : troisième forme normale (3NF).**

Soient R une relation, X un sous-ensemble des attributs de R, A un attribut de R.

R est sous 3NF

$\Leftrightarrow$

$X \rightarrow A$ et $A \notin X \Rightarrow$ soit X est un identifiant (non nécessairement strict) de R
soit A est un composant d'un identifiant de R

**Algorithme classique de normalisation en 3NF.**

1. Recherche des identifiants et des DF.
2. Recherche d'une couverture minimale des DF.
3. Recherche des DF non triviales[19] dont le déterminant n'est pas un identifiant.
4. Décomposition successive de la relation en se basant sur les DF du point 3 dont le déterminé n'entre dans la composition du déterminant d'aucune DF.

A chaque décomposition, le déterminant va devenir clé étrangère dans l'une des relations, et identifiant référencé par cette clé dans l'autre. On aura en outre une contrainte d'égalité entre cette clé étrangère et cet identifiant.

Remarque

Dans le schéma supérieur de la figure 6.5, les connectivités minimales des rôles étaient toutes 1. L'algorithme présenté dans cette partie ne fonctionne que si on prend cela comme hypothèse. Dans le tome 2 nous montrons comment dénormaliser un schéma dont les rôles ont des connectivités 0 (et comment revenir au schéma de départ).

[19]Une DF est triviale si lorsqu'on retire un attribut du déterminant, il n'y a plus de DF.

## *6.3.2 La redondance structurelle*

L'introduction d'une redondance structurelle consiste à ajouter à un schéma des structures qui ont la propriété que leurs instances peuvent être dérivées à partir d'instances d'autres structures. Nous allons envisager deux types de redondance : les attributs dérivables et la redondance entre types d'associations.

### 6.3.2.1 Les attributs dérivables

**Définition**

Un attribut dérivable est un attribut présent dans une entité ou dans une association et qui peut être calculé à partir d'un ou plusieurs autre attributs et/ou rôles.

Le schéma de la figure 6.6 donne un exemple d'attribut dérivable.

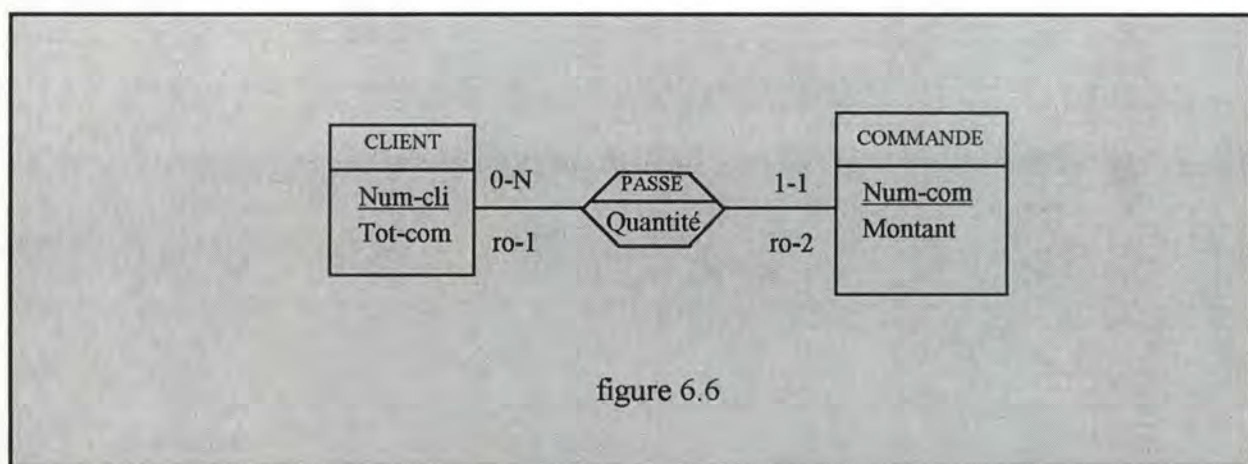



figure 6.6

Dans ce schéma, le montant total des commandes d'un client, `Tot-com`, est la somme des montants des commandes passées par ce client.

On peut trouver une telle redondance pour deux raisons :

- Le concepteur peut estimer que le montant total est un concept important du réel perçu qui doit être repris dans son schéma conceptuel. Cependant, ce genre d'attribut introduit une redondance structurelle, et ne doit pas être présent dans un schéma canonisé.
- L'attribut peut avoir été ajouté pour augmenter les performances de la BD. Par exemple si le montant total est souvent utilisé dans le programme d'application, il est préférable d'avoir un attribut dérivable `Tot-com`, comme ci-dessus, dans le schéma physique.

## Recherche

Le schéma relationnel de la figure 6.7 donne la traduction du schéma de la figure 6.6.

```
CLIENT (Num-cli , Tot-com)
COMMANDE (Num-com , Montant , Num-cli)

COMMANDE.Num-cli ⊆ CLIENT.Num-cli

∀ cli ∈ CLIENT, cli.Tot-com = Σ com.Montant |
com ∈ COMMANDE et com.Num-cli = cli.Num-cli
```

figure 6.7

La deuxième contrainte est la contrainte de redondance. Elle signifie que pour toute ligne de CLIENT, la valeur de l'attribut Tot-com doit être égale à la somme des valeurs des attributs Montant des lignes de COMMANDE liées.

Il existe diverses manières de rechercher cette contrainte :

1. On peut chercher des triggers ou des requêtes des programmes d'application qui, lors d'un ajout ou d'une modification d'une ligne de CLIENT, vérifient que l'attribut Tot-com de la ligne insérée ou modifiée respecte la contrainte.
2. On peut aussi chercher des triggers ou requêtes qui, lors de l'ajout, la suppression ou la modification d'une ligne de COMMANDE, vérifient que l'attribut Tot-com de la ligne de CLIENT liée respecte la contrainte.
3. On peut également chercher des triggers ou requêtes qui calculent la valeur de Tot-com lors d'un ajout d'une ligne dans CLIENT.
4. On cherchera aussi des triggers ou requêtes qui, lors d'un ajout, d'une suppression ou d'une modification d'une ligne de COMMANDE, modifient l'attribut Tot-com de la ligne de CLIENT liée, de telle façon que la contrainte de redondance soit respectée.
5. Une recherche sur les noms est aussi possible. Dans l'exemple, on a 'Tot' suivi de 'Com', qui est une abréviation de 'COMMANDE'.

Remarque

Il existe plusieurs autres formes d'attributs dérivables. Nous en développerons certaines dans le tome 2.

## 6.3.2.2 La redondance entre types d'associations

Nous allons considérer ici la redondance entre types d'associations one-to-many. Dans le tome 2 (cf annexe 4), nous envisageons aussi la redondance entre types d'associations many-to-many.

### Définition

Un type d'associations one-to-many peut parfois être redondant avec la composition d'autres types d'associations one-to-many. Le schéma de la figure 6.8 illustre cette situation.

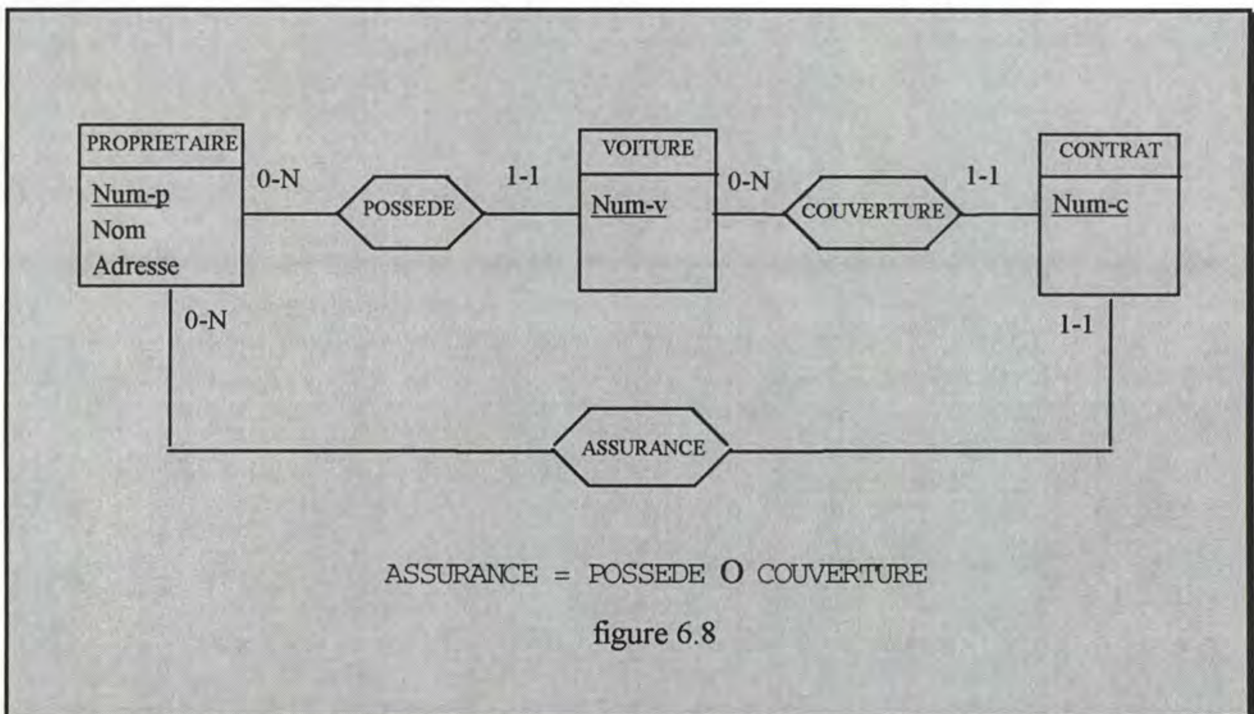



figure 6.8

Le type d'associations ASSURANCE est redondant avec la composition de POSSEDE et COUVERTURE. En relationnel, cela se traduira par une clé étrangère redondante dans CONTRAT (il s'agira de la clé étrangère référencant PROPRIETAIRE). Ce genre de redondance permet d'établir un lien direct entre deux tables éloignées reliées via d'autres tables.

### Recherche

Considérons la figure 6.9, qui donne la traduction du schéma de la figure 6.8 en relationnel.

```
PROPRIETAIRE (Num-p , Nom , Adresse)
VOITURE (Num-v , Num-p)
CONTRAT (Num-c , Num-v , Num-p)

VOITURE.Num-p ⊆ PROPRIETAIRE.Num-p
CONTRAT.Num-v ⊆ VOITURE.Num-v
CONTRAT.Num-p ⊆ PROPRIETAIRE.Num-p

∀ contrat ∈ CONTRAT :
∃ voiture ∈ VOITURE | voiture.Num-v = contrat.Num-v et
voiture.Num-p = contrat.Num-p
```

figure 6.9

La dernière contrainte est la contrainte de redondance. Elle signifie que pour toute ligne de CONTRAT, la clé étrangère référencant PROPRIETAIRE implémente le même lien que le lien CONTRAT-VOITURE-PROPRIETAIRE.

Il existe plusieurs moyens de rechercher cette contrainte :

1. On pourrait chercher des triggers ou des requêtes qui, lors d'une modification de la clé étrangère de CONTRAT référençant VOITURE, modifient aussi la clé étrangère référencant PROPRIETAIRE. Des triggers ou requêtes pourraient aussi vérifier que la modification a déjà été faite.
2. On peut aussi chercher des triggers ou requêtes qui, lors d'une suppression ou d'une modifcation d'une ligne de VOITURE, suppriment ou modifient la clé étrangère de CONTRAT implémentant le lien supprimé ou modifié avec PROPRIETAIRE.

### 6.3.3 Transformations de restructuration

Les transformations de restructuration servant à améliorer les performances d'un schéma n'introduisent pas de redondance. Néanmoins, elles doivent être prises en compte dans la phase de dé-optimisation. Il y a 4 techniques de restructuration : le partitionnement vertical et horizontal et leurs inverses.

Nous allons les définir et essayer de retrouver des indices indiquant qu'une de ces transformations a été appliquée sur le schéma.

### 6.3.3.1 Le partitionnement horizontal

#### Définition

Considérons le schéma de la figure 6.10.

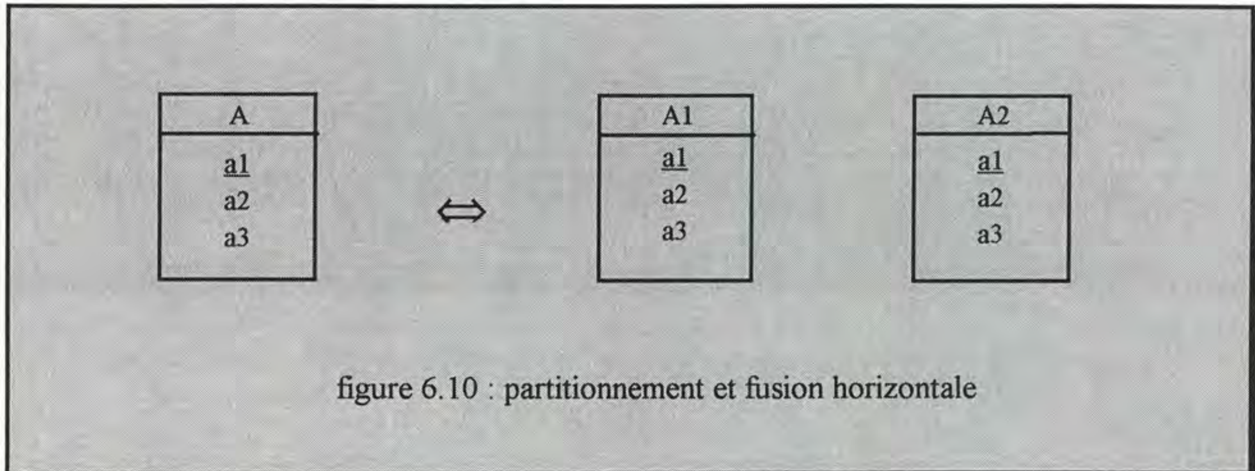


figure 6.10 : partitionnement et fusion horizontale

Le partitionnement horizontal consiste à remplacer le type d'entités `A` par les deux types d'entités `A1` et `A2`, de même structure, et de telle façon que la population de `A` soit partitionnée en les populations de `A1` et `A2`.

Cela permet d'avoir des ensembles de données plus petits, des index plus petits, et cela permet une meilleure allocation et gestion de l'espace.

#### Recherche

On peut tout d'abord dire que si l'on retrouve deux tables ayant exactement la même structure (nombre d'attributs, type de chaque attribut, identifiant, etc.), associées aux mêmes types d'entités et sur lesquelles portent les mêmes contraintes d'intégrité, et n'étant pas liées entre elles, il y a une chance pour qu'il s'agisse d'une table partitionnée horizontalement. On peut affiner cet indice en s'intéressant aux noms des attributs de chaque table : s'ils sont identiques, ou si seul leur préfixe ou suffixe change, cela permet d'améliorer l'heuristique.

On pourrait aussi faire une recherche dans les vues. On pourrait par exemple trouver la vue

```
CREATE VIEW A (a1, a2, a3)
AS (SELECT a1, a2, a3 FROM A1)
       UNION
   (SELECT a1, a2, a3 FROM A2)
```

### 6.3.3.2 La fusion horizontale

**Définition**

C'est la transformation inverse du partitionnement horizontal (figure 6.10). Cela permet de réduire le nombre de types d'entités, et cela facilite le clustering physique.

**Recherche**

Un bon moyen de retrouver une trace de fusion horizontale est de regarder les définitions de vues. Si on trouve deux ou plusieurs vues qui reprennent tous les attributs de la table, on peut faire l'hypothèse qu'il s'agit d'une table résultant de la fusion de deux ou plusieurs tables.

Si a2 était un attribut de type numérique, on pourrait avoir des vues du type :

```
CREATE VIEW A1
AS SELECT a1, a2, a3, a4
      FROM A
      WHERE a2 ≤ 100

CREATE VIEW A2
AS SELECT a1, a2, a3, a4
      FROM A
      WHERE a2 > 100
```

Si de telles vues n'étaient pas définies, on aurait peut-être des requêtes pour lesquelles on trouverait dans les WHERE des conditions semblables à celles présentes dans les définitions des vues.

### 6.3.3.3 Le partitionnement vertical

**Définition**

Considérons le schéma de la figure 6.11.

Le partitionnement vertical consiste à scinder le type d'entité A en deux (ou plusieurs) types d'entités liés par des types d'associations one-to-one. Les attributs de A qui sont souvent utilisés en même temps sont regroupés en entités. Le partitionnement vertical permet de réduire la taille physique des entités A, et d'améliorer le temps d'accès.

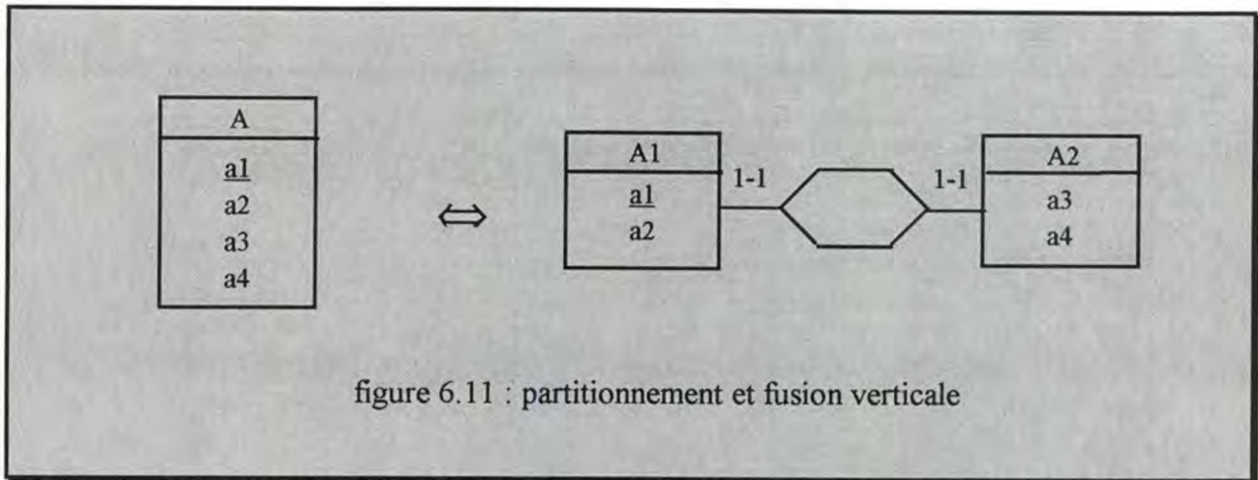


figure 6.11 : partitionnement et fusion verticale

### Recherche

On s'intéressera ici aux tables liées par des liens one-to-one.

Les indices pourront être retrouvés dans les déclarations de vues. Si on trouve une vue reprenant tous les attributs de deux ou plusieurs tables, on pourrait supposer qu'il s'agissait d'une table qui a été partitionnée verticalement :

```
CREATE VIEW A
AS SELECT a1, a2, a3, a4
      FROM A1, A2
      WHERE A1.a1 = A2.ref-a1
```

On peut aussi se baser sur les noms des tables et des attributs : si on trouve des similitudes entre les noms des tables, ou que les attributs de plusieurs tables sont tous préfixés de la même manière, cela donne un indice.

## 6.3.3.4 La fusion verticale

### Définition

C'est l'inverse du partitionnement vertical (figure 6.11). On fusionne deux types d'entité reliés par un lien one-to-one. Cela permet de d'éviter un double accès à des entités très liées.

### Recherche

Dans ce cas aussi, la recherche dans les déclarations de vues sera intéressante : si on trouve des vues définies sur seule table et reprenant des attributs distincts de cette table, on peut soupçonner que la table sur laquelle sont définies les vues est le résultat d'une fusion verticale.

On aurait par exemple :

```
CREATE VIEW A1 (a1 , a2)
AS SELECT a1, a2
      FROM A
```

```
CREATE VIEW A2 (a3 , a4)
AS SELECT a3, a4
      FROM A
```

Si on n'a pas ces vues, on peut aussi regarder dans les requêtes sur `A` : elles concerneront probablement rarement les 4 attributs, mais plutôt `a1 , a2` ou `a3 , a4`.

# 6.4 EXPRESSION DU SCHEMA DANS UN MODELE DE PLUS HAUT NIVEAU

Nous allons rechercher ici les types d'associations complexes, des attributs particuliers (attributs facultatifs transformés en types d'entités, attributs multivalués et attributs décomposables), les contraintes d'intégrité sur les rôles et les types d'associations et les relations IS-A.

## *6.4.1 Les types d'associations complexes*

Nous allons considérer trois sortes de T.A. complexes : les T.A. de degré supérieur à 2, les T.A. dotés d'attributs, et les T.A. de degré 2 mais qui sont de connectivités maximales N (c-à-d les T.A. many-to-many). La transformation de ces T.A. en structures conformes au modèle relationnel peut être effectuée via une transformation de ceux-ci en types d'entités. C'est la manière classique, mais il y a d'autres façons de les transformer (transformations non-classiques).

Nous allons considérer pour les trois cas la transformation classique et (le cas échéant) non-classique.

### 6.4.1.1 Les types d'associations many-to-many

La **transformation classique** d'un type d'associations many-to-many est schématisée à la figure 6.12.

On voit que le type d'associations COMMANDE a été transformé en type d'entités. Il n'y a donc plus dans le schéma inférieur que des types d'associations one-to-many.

Si le type d'associations COMMANDE contenait des attributs, ils feraient maintenant partie du type d'entités COMMANDE.

Nous allons voir maintenant de quelle façon, partant du schéma inférieur, on peut retrouver le schéma supérieur sur la figure 6.12.

On peut détecter le type d'associations many-to-many en s'intéressant aux connectivités. En effet, on voit que les entités du type COMMANDE ne jouent que des rôles de connectivités 1-1 dans leurs associations avec CLIENT et PRODUIT. On peut alors supposer qu'il s'agit d'un type d'entités provenant de la transformation d'un type d'associations many-to-many. On peut le remplacer par un type d'associations en appliquant la transformation inverse.

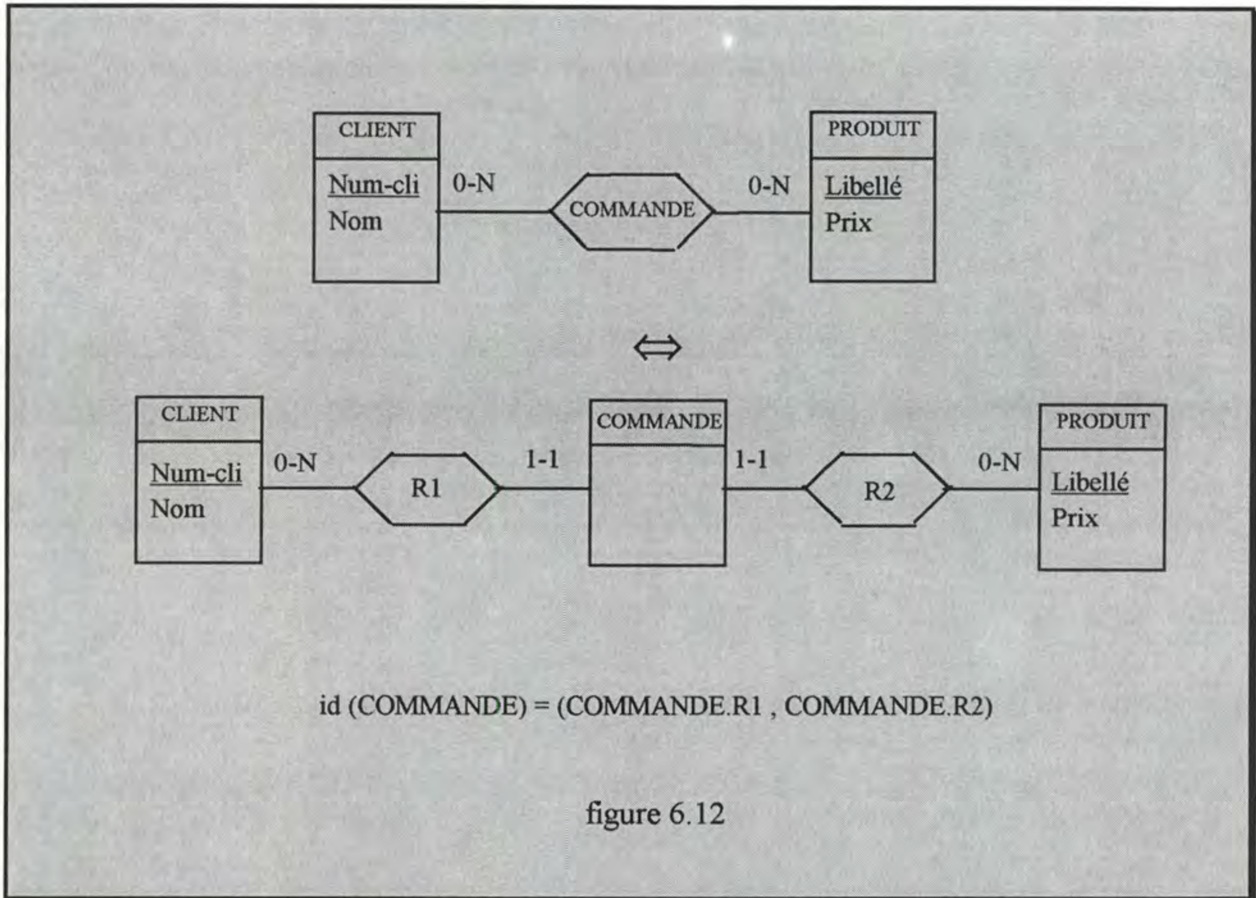



figure 6.12

Si on connaît le nombre maximum (soit m) de rôles que CLIENT (PRODUIT) joue dans COMMANDE, on peut appliquer une **transformation non classique** en remplacant le type d'assocations par une liste de m clés étrangères dans PRODUIT (CLIENT). Par exemple, sur le schéma supérieur de la figure 6.12, si la connectivité maximale de PRODUIT.COMMANDE était 3, on recopierait 3 fois l'attribut identifiant CLIENT.Num-cli dans PRODUIT. On obtiendrait alors le schéma conforme au relationnel de la figure 6.13.

Dans ce schéma, $Cli_1$, $Cli_2$ et $Cli_3$ représentent des attributs facultatifs, à cause de la connectivité minimale 0 pour PRODUIT.COMMANDE. Si cette connectivité avait été 1, on aurait eu la contrainte suivante : pour un tuple (Cli1 , Cli2 , Cli3), il y au moins un des éléments qui n'est pas la valeur nulle.

Nous donnons dans le tome 2 diverses recherches d'un type d'associations many-to-many transformé de manière non-classique (citons notamment la recherche sur les noms, dans les jointures, etc.).

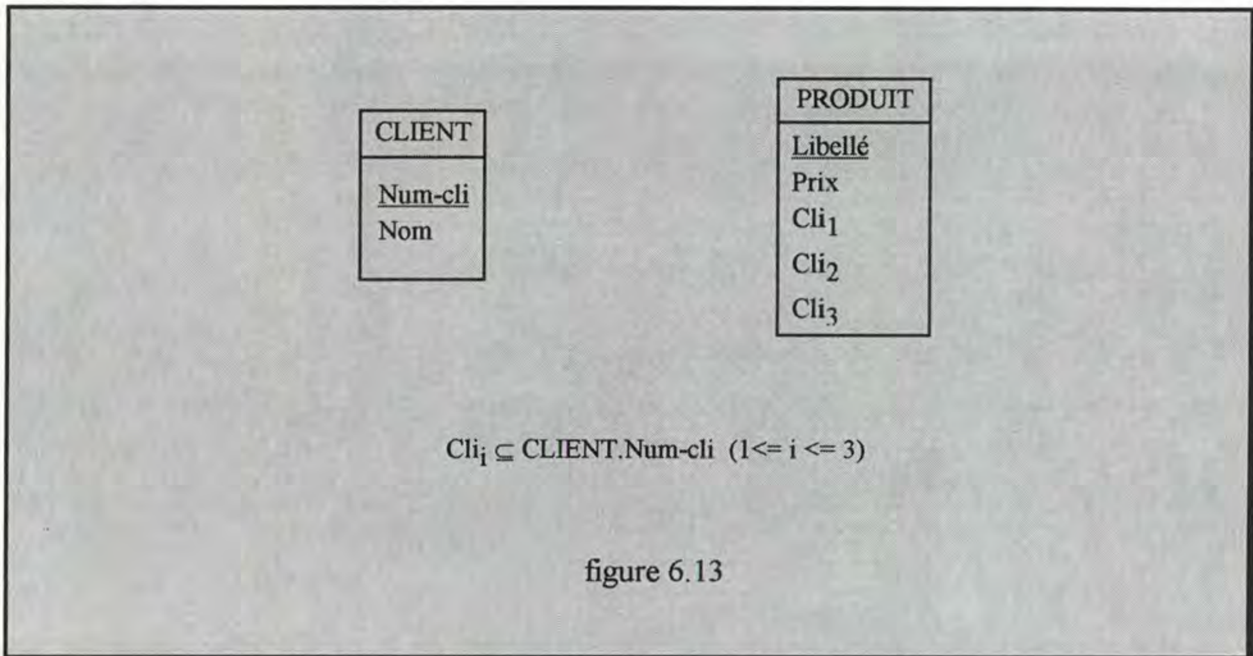



figure 6.13

## 6.4.1.2 Les types d'associations contenant un attribut

La **transformation classique** est la même que pour un type d'associations many-to-many. On remplace le type d'associations par un type d'entités. Les attributs du type d'associations deviendront des attributs du type d'entités. La recherche est similaire à celle d'un type d'associations many-to-many transformé de manière classique.

La **transformation non classique** peut s'appliquer dans le cas où le type d'associations transformé est un one-to-many. On déplace d'abord le (les) attribut(s) du type d'associations vers le type d'entités se trouvant du côté one. Ensuite, on applique la transformation classique d'une one-to-many (cf. 6.2.1). On aura une dépendance fonctionnelle de la clé étrangère vers le (les) attribut(s). La présence d'une telle dépendance fonctionnelle constitue un indice de l'application d'une telle transformation[20].

## 6.4.1.3 Les types d'associations de degré supérieur à 2

La **transformation classique** consiste dans ce cas-ci aussi à remplacer le type d'associations par un type d'entités. Celui-ci sera associé à tous les types d'entités qui étaient associés par le type d'associations transformé. Pour la recherche, on s'intéressera aux types d'entités impliqués dans plusieurs types d'associations par des rôles de connectivités 1-1.

[20]Cet indice ne pourra être retrouvé que si la normalisation n'a pas encore été faite.

## *6.4.2 Les attributs particuliers*

Nous allons rechercher ici les attributs facultatifs[21], les attributs décomposables et les attributs multivalués. Ces attributs ne sont pas directement implémentables en relationnel. Il faut donc appliquer des transformations lors de la conception.

Les transformations que nous allons évoquer ici sont tirées de [BEL93]. Nous allons nous contenter ici de donner un exemple de transformation pour chacun de ces types d'attributs. Nous développerons davantage de transformations dans le tome 2.

### 6.4.2.1 Les attributs facultatifs

Un attribut facultatif peut être transformé en un type d'entités. La figure 6.14 donne la représentation par instance. Une occurence du type d'entités `A''` représente une instance de l'attribut `a2`.

Pour détecter le résultat d'une telle transformation, on se basera sur les connectivités et le nombre d'attributs dans A" (il n'y en a qu'un).

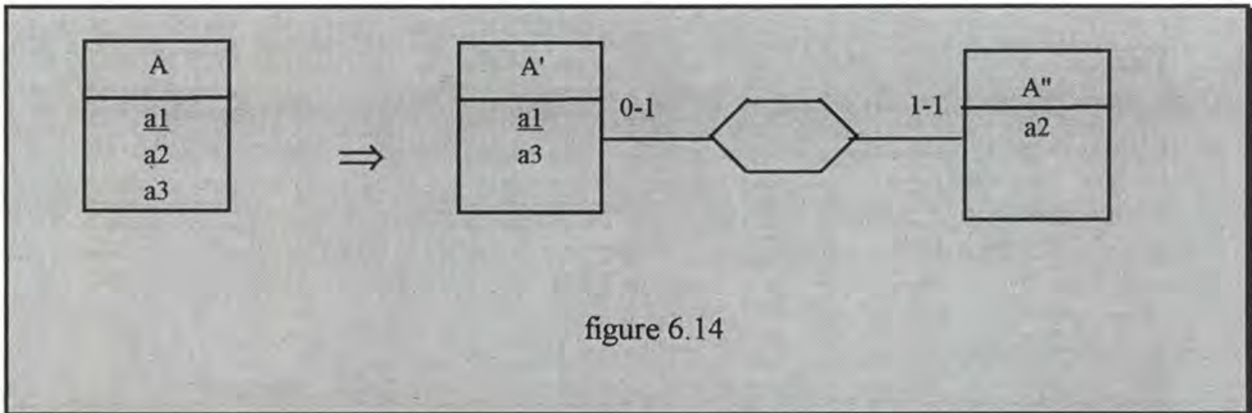


figure 6.14

### 6.4.2.2 Les attributs décomposables

Un attribut décomposable peut être transformé en type d'entités. Le schéma de la figure 6.15 illustre la représentation par valeur. Pour chaque instance du couple `(a21,a22)` de `A`, une instance de `A''` est créée. L'identifiant de `A''` est donc `(a21,a22)`.

La recherche se base sur les connectivités du type d'associations et sur le nombre d'attributs présents dans `A''`. On peut aussi faire une recheche sur les noms.

---

[21]Nous n'envisagerons ici que ceux qui ont été transformés en types d'entités. Les autres façons de représenter un attribut facultatif ont déjà été traitées (cf. 5.3).

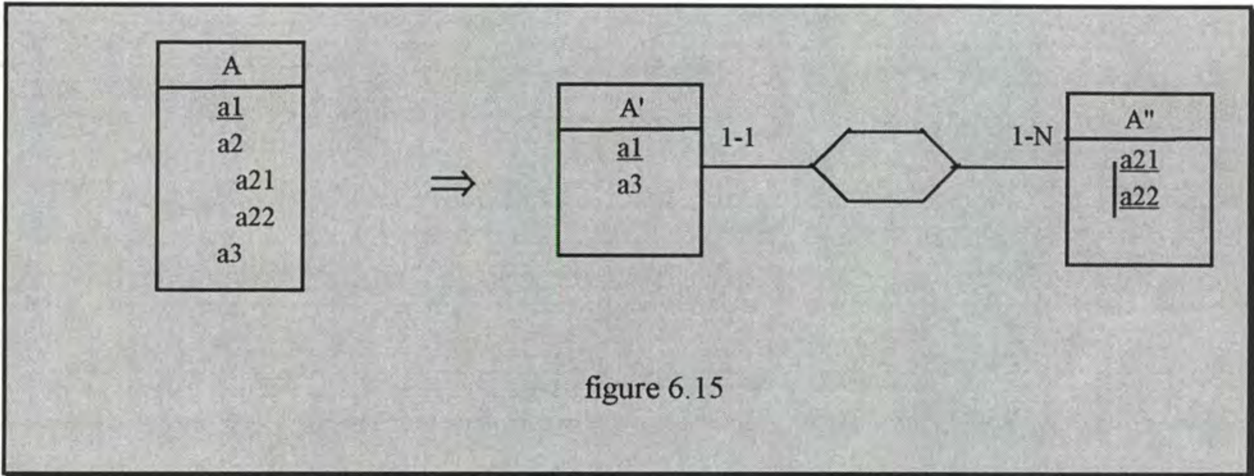



figure 6.15

## 6.4.2.3 Les attributs multivalués

Voyons la technique de l'instanciation. Sur la figure 6.16, on voit que cette technique constiste à représenter chaque instance de l'attribut multivalué par un attribut monovalué. Cette technique n'est applicable que si l'on connaît la valeur de N.

Pour la recherche, on s'intéressera aux types d'entités contenant un grand nombre d'attributs. La recherche se basera sur diverses caractéristiques de ces attributs (nom identique à un suffixe ou préfixe près, même domaine, etc.).

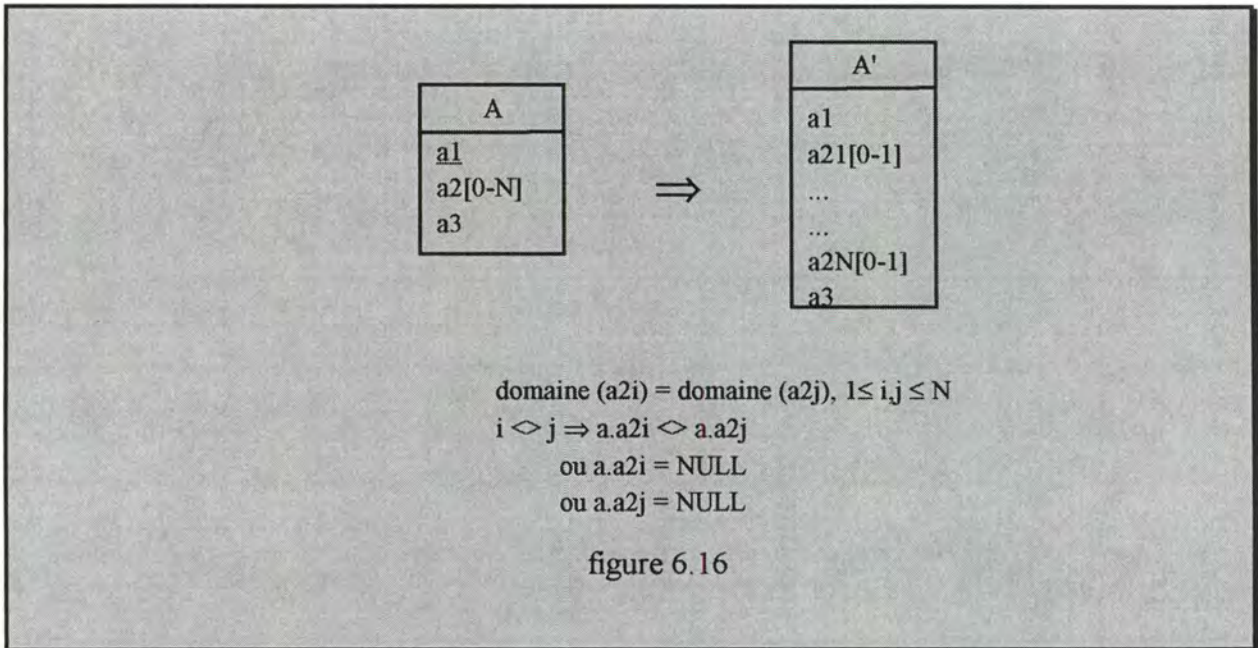



figure 6.16

## *6.4.3 Les contraintes d'intégrité sur les rôles*

On peut trouver dans un schéma conceptuel des contraintes d'intégrité sur les rôles. Nous allons voir comment ces contraintes ont pu être traduites dans les sources d'informations à notre disposition, et les moyens de les rechercher. Dans le tome 2, nous envisageons aussi les contraintes d'intégrité sur les types d'associations.

### 6.4.3.1 Les contraintes d'inclusion

Considérons le schéma de la figure 6.17.

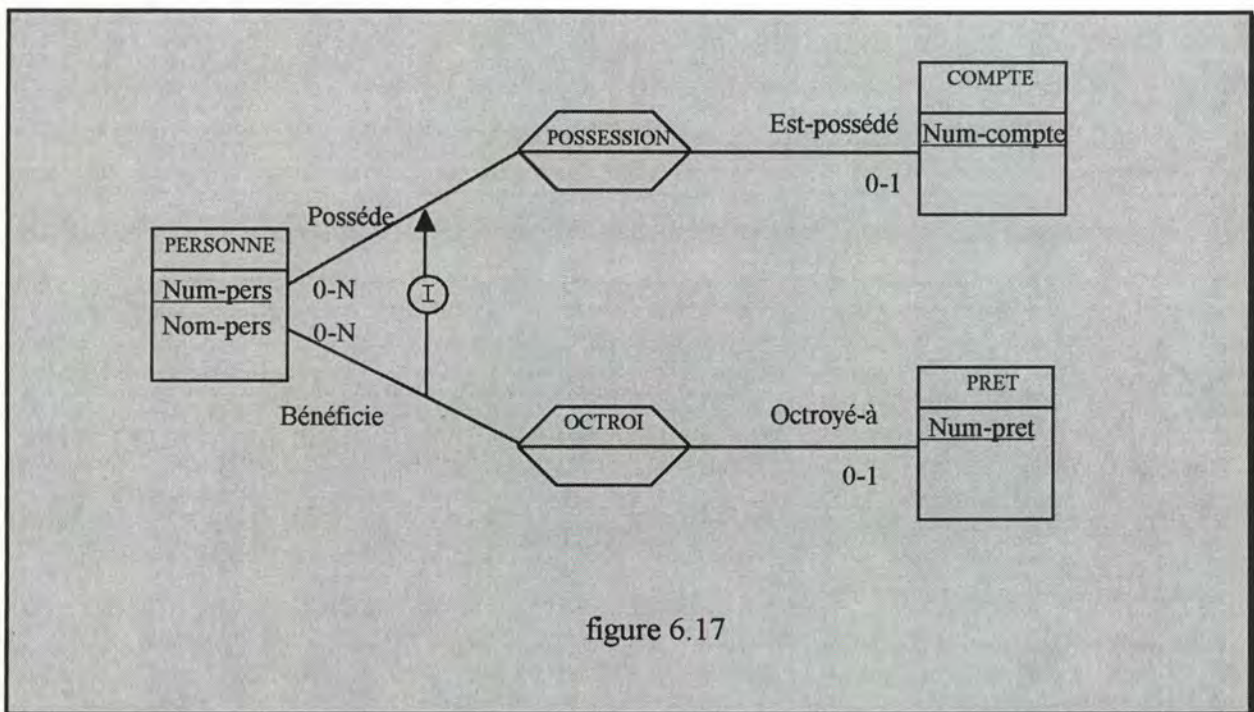


figure 6.17

On voit dans ce schéma qu'il y a une contrainte d'inclusion du rôle `Bénéficie` vers le rôle `Possède`. Cela signifie qu'une personne ne peut bénéficer d'un prêt que si elle possède au moins un compte. La traduction de ce schéma en relationnel est donnée à la figure 6.18.

```
PERSONNE(Num-pers, Nom-pers)
COMPTE(Num-compte, Num-pers)
PRET (Num-pret, Num-pers)

COMPTE.Num-pers ⊆ PERSONNE.Num-pers
PRET.Num-pers ⊆ PERSONNE.Num-pers

∀ pers ∈ PERSONNE :
(∃ pret ∈ PRET | pret.Num-pers = pers.Num-pers)
⇒ ∃ compte ∈ COMPTE | compte.Num-pers  = pers.Num-pers
```

figure 6.18

La dernière contrainte d'intégrité signifie que si une ligne de PERSONNE est reférencée par une ligne de PRET alors elle est référencée par une ligne de COMPTE.

Pour retrouver cette contrainte on peut :

1. Rechercher des triggers ou des configurations de requêtes qui lors de l'ajout d'une nouvelle valeur de clé étrangère[22] dans PRET (signifiant qu'une personne qui n'avait pas encore de prêt vient de se voir en octroyer un) vérifient que la ligne de PERSONNE référencée par cette clé est bien référencée par une ligne de COMPTE (on vérifie donc que la personne a bien un compte).
2. Rechercher des triggers ou des configurations de requêtes qui lors de la suppression d'une valeur de clé étrangère[23] dans COMPTE (signifiant qu'une personne n'a plus de compte), vérifient que la ligne de PERSONNE référencée par cette clé n'est pas référencée par une ligne de PRET (on vérifie donc que la personne n'avait pas de prêt).
3. Rechercher des triggers qui lors de la suppression d'une valeur de clé étrangère dans COMPTE référencant pers ∈ PERSONNE (signifiant qu'une personne n'a plus de

[22]Cela peut être réalisé lors d'un ajout d'une ligne dans PRET ou lors d'une modification de la clé étrangère de pret ∈ PRET qui étais à nul.

[23]Cela peut être réalisé lors d'une suppression d'une ligne de COMPTE ou lors d'une modification (mise à nul) de la clé étrangère de compte ∈ COMPTE

compte), suppriment la clé étrangère[24] des lignes de PRET qui référencent pers (on supprime les prêts de cette personne). Cette double suppression peut également apparaître dans des requêtes du programme d'application.

4. Rechercher des configurations de requêtes qui lors de l'ajout d'une nouvelle valeur de clé étrangère dans PRET, ajoutent une nouvelle valeur de clé étrangère[25] dans COMPTE (qui a la même valeur). Ce revient à faire en sorte que lorsqu'une personne se voit octroyer un premier prêt, on lui ouvre un compte.
5. Analyser l'extension (une recherche dans l'extension sera peu coûteuse puisqu'on connaît les tables et les attributs sur lesquels la recherche doit être effectuée).

Remarque

Le cas que nous avons traité peut être adapté si les connectivités des associations one-to-many du schéma conceptuel de départ avait été différentes.

### 6.4.3.2 Les contraintes d'égalité

Une contrainte d'égalité correspond à deux contraintes d'inclusion réciproques. On peut donc procéder d'une manière analogue à celle présentée ci-dessus.

### 6.4.3.3 Les contraintes d'exclusion

Considérons le schéma E/A de la figure 6.19.

---

[24]Cela peut être réalisé par la suppression d'une ou plusieurs lignes de PRET ou par la modification (mise à nul) de la clé étrangère de un ou plusieurs pret ∈ PRET

[25]Cela peut être réalisé par l'ajout d'une ligne dans COMPTE ou par la modification de la clé étrangère d'un compte ∈ COMPTE

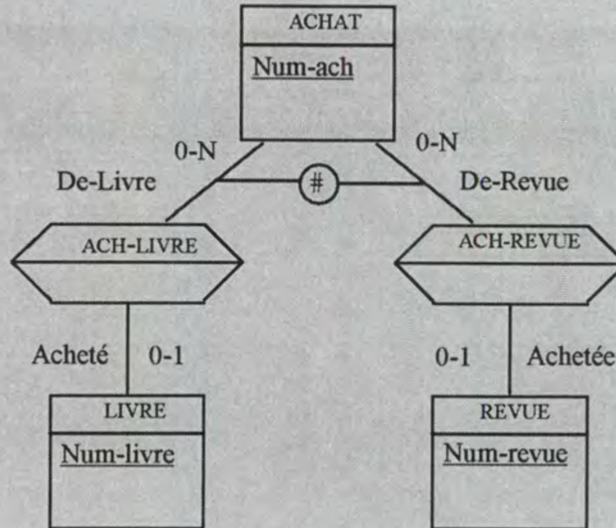



figure 6.19

La contrainte d'exlusion entre les rôles `De-livre` et `De-Revue` exprime le fait qu'un achat concerne soit un ou plusieurs livres, soit une ou plusieurs revues.

La figure 6.20 donne la traduction en relationnel du schéma de la figure 6.19.

```
ACHAT(Num-ach)
LIVRE(Num-livre, Num-ach)
REVUE(Num-revue, Num-ach)
LIVRE.Num-ach ⊆ ACHAT.Num-ach
REVUE.Num-ach ⊆ ACHAT.Num-ach

LIVRE.Num-ach ∩ REVUE.Num-ach = Ø
```

figure 6.20

La dernière contrainte d'intégrite signifie qu'un `achat` ∈ `ACHAT` ne peut être référencé en même temps par un `livre` ∈ `LIVRE` et par une `revue` ∈ `REVUE`.

On peut retrouver des traces de cette contrainte par :

* Une recherche des triggers ou des configurations de requêtes qui lors de l'ajout d'une nouvelle valeur de clé étrangère dans LIVRE vérifient que la ligne de ACHAT ainsi référencée, n'est déjà pas référencée par une ligne de REVUE (et reciproquement lors de l'ajout d'une nouvelle valeur de clé étrangère dans REVUE).
* Une recherche des triggers ou des configurations de requêtes qui lors de l'ajout d'une nouvelle valeur de clé étrangère dans LIVRE (référençant achat ∈ ACHAT) suppriment les clés étrangères de REVUE référencant les lignes de achat (et réciproquement lors de l'ajout d'une nouvelle valeur de clé étrangère dans REVUE).
* Une analyse de l'extension.

Remarque

Le cas que nous venons de traiter peut être adapté à des connectivités différentes sur le schéma de départ.

### 6.4.3.4 Les contraintes d'existence

Considérons le schéma conceptuel de la figure 6.21.

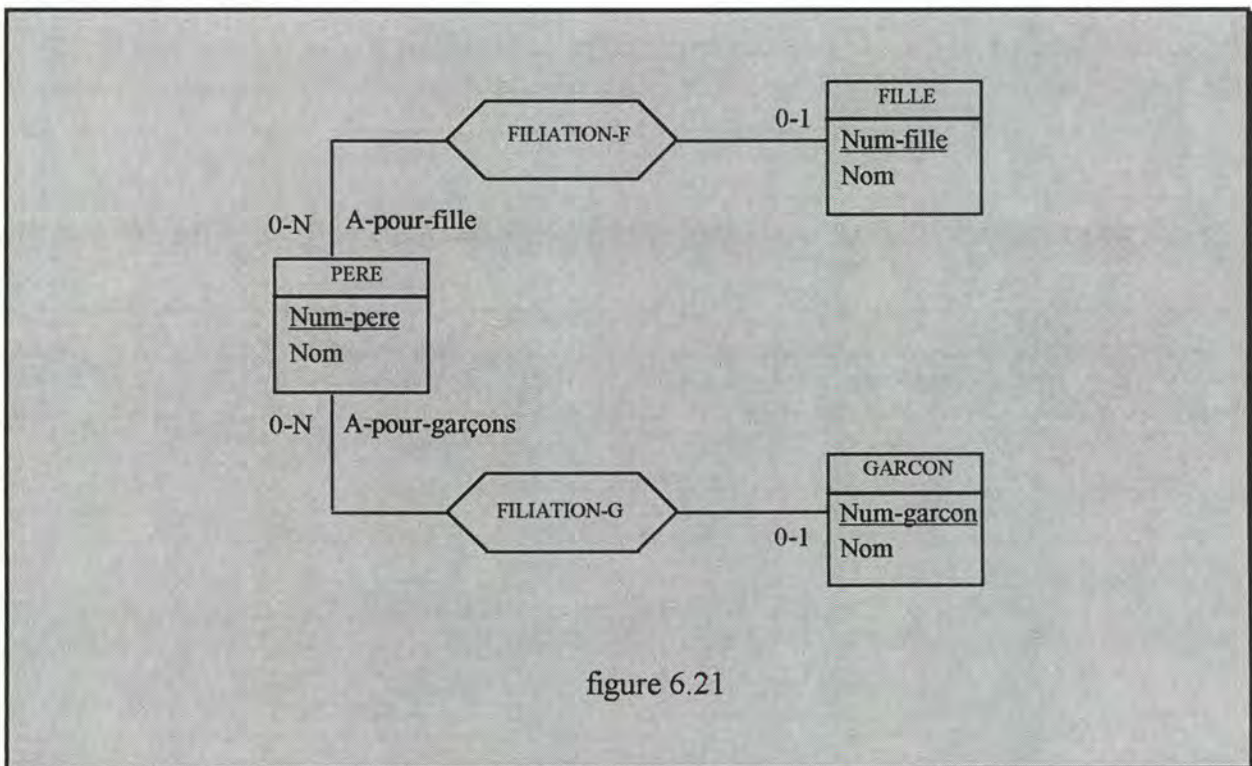



figure 6.21

Et la contrainte d'intégrité que Pere joue au moins l'un des deux rôles A-pour-fille ou A-pour-garçons. La traduction de ce schéma en relationnel est donnée à la figure 6.22.

```
PERE(Num-pere, Nom)
FILLE(Num-fille, Nom, Num-pere)
GARCON(Num-garcon, Nom, Num-pere)

FILLE.Num-pere ⊆ PERE.Num-pere
GARCONS.Num-pere ⊆ PERE.Num-pere
PERE.Num-pere ⊆ (FILLE.Num-pere ∪ GARCON.Num-pere)
```

figure 6.22

La dernière contrainte signifie que chaque ligne de `PERE` est référencée par au moins une ligne de `FILLE` ou de `GARCON`.

Pour retrouver cette contrainte d'intégrité, on peut :

- Rechercher des triggers ou des configurations de requêtes qui, lors de la suppression d'une valeur de clé étrangère dans `FILLE` (`GARCON`), vérifient que cette valeur existe toujours dans la clé étrangère d'une ligne de `GARCON` (`FILLE`).
- Rechercher des configurations de requêtes qui lors d'une suppression d'une valeur de clé étrangère dans `FILLE` (`GARCON`), ajoutent cette valeur dans `GARCON.Num-pere` (`FILLE.Num-pere`).
- Analyser l'extension

Remarque

Le cas que nous venons de traiter peut être adapté à des connectivités différentes.

## 6.4.4 Les relations IS-A

Dans les premières phases du processus de reverse engineering, nous sommes arrivés à un schéma conceptuel ne contenant que des types d'associations pour faire le lien entre les types d'entités. Nous allons essayer ici de retrouver les relations IS-A du schéma conceptuel à partir des entités et associations retrouvées.

Les relations IS-A ne sont pas gérées par les SGBD relationnels actuels. Dès lors, dans un processus de conception de base de données, on est amené à les traduire dans le schéma E/A de base. Il existe des transformations pour cela. Nous allons en examiner trois, tirées de [HAI89a], que nous adapterons dans certains cas au modèle relationnel. Il s'agit de la transformation en type d'associations, de la représentation des types d'entités spécifiques et de la

représentation des types d'entités génériques. Pour chaque transformation, nous rechercherons des indices permettant de trouver le résultat de son application.

Prenons comme exemple de départ le schéma de la figure 6.23.

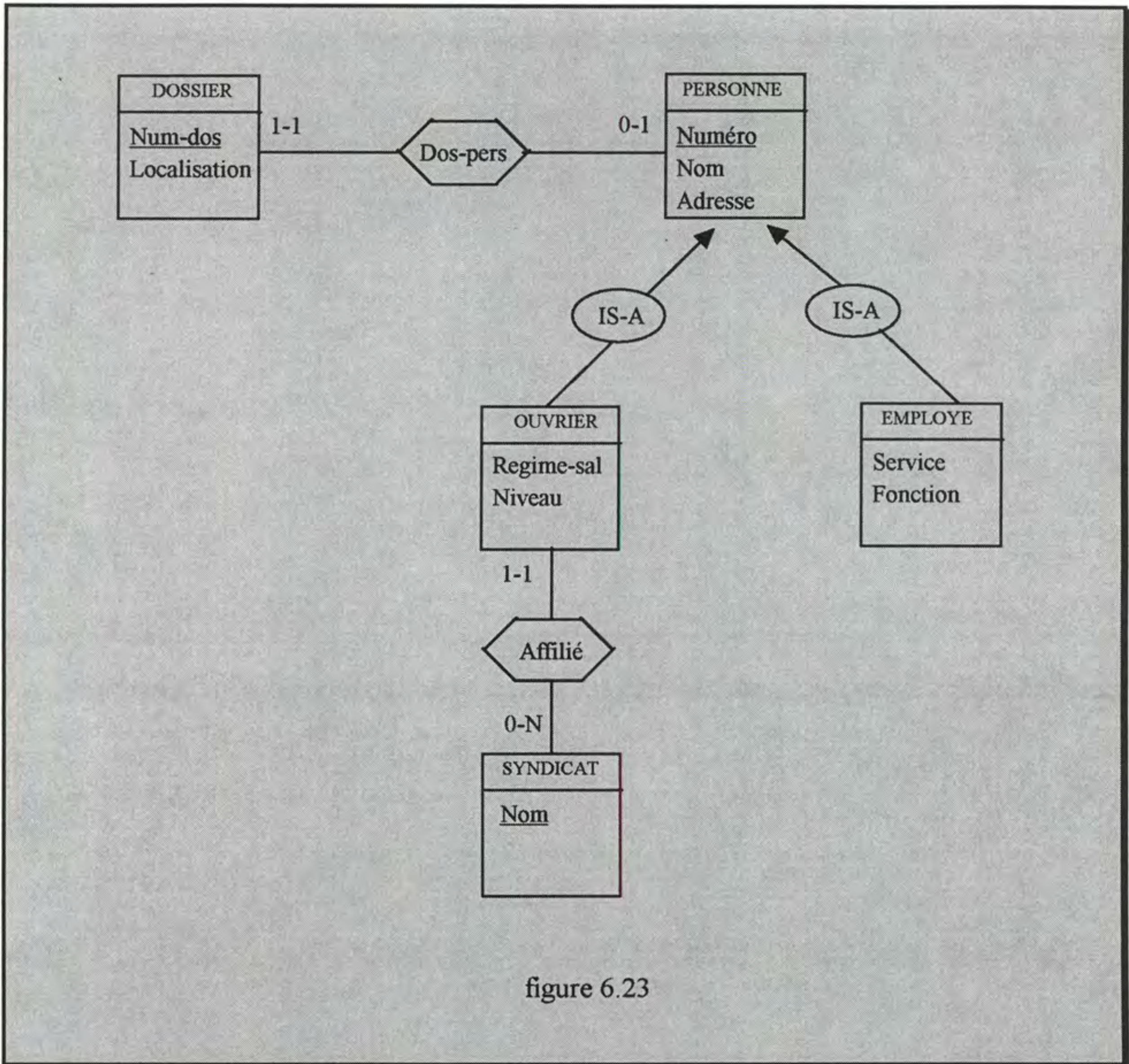



figure 6.23

## 6.4.4.1 Transformation en types d'associations

### Définition

On représente directement chaque type d'entités de la hiérarchie, et on définit des types d'associations mettant en correspondance les entités qui représentent le même objet du réel perçu. On aura le schéma de la figure 6.24.

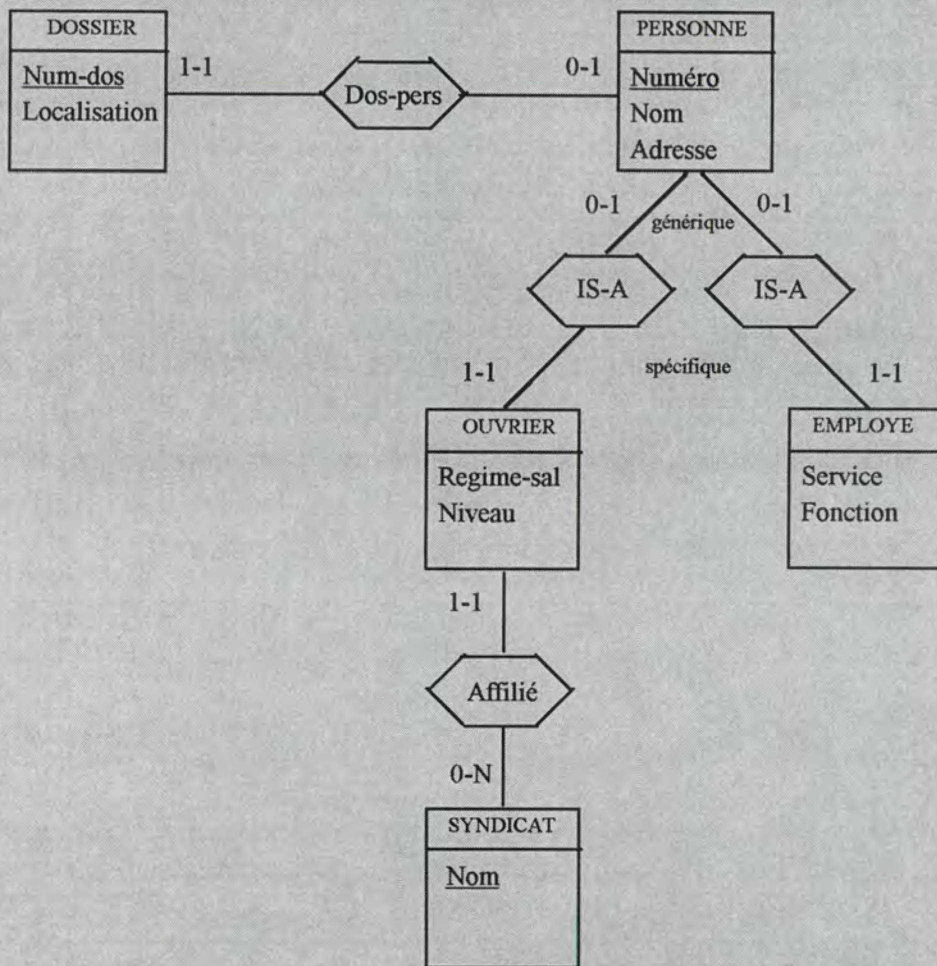



figure 6.24

Remarque

Les noms des types d'associations et des rôles devraient normalement être différenciés.

Cette solution est la plus naturelle en apparence, mais elle pose le problème de la représentation multiple des objets du réel perçu. En effet, un employé est représenté par une entité de type `EMPLOYE` et une entité de type `PERSONNE`. Cela augmente la complexité de la gestion des données.

## Recherche de la relation IS-A

On peut tout d'abord dire que si on retrouve une structure de T.E. dans laquelle un T.E. est lié à d'autres par des T.A. one-to-one sans attributs, il y a une chance pour qu'il s'agisse de relations IS-A. Etant donné que la seule façon de transformer une IS-A en types d'associations

est d'utiliser des connectivités one-to-one, les types d'associations autres que les one-to-one pourront être écartés de la recherche.

Un indice supplémentaire est le fait que les connectivités du côté des sous-type sont toujous 1-1.

Un autre indice est que `OUVRIER` et `EMPLOYE` n'ont pas d'identifiant propre : ils sont identifiés par le rôle joué par `PERSONNE`. Or souvent, un type d'entités a son propre identifiant.

On aura aussi un indice si on trouve des comportements imposant la suppression d'une personne lors de la suppression d'un ouvrier et/ou d'un employé.

On peut aussi trouver des indices dans les vues. Il se peut en effet que le concepteur ait défini une vue reprenant les personnes qui sont des ouvriers et une vue reprenant les personnes qui sont des employés. On aurait les vues suivantes :

```
CREATE VIEW PERS-OUVRIER (Numero , Nom , Adresse , Regime-sal , Niveau)
AS SELECT PERSONNE.Numero , PERSONNE.Nom , PERSONNE.Adresse ,
          OUVRIER.Regime-sal, OUVRIER.Niveau
   FROM PERSONNE , OUVRIER
   WHERE OUVRIER.Numero = PERSONNE.Numero
```

```
CREATE VIEW PERS-EMPLOYE (Numero , Nom , Adresse , Service , Fonction)
AS SELECT PERSONNE.Numero, PERSONNE.Nom, PERSONNE.Adresse, EMPLOYE.Service,
          EMPLOYE.Fonction
   FROM PERSONNE , EMPLOYE
   WHERE EMPLOYE.Numero = PERSONNE.Numero
```

Si on a retrouvé ces indices, on peut effectuer la transformation inverse, c-à-d transformer les types d'associations en relations IS-A.

### Recherche du type de structure

Les contraintes sur les rôles joués par le T.E. générique permettent de retrouver le type de structure. La disjonction correspond à la présence de contraintes d'exclusion entre tous les rôles joués par le T.E. générique dans ses associations avec ses T.E. spécifiques. Une structure couvrante correspondra à une contrainte d'existence entre ces mêmes rôles.

## 6.4.4.2 Représentation des types d'entités spécifiques

### Définition

On ne représente que les T.E. spécifiques, et on leur attribue par héritage descendant les attributs et les rôles du T.E. générique. On obtient alors le schéma de la figure 6.25.

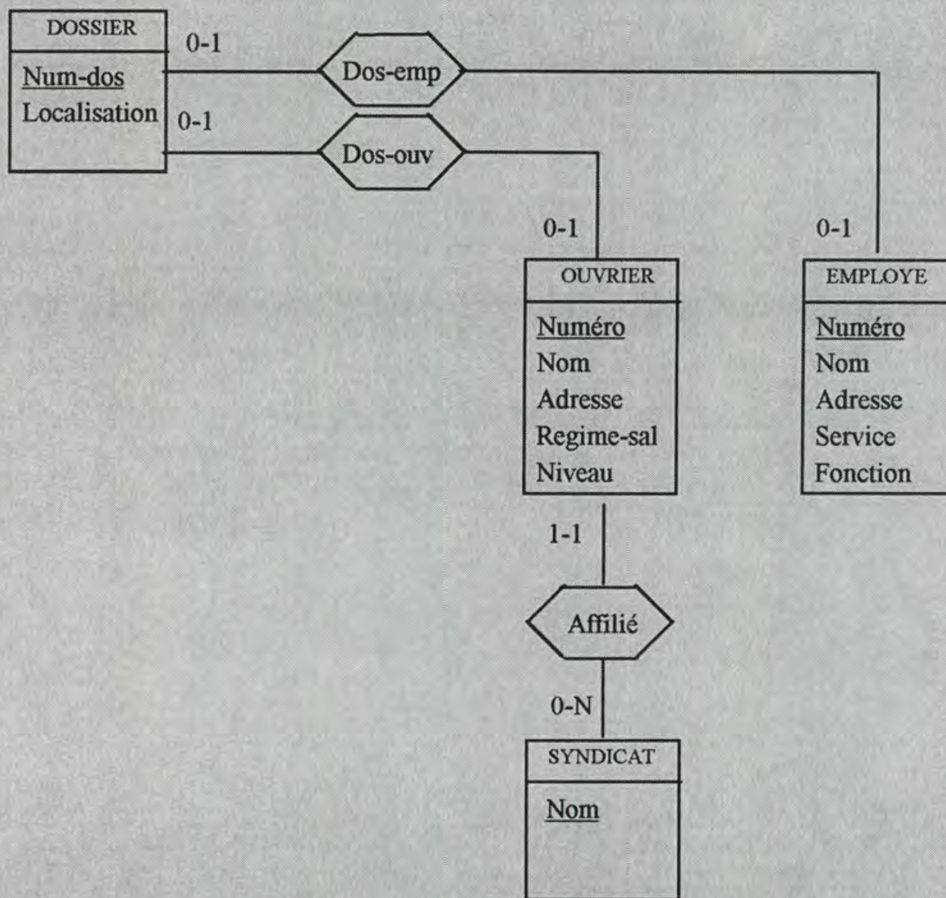



figure 6.25

Il y a une contrainte de rôles exclusifs imposée à `DOSSIER`. En effet, un dossier concerne un employé ou un ouvrier, mais pas les deux en même temps. Il y a aussi une contrainte d'identification qui fait des attributs `Numéro` de `OUVRIER` et `EMPLOYE` un identifiant global pour les deux types d'entités.

Cette solution respecte le principe de représentation unique des objets du réel perçu, mais il faut gérer les contraintes d'identifiant global et de rôles exclusifs.

## Recherche de la relation IS-A

Tout d'abord, si on trouve deux ou plusieurs T.E. ayant quelques attributs identiques (dont l'identifiant), et quelques attributs spécifiques, il y a une chance pour qu'il s'agisse de la traduction de relations IS-A par représentation des T.E. spécifiques. On peut détecter deux attributs identiques par une recherche sur les noms et par une recherche sur les types des attributs. Pour la recherche sur les noms, on regardera si deux attributs ont exactement le même nom, ou s'ils sont différents à un suffixe ou un préfixe près (exemple : "Numéro-ouv" et "Numéro-emp"). La

recherche des correspondances de types en elle-même offre un niveau de certitude très peu élevé. Cependant, elle peut être utile pour confirmer ou réfuter des hypothèses tirées de l'étude sur les noms.

Un autre indice est que ces T.E. sont souvent associés aux mêmes T.E..

On peut ensuite s'intéresser à l'identifiant de ces T.E.. En effet, il s'agit d'un identifiant global défini sur l'ensemble des entités de ces types. Nous en reparlerons dans la recherche du type structure.

On peut aussi effectuer une recherche dans les vues. On cherchera les vues qui isolent le T.E. générique des T.E. spécifiques. On aura par exemple

```
CREATE VIEW PERSONNE (Numero , Nom , Adresse)

AS (SELECT Numero, Nom , Adresse
    FROM OUVRIER)

      UNION

   (SELECT Numero, Nom , Adresse
    FROM EMPLOYE)
```

Lorsqu'on a retrouvé tous ces indices, on peut appliquer la transformation en relations IS-A. Pour cela, il suffira de regrouper les attributs communs à tous ces T.E. pour former le T.E. générique, et les groupes d'attributs restants formeront les T.E. spécifiques. Les T.A. "communs" aux T.E. spécifiques deviendront T.A. du T.E. générique, et chaque T.E. spécifique gardera ses "propres" T.A..

## Recherche du type de structure

Une structure non-couvrante aurait été traduite dans le schéma de départ en créant un T.E. supplémentaire regroupant les personnes qui ne sont ni des ouvriers ni des employés. Ce T.E. n'aurait possédé que 3 attributs : `Numéro`, `Nom` et `Adresse`. On peut donc dire que si, en plus des T.E. spécifiques, on trouve un T.E. qui possède les attributs communs aux T.E. spécifiques et qui est relié à un T.A. équivalent aux T.A. 'communs' aux T.E. spécifiques, alors on peut supposer que la structure IS-A est une structure non-couvrante.

Voyons maintenant comment retrouver une structure de disjonction. Il faut pour cela s'intéresser à l'identifiant global aux T.E. représentant les T.E. spécifiques. On peut se baser sur une recherche dans l'extension ou sur une recherche dans les comportements imposés aux données. La recherche dans l'extension se fera de la façon suivante : si les valeurs de l'identifiant sont distinctes entre les T.E., on peut supposer que l'on a à faire à une structure de disjonction. Pour la recherche dans les comportements, on cherchera les comportements de gestion de l'exclusion entre les T.E. spécifiques (donc entre leurs identifiants). On pourra trouver un comportement concernant l'ajout dans chacun des T.E.. Lors de cet ajout, on vérifiera qu'il n'existe pas un élément dans les autres T.E. ayant la même valeur d'identifiant que l'identifiant de l'élé-

ment ajouté. Il se peut aussi que, dans le cas d'identifiants de type numérique, on trouve des comportements de gestion d'assignation automatique de valeur à l'identifiant ajouté. Par exemple, si on ajoute un employé, on pourrait prendre comme identifiant le plus grand numéro parmi les numéros de `EMPLOYE` et les numéros de `OUVRIER`, incrémenté de 1.

## 6.4.4.3 Représentation des types d'entités génériques

### Définition

On ne représente que le type d'entité générique, et on lui attribue par héritage ascendant les attributs et les rôles des types d'entité spécifiques. On obtient le schéma de la figure 6.26.

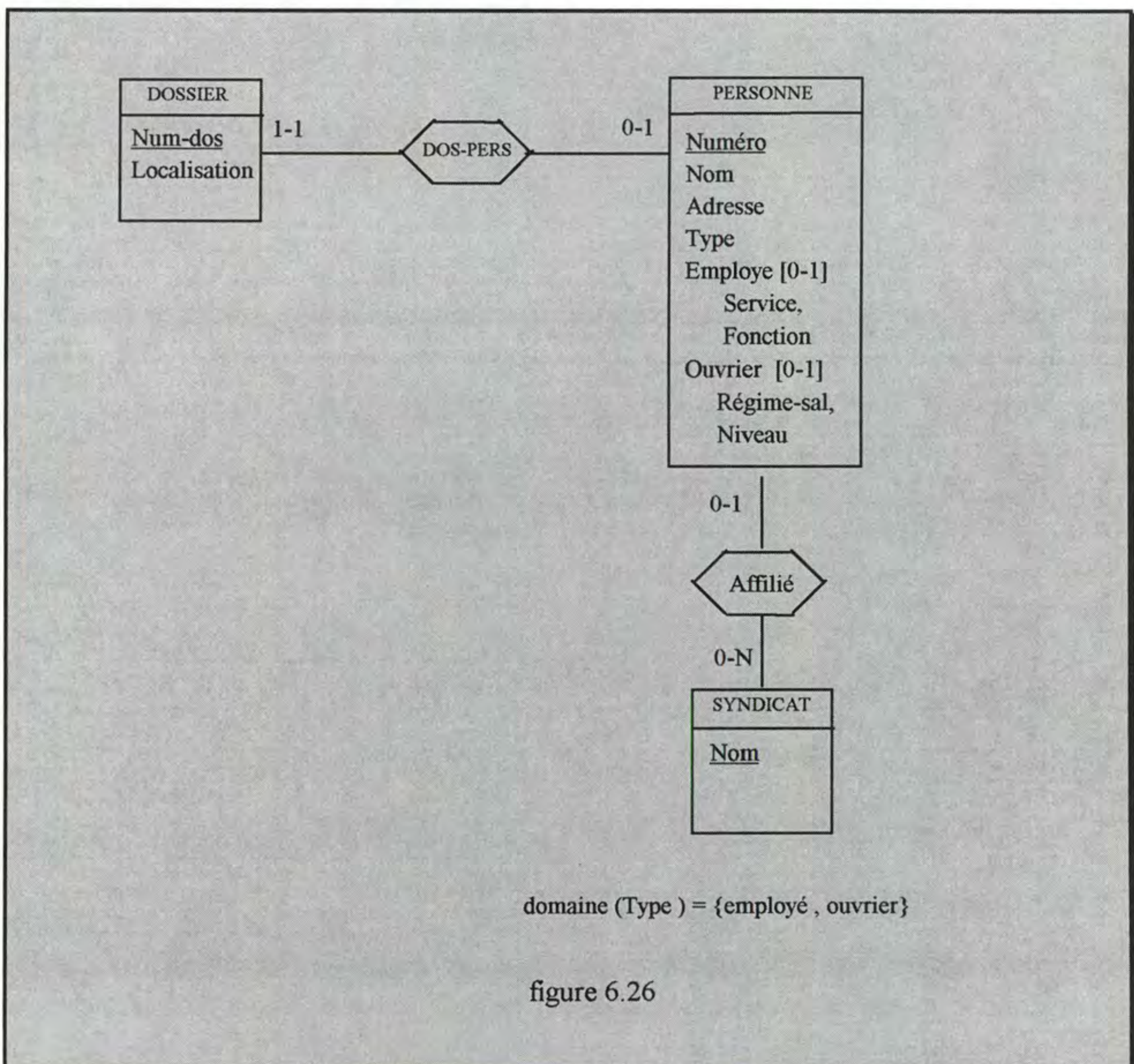



figure 6.26

Les attributs hérités de chaque T.E. spécifique ont été groupés en attributs décomposables et facultatifs. L'attribut *flag* `Type`, de domaine `{Employé , Ouvrier}`, sert à déterminer le

type de chaque entité. Tout rôle hérité devient facultatif (c-à-d de connectivité minimale 0). Il faut imposer des contraintes de structures, en fonction de la valeur que peut prendre l'attribut `Type`. Il y aura par exemple

```
∀ p ∈ personne, p.type = 'employé' => p.employé existe
                                       et p.ouvrier n'existe pas
                                       et p ne joue pas de rôle dans affilié
```

Cette solution respecte le principe de représentation unique des objets du réel perçu, mais elle introduit des contraintes structurelles complexes.

Si on admettait qu'une personne soit à la fois un ouvrier et un employé, le domaine de `Type` serait élargi. On aurait par exemple `{employé , ouvrier , emp-ouv}`.

S'il était possible qu'une personne ne soit ni un employé ni un ouvrier, l'attribut `Type` serait facultatif.

Notons enfin que l'attribut `Type` pourrait aussi être remplaçé par deux attributs de type booléen, exprimant chacun le fait que l'occurence du type d'entités `PERSONNE` représente ou non un employé ou un ouvrier.

## Recherche de la relation IS-A

Ici, on cherchera d'abord des T.E. dans lesquels on trouve des attributs qui sont décomposables, facultatifs et non-répétitifs. Parmi ces T.E., on en cherchera de deux sortes :

- Soit ceux dans lesquels on trouve un attribut flag dont le domaine est composé de peu de valeurs. Il se peut que le domaine soit composé de valeurs significatives (comme dans l'exemple, où le domaine de l'attribut `Type` est `{Employé , Ouvrier}`). Les valeurs du domaine peuvent aussi être des numéros ou des caractères non significatifs. L'utilisateur devra toujours déterminer la signification de ces valeurs.
- Soit ceux dans lesquels on trouve un (ou plusieurs) attributs flag dont le domaine est booléen. On pourrait par exemple en avoir un pour `EMPLOYE` et un pour `OUVRIER`. Le nom de ces attributs flags devrait être proche du nom du T.E. spécifique dont ils déterminent la présence.

Un autre indice est le fait que lorsqu'on accède à ce genre de T.E., on accède aux attributs facultatifs décomposables non-répétitifs en fonction de la valeur du (des) attribut(s) flag. Par exemple, si on veut accéder à un employé, on fait un test pour voir si la valeur de l'attribut `Type` est 'employé'.

Voyons maintenant les indices que l'on pourrait retrouver dans les vues. On pourrait faire une vue reprenant les informations sur les employés et une vue reprenant les informations sur les ouvriers. On aura par exemple :

```
CREATE VIEW EMPLOYE (Numero, Nom, Adresse, Service, Fonction)
AS SELECT Numero, Nom, Adresse, Service, Fonction
FROM PERSONNE
WHERE Type = 'employé'
```

```
CREATE VIEW OUVRIER (Numero, Nom, Adresse, Regime-sal, Niveau)
AS SELECT Numero, Nom, Adresse, Regime-sal, Niveau
FROM PERSONNE
WHERE Type = 'ouvrier'
```

On pourrait aussi avoir des vues reprenant les informations sur les personnes, les employés et les ouvriers :

```
CREATE VIEW VUE-PERSONNE (Numero, Nom, Adresse)
AS SELECT Numero, Nom, Adresse
FROM PERSONNE
```

```
CREATE VIEW EMPLOYE (Numero, Service, Fonction)
AS SELECT Numero, Service, Fonction
FROM PERSONNE
WHERE Type = 'employé'
```

```
CREATE VIEW OUVRIER (Numero, Regime-sal, Niveau)
AS SELECT Numero, Regime-sal, Niveau
FROM PERSONNE
WHERE Type = 'ouvrier'
```

Si ces indices ont été retrouvés, on peut effectuer la transformation en relation IS-A. Les attributs facultatifs et répétitifs deviendront les T.E. spécifiques, et le reste des attributs (sauf l'attribut flag) formeront le T.E. générique.

## Recherche du type de structure

Le type de structure dépendra du (des) attribut(s) flag.

- Considérons le cas où on a un seul attribut flag.. Si on trouve un élément du domaine de cet attribut dont le nom est la concaténation des noms (ou d'une partie des noms) des autres éléments (par exemple `emp-ouv`), on n'a normalement pas une structure de disjonction. Si l'attribut flag est obligatoire on aura une structure couvrante. S'il est facultatif, on aura une structure non-couvrante.
- Considérons le cas où on a plusieurs attributs flag booléens. Dans l'exemple, si à chaque occurence de `PERSONNE`, il n'y a qu'un seul attribut flag qui prend la valeur "vrai", on a une structure de disjonction. Si on trouve une occurence de `PERSONNE` pour laquelle tous les attributs flag ont la valeur "faux", alors ce n'est pas une structure couvrante (en effet, l'occurence de `PERSONNE` représenterait alors une personne qui n'est ni un employé ni un ouvrier).

# CONCLUSION

Nous avons proposé dans cet ouvrage des moyens permettant de retrouver les structures conceptuelles d'une BD relationnelle SQL. Autrement dit, il s'agissait de retrouver le schéma conceptuel (exprimé dans le modèle E/A) d'une BD.

***Les problèmes***

Nous avons mentionné au début du travail les deux principaux problèmes rendant difficile le RE de BD relationnelles.

Le premier problème est la correspondance multiple entre le modèle relationnel et le modèle E/A. A un schéma relationnel peuvent correspondre plusieurs schémas conceptuels. Il faut donc effectuer des choix tout au long du processus de reverse engineering. Ces choix ne se basent pas que sur des connaissances formelles. Ils se basent aussi sur des connaissances sur les habitudes de conception et de programmation, sur le domaine des applications, etc. Il en résulte que le processus de reverse engineering ne peut être totalement automatisé. La participation de l'utilisateur est donc indispensable. Comme nous l'avons vu tout au long de ce travail, nous proposons des recherches permettant de trouver des indices qui n'offrent généralement pas un niveau de certitude maximal. Ces indices doivent donc être confirmés par l'utilisateur.

Le deuxième problème vient des possibilités offertes pour la gestion des contraintes d'intégrité. Celles-ci ne seront pas nécessairement toutes déclarées explicitement dans le schéma de la BD. Elles pourront être gérées différemment. Il faut donc utiliser d'autres sources d'informations que la déclaration du schéma. Nous avons considéré diverses sources d'informations supplémentaires. La principale est l'implémentation des comportements imposés aux données. Les développements techniques du tome 2 sont en grande partie consacrés à l'analyse de ces comportements.

***Recherches proposées***

Nous allons maintenant rappeler brièvement de quelle façon nous avons envisagé le reverse engineering.

La méthode sur laquelle nous nous sommes basés est constituée de deux phases, l'extraction et la conceptualisation des structures de données.

- La phase d'extraction des structures de données fournit un schéma de la base de données conforme au modèle relationnel. Au cours de cette phase, nous examinons dans les sources d'information à notre disposition quels sont les signes distinctifs permettant de détecter la présence de telle ou telle contrainte.

- La phase de conceptualisation aboutit à un schéma conceptuel clair, normalisé et naturel de la BD. Au cours de cette phase, nous avons examiné les transformations qui ont pu être utilisées lors de la conception, et les indices permettant de retrouver le résultat de ces transformations. Ces indices sont des heuristiques permettant de savoir là où il faut appliquer une transformation inverse.

*Avantages et inconvénients*

Examinons maintenant les avantages et les inconvénients des recherches que nous avons proposées.

**Avantages**

- Le principal avantage de ces recherches se situe au niveau de leur variété.

  En effet, même si ces recherches ne permettent de trouver que des indices, la diversité des sources d'information utilisées augmente les chances de retrouver le schéma conceptuel le plus proche possible de la réalité. Notons que la recherche dans les comportements imposés aux données semble assez nouvelle. Nous avons en effet trouvé très peu de recherches de ce genre dans la littérature.
- De plus, ces recherches (à la différence de la plupart des méthodes de la littérature) permettent de prendre en compte des schémas obtenus de façon non classique.
- Un autre avantage des recherches proposées est qu'elles ne se basent sur aucune hypothèse de départ. La plupart des méthodes de la littérature considèrent que le schéma relationnel a été retrouvé. Autrement dit, elles considèrent que la phase d'extraction des données a été réalisée. On a pu se rendre compte à la lecture de ce travail qu'il s'agit d'une hypothèse assez forte. Retrouver le schéma conforme au relationnel n'est pas si facile qu'il n'y paraît.

**Inconvénients**

- Un inconvénient des recherches proposées est leur manque de certitude. Nous l'avons déjà expliqué, nous proposons des recherches d'indices dont le niveau de certitude n'est pas maximal. Nous ne voyons pas de quelle façon remédier à ce problème, si ce n'est en combinant les recherches dans diverses sources d'information. La participation de l'utilisateur sera cependant toujours nécessaire.
- Un autre inconvénient des recherches proposées se situe au niveau des moyens de recherche dans les sources d'informations. Si on considère les textes sources de la BD et des applications, il faut bien admettre que la recherche de structures particulières (triggers, configurations de requêtes, etc.) est assez longue et fastidieuse. En effet, ces textes sont généralement assez longs, et il est probable qu'ils ne soient pas très faciles à lire. On peut remédier partiellement à ce problème en utilisant un outil d'analyse de textes semblable à celui que nous proposons dans le tome 2.

***Reverse engineering et conception de bases de données***

A la lecture de cet ouvrage, on peut se rendre compte que le reverse engineering de BD est intimement lié à la conception de BD. En effet, la recherche de structures conceptuelles passe nécessairement par une étape dans laquelle on se pose la question "de quelle façon peut-on traduire une telle structure?". C'est en fonction de la réponse à cette question que des recherches peuvent être proposées. Plus on envisagera de façons de traduire les structures conceptuelles, plus on aura de chances d'en retrouver beaucoup.

Une grande partie du travail a donc été de montrer comment pouvait être réalisée la conception d'une base de données. Notre travail (surtout le tome 2) peut, à la limite, servir tant à la personne qui fait du reverse engineering qu'au concepteur d'une base de données.

***Travail en perspective***

Pour terminer, envisageons ce qu'il reste à faire. Une étape serait la mise au point d'un outil d'aide aux recherches. L'ébauche d'outil d'analyse de textes que nous avons proposé dans ce travail s'inscrirait parfaitement dans un tel outil. De plus, des algorithmes interactifs de recherche pourraient être développés et s'inscrire dans le cadre de cet outil.

Il serait aussi utile d'envisager un ordre de recherches adéquat. Les recherches ont en effet été présentées dans cet ouvrage dans un ordre "théorique", en suivant les étapes de la méthode sur laquelle nous nous sommes basés. Une application pratique de cette méthode ne suivrait pas nécessairement cet ordre.

# REFERENCES


**[AND94]** : M. Andersson, "Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering", EPFL, 1994

**[BAT92]** : C. Batini, C. Ceri, S.B. Navathe, "Conceptual Database Design", Benjamin/Cummings, 1992

**[BEL93]** : J-F. Bellem et X. Deflorenne, "Rétro-ingénierie de bases de données relationnelles et CODASYL", Mémoire de Licence et Maîtrise en Informatique, FUNDP, 1993

**[BOD89]** : F. Bodart et Y. Pigneur ,"Conception Assistée des Systèmes d'Information", Ed. Masson, 1989

**[CAR83]** : C.R. Carlson et A.K. Arora, "UPM : a formal tool for expressing database update semantics", in Proc. 3rd Entity-Relationship Conference, North Holland, 1983

**[CAS93]** : M.G. Castellanos, "Semiautomatic semantic enrichment for the integrated acces in interoperable databases", Thèse de Doctorat, Université Polytechnique de Catalogne, 1993

**[CHE76]** : P.P. Chen, "The Entity-Relationship Model : Toward a Unified View of Data", ACM TODS, vol. 1, n°1, 1976

**[COD70]** : E.F. Codd, "A Relationnal Model of Data for Large Shared Data Banks", Comm. ACM, vol. 13, N° 6, Juin 1970

**[DAV85]** : K.H. Davis, K.A. Adarsh, "A Methodology for Translating a Conventional File System into an Entity-Relationship", in Proc. of E/R Approach, 1985

**[DAV88]** : H.K. Davis et A.K. Arora, "Converting a Relationnal Database Model into an ER model", in Proc. 6th Int. Conf. on Entity-Relationship Approach, North Holland, 1988

**[FON92]** : M. Fonkam et W.Gray, "An approach to eliciting the semantics of relationnal databases", in Proc. CAISE 1992

**[HAI89a]** : J-L. Hainaut, "Bases de données et bases de connaissances en gestion des organisations", Cinquième école d'automne de bases de données, Port Barcarès, 1989

**[HAI89b]** :J-L. Hainaut,"Introduction à la Théorie Relationnelle des Bases de Données", FUNDP, Novembre 1989

**[HAI91]** : J-L. Hainaut, "Entity-generating Schema Transformation for Entity-Relationship Models", in Proc. 10th Conf. on Entity-Relationship Approach, San Mateo, 1991

**[HAI92]** : J-L. Hainaut, syllabus du cours "Conception et Technologie des Bases de Données", Première Licence et Maîtrise en Informatique (Année Académique 1992-1993), , FUNDP

**[HAI93a]** : J-L. Hainaut, M. Chandelon, C. Tonneau, M. Joris, "Contribution to a Theory of Database Reverse Engineering", IEEE Working Conference on Reverse Engineering, Baltimore, 1993

**[HAI93b]** : J-L. Hainaut, "Schema Transformations for Database Engineering, synthesis and illustration", FUNDP, 1993

**[JOH89]** : Johannesson et Kalman, "A Method for Translating Relationnal Schemas into Conceptual Schemas", in Proc. 8th Int. Entity-Relationship conference, Toronto, 1989

**[NAV88]** : S. Navathe, A. Awong, "Abstracting relationnal and hierarchical data with a semantic data model", in Proc. 6th Int. Conf. on Entity-Relationship Approach, North Holland, 1988, pages 305-333

**[NIL85]** : E.G. Nilsson, "The Translatiion of COBOL Data Structure to an Entity-Relationship Type Conceptual Schema", in proc. of ER Approach, 1985

**[PET94]** : J-M. Petit, J. Kouloumdjian, J-F. Boulicaut, F. Toumani, "Using Queries to Improve Database Reverse Engineering", proposé pour la conférence E/R, Manchester, 1994

**[PHE93]** : PHENIX Project, "Database Reverse Engineering", second version, BIKIT / FUNDP, 1993

**[PRE93]** : W.J. Premerlani, M.R. Blaha, "An approach for reverse enginerring of relationnal databases", IEEE Working Conference on Reverse Engineering, Baltimore, 1993

**[SPA]** : Stefano Spaccapietra, Christine Parent, "Intégration de vues et relativisme sémantique", EPFL.

**[WIN90]** : J. Winans, K.H. Davis, "Software Reverse Engineering From a Currently Existing IMS Database to an Entity-Relationship Model", in Proc. of E/R approach, 1990

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Rue Grandgagnage, 21b
B-5000 NAMUR

# Reverse Engineering de Bases de Données Relationnelles SQL

**Tome 2** : Développements Techniques

*Nicolas de Ryckel*
*Philippe Soupart*

Promoteur : Jean-Luc Hainaut

Mémoire présenté en vue
de l'obtention du grade de
Licencié et Maître en Informatique

Année Académique 1993-1994

# TABLE DES MATIERES

**CHAPITRE 1. ETAT DE L'ART** .......... **1**

1.1 COMPARAISON SUR LES MODELES CIBLES .......... 1
1.2 COMPARAISON SUR LES HYPOTHESES DE DEPART .......... 2
1.3 COMPARAISON SUR LES METHODES .......... 3
1.4 COMPARAISON SUR LES RESULTATS .......... 4

**CHAPITRE 2. LES UNITES DE COMPORTEMENT** .......... **6**

2.1 DEFINITION .......... 6
*2.1.1 Syntaxe des UC* .......... *7*
*2.1.2 Sémantique des UC* .......... *7*
2.2 UTILISATION DES UC ET DE LEURS IMPLEMENTATIONS DANS LES RECHERCHES .......... 8

**CHAPITRE 3. EXTRACTION DES STRUCTURES DE DONNEES** .......... **10**

3.1 LES TYPES D'ENTITES ET LEURS ATTRIBUTS .......... 10
3.2 LES IDENTIFIANTS .......... 11
*3.2.1 Recherche dans la déclaration du schéma.* .......... *11*
3.2.1.1 Recherche dans les déclarations d'index .......... 12
3.2.1.2 Recherche dans les déclarations de tables .......... 12
*3.2.2 Recherche sur les noms* .......... *12*
*3.2.3 Recherche dans les vues* .......... *12*
*3.2.4 Recherche dans les comportements imposés aux données* .......... *13*
3.2.4.1 Définition des UC .......... 13
3.2.4.2 Recherche dans les triggers .......... 13
3.2.4.3 Recherche dans les requêtes .......... 15
*3.2.5 Recherche dans les consultations* .......... *16*
*3.2.6 Recherche dans l'extension de la base de données* .......... *16*
3.3 LES ATTRIBUTS FACULTATIFS .......... 18
3.4 LES DEPENDANCES FONCTIONNELLES .......... 20
*3.4.1 Définition* .......... *20*
*3.4.2 Recherche dans les vues* .......... *22*
*3.4.3 Recherche dans les comportements imposés aux données* .......... *22*
3.4.3.1 Définition des UC .......... 22
3.4.3.2 Recherche dans les triggers .......... 23
3.4.3.3 Recherche dans les requêtes .......... 24
*3.4.4 Recherche dans les consultations* .......... *24*
*3.4.5 Recherche dans l'extension.* .......... *25*
3.5 LES CONTRAINTES REFERENTIELLES .......... 27
*3.5.1 Recherche dans la déclaration du schéma* .......... *28*
*3.5.2 Recherche sur les noms* .......... *28*
*3.5.3 Recherche dans les vues* .......... *29*
*3.5.4 Recherche dans les checks* .......... *29*
*3.5.5 Recherche dans les comportements imposés aux données* .......... *29*

3.5.5.1 Contraintes d'inclusion ..... 29
3.5.5.1.1 Définition des UC ..... 29
3.5.5.1.2 Recherche dans les triggers ..... 31
3.5.5.1.3 Recherche dans les requêtes ..... 34
3.5.5.2 Contraintes d'égalité ..... 36
*3.5.6 Recherche dans l'extension de la base de données ..... 37*
*3.5.7 Recherche dans les consultations ..... 39*
3.5.7.1 Recherche basée sur les jointures ..... 41
3.5.7.2 Extension de la méthode. ..... 41
**CHAPITRE 4. CONCEPTUALISATION DES STRUCTURES DE DONNEES ..... 44**
4.1 NETTOYAGE ET RENOMMAGE DU SCHEMA ..... 44
4.2 DE-TRADUCTION DU SCHEMA CONFORME ..... 44
*4.2.1 Les types d'associations one-to-many ..... 44*
4.2.1.1 Transformation classique ..... 44
4.2.1.1.1 Principe ..... 44
4.2.1.1.2 Recherche ..... 45
4.2.1.2 Transformation non-classique ..... 47
4.2.1.2.1 Principe ..... 47
4.2.1.2.2 Recherche ..... 53
*4.2.2 Les identifiants dont l'un des composants est un rôle. ..... 53*
*4.2.3 Les types d'entités manquants ..... 55*
4.3 DE-OPTIMISATION D'UN SCHEMA RELATIONNEL ..... 62
*4.3.1 La redondance de dénormalisation ..... 62*
4.3.1.1 Définition ..... 62
4.3.1.2 Normalisation en 3NF ..... 63
4.3.1.2.1 Définitions ..... 63
4.3.1.2.2 Théorème de décomposition ..... 64
4.3.1.2.3 Algorithme classique de normalisation en 3NF ..... 65
4.3.1.2.4 Révision de l'algorithme classique ..... 66
*4.3.2 La redondance structurelle ..... 67*
4.3.2.1 Les attributs dérivables ..... 67
4.3.2.1.1 Cas 1 : redondance entre un attribut et la somme sur un attribut ..... 67
4.3.2.1.2 Cas 2 : redondance rôle / attribut ..... 74
4.3.2.1.3 Cas 3 : redondance entre un attribut d'une table et un attribut d'une autre table ..... 76
4.3.2.1.4 Cas 4 : redondance entre attributs d'une même table ..... 80
4.3.2.1.5 Conclusion ..... 81
4.3.2.2 La redondance entre types d'associations ..... 82
4.3.2.2.1 Recherche basée sur les connectivités ..... 83
4.3.2.2.2 Recherche dans les comportements imposés aux données. ..... 84
4.3.2.2.3 Généralisation ..... 90
*4.3.3 Les transformations de restructuration ..... 90*
4.4 EXPRESSION DU SCHEMA DANS UN MODELE DE PLUS HAUT NIVEAU ..... 91
*4.4.1 Les types d'associations complexes ..... 91*
4.4.1.1 Les types d'associations many-to-many ..... 91
4.4.1.1.1 Transformation classique ..... 91
4.4.1.1.2 Transformation non-classique ..... 92
4.4.1.2 Les types d'associations contenant un attribut ..... 96
4.4.1.2.1 Transformation classique ..... 96
4.4.1.2.2 Transformation non-classique ..... 96
4.4.1.3 Les types d'associations de degré supérieur à 2 ..... 97

4.4.1.3.1 Principe ..... 97
4.4.1.3.2 Recherche ..... 99
*4.4.2 Les attributs particuliers ..... 99*
4.4.2.1 Les attributs facultatifs ..... 99
4.4.2.1.1 Représentation par instance ..... 99
4.4.2.1.2 Représentation par valeur ..... 100
4.4.2.2 Les attributs décomposables ..... 100
4.4.2.2.1 Transformations par représentation par des attributs simples ..... 100
4.4.2.2.2 Transformations par représentation par des types d'entités ..... 102
4.4.2.3 Les attributs multivalués ..... 103
4.4.2.3.1 Représentation des instances ..... 103
4.4.2.3.2 Représentation des valeurs ..... 104
4.4.2.3.3 Augmentation de l'identifiant ..... 105
4.4.2.3.4 Instanciation ..... 105
4.4.2.3.5 Concaténation ..... 106
*4.4.3 Les contraintes d'intégrité sur les rôles et les types d'associations ..... 108*
4.4.3.1 Les contraintes sur les rôles ..... 108
4.4.3.1.1 Contraintes d'inclusion ..... 108
4.4.3.1.2 Contraintes d'égalité ..... 116
4.4.3.1.3 Contraintes d'exclusion ..... 116
4.4.3.1.4 Contraintes d'existence ..... 120
4.4.3.2 Les contraintes sur les types d'associations ..... 123
4.4.3.2.1 Les contraintes sur les types d'associations one-to-many ..... 123
4.4.3.2.2 Les contraintes sur les types d'associations many-to-many ..... 125
*4.4.4 Les relations IS-A ..... 126*

**CHAPITRE 5. PROPOSITION D'UN OUTIL POUR L'ANALYSE DE TEXTES ..... 127**

5.1 ATELIER ..... 127
5.2 LANGAGE DE DEFINITION DE PATTERNS ..... 128
5.3 DEFINITION DE SCRIPTS ..... 132

**REFERENCES ..... 136**

**ANNEXE 1. LE MODELE RELATIONNEL ..... A1**

1.1 LES CONCEPTS DE BASE ..... A1
*1.1.1 Les domaines simples ..... A1*
*1.1.2 Les relations ..... A1*
*1.1.3 Le concept d'attribut ..... A2*
1.2 LES CONTRAINTES D'INTEGRITE ..... A3
*1.2.1 Les dépendances ..... A3*
1.2.1.1 Identifiant d'une relation ..... A3
1.2.1.2 Dépendances fonctionnelles ..... A3
1.2.1.3 Dépendances multivaluées ..... A4
*1.2.2 Les contraintes d'inclusion ..... A4*
1.3 L'ALGEBRE RELATIONNELLE ..... A5
1.4 REMARQUE ..... A6

**ANNEXE 2. LA REQUETE DE SELECTION SQL ..... A7**

2.1 PRINCIPE ..... A7
2.2 EXTRACTION ET SELECTION SANS SOUS-REQUETES ..... A7
2.3 EXTRACTION DE DONNEES AGREGEES ..... A8

2.4 LES SOUS REQUETES ..... A9
2.5 EXTRACTION DE DONNEES GROUPEES ..... A9
2.6 ORDRE DU RESULTAT ..... A9

**ANNEXE 3. LE MODELE ENTITE/ASSOCIATION ..... A10**

3.1 ENTITES ET TYPES D'ENTITES ..... A10
3.2 SOUS-TYPES ET SUR-TYPES ..... A10
3.3 ASSOCIATIONS ET TYPES D'ASSOCIATIONS ..... A12
3.4 VALEURS D'ATTRIBUTS, ATTRIBUTS ET DOMAINES ..... A13
3.5 IDENTIFIANTS DES TYPES D'ENTITES ..... A14
3.6 IDENTIFIANTS DES TYPES D'ASSOCIATIONS ..... A15
3.7 CONTRAINTES D'INTEGRITE SUPPLEMENTAIRES ..... A15
*3.7.1 Contrainte d'inclusion entre rôles ..... A16*
*3.7.2 Contrainte d'égalité entre rôles ..... A16*
*3.7.3 Contrainte d'exclusion entre rôles. ..... A17*
*3.7.4 Contrainte d'existence entre rôles ..... A17*
*3.7.5 Les contraintes sur les types d'associations ..... A18*
*3.7.6 Dépendances fonctionnelles ..... A18*

**ANNEXE 4. REDONDANCE ENTRE TYPES D'ASSOCIATIONS MANY-TO-MANY ..... A19**

4.1 DEFINITION ..... A19
4.2 RECHERCHE BASEE SUR LES CONNECTIVITES ..... A19
4.3 RECHERCHE DANS LES COMPORTEMENTS IMPOSES AUX DONNEES ..... A21

# CHAPITRE 1

# ETAT DE L'ART

Nous allons comparer quelques méthodes de la littérature qui portent sur la dérivation d'un schéma conceptuel à partir d'un schéma relationnel. Nous les comparerons aussi avec la méthode sur laquelle nous nous basons [HAI93a] et les recherches que nous proposons. La comparaison portera sur les modèles cibles, sur les hypothèses de départ, sur les méthodes en elles-mêmes et sur les résultats obtenus. Les 6 méthodes que nous nous proposons de comparer sont les suivantes :

1) méthode de Davis et Arora [DAV88] ;
2) méthode de Navathe et Awong [NAV88] ;
3) méthode de Johanesson et Kalman [JOH89] ;
4) méthode de Fonkam et Gray [FON92] ;
5) méthode de Prémerlani et Blaha [PRE93] ;
6) méthode de Castellanos [CAS93].

## 1.1 COMPARAISON SUR LES MODELES CIBLES

Les 4 premières méthodes utilisent comme modèle cible un modèle dérivé du modèle E/A de Chen [CHE76] . On peut faire quelques remarques sur les principales caractéristiques et différences entre les modèles de ces 4 méthodes.

- Les méthodes 1, 2 et 4 utilisent le concept de type d'entités faible. Leurs interprétations des T.E. faibles divergent quelque peu. Ainsi, dans les méthodes 1 et 4, un T.E. faible est un T.E. dont l'existence dépend de l'existence d'un autre T.E., tandis que dans la méthode 2, un T.E. entité faible est un T.E. ayant un identifiant hybride.
- Les méthodes 2, 3 et 4 utilisent le concept de relation de spécialisation/généralisation sans toutefois préciser le type de structure (disjonction et couvrance).
- Les 4 méthodes utilisent des modèles dans lesquels la possibilité d'exprimer des connectivités minimales n'est pas a priori prévue. Le concept de type d'entités faible permet de gérer en partie les connectivités minimales 1 (mais cela n'offre pas la souplesse d'une expression directe de ces connectivités).

- Dans les modèles des 4 méthodes, les attributs ne peuvent être ni décomposables, ni répétitifs. On aura donc un schéma conceptuel moins clair, dans lequel on pourra avoir des objets du monde réel représentés par plusieurs entités.

La méthode 5 utilise le modèle orienté objet OMT, qui est plus riche que le modèle E/A étendu que nous utilisons (citons notamment les agrégations, et le fait qu'une association puisse jouer un rôle dans une association). Cependant, les contraintes d'intégrité sur les rôles et les types d'associations n'y sont pas définies explicitement.

La méthode 6 utilise le modèle orienté objets BLOOM. On y trouve notamment les concepts de classes (qui correspondent à des types d'entités), de types d'associations one-to-many (sans attributs), et d'agrégation.

Le modèle que nous utilisons comprend les connectivités minimales, les identifiants hybrides, les attributs répétitifs et les attributs décomposables. De plus, il y a la possibilité d'exprimer des contraintes d'intégrité sur des rôles ou des types d'associations. On peut aussi exprimer le type de structure d'une relation de spécialisation/généralisation.

## 1.2 COMPARAISON SUR LES HYPOTHESES DE DEPART

Les 4 premières méthodes font diverses hypothèses. Ainsi, les méthodes 1 et 2 considèrent qu'il y a une **équivalence des noms**, la méthode 3 considère que les **contraintes d'inclusion** sont connues. Les **identifiants primaires** sont supposés connus dans la méthode 1, tandis que les méthodes 2, 3 et 4 considèrent qu'elles disposent des identifiants primaires et candidats. Il s'agit d'hypothèses assez fortes, qui dans la réalité sont difficilement réalisables (il ne serait pas facile pour un utilisateur de trouver toutes ces informations). Les méthodes 5 et 6 ainsi que la méthode sur laquelle nous nous basons ne présupposent pas la connaissance de toutes ces informations. Elles ont besoin des contraintes d'inclusion et des identifiants, mais la recherche de ces informations en fait partie intégrante.

Une autre hypothèse faite par les 4 premières méthodes est que le schéma de départ est en **troisième forme normale**. Cette hypothèse en elle-même n'est pas gênante et ne peut constituer une critique à l'encontre de ces algorithmes. En effet, il existe de nombreux algorithmes de normalisation. Cependant, les **dépendances fonctionnelles** sont nécessaires pour appliquer ces algorithmes. La méthode 5 ne dit rien en ce qui concerne la normalisation du schéma. Nous pouvons donc supposer qu'elle aboutit à un schéma non-normalisé. La méthode 6 comprend des moyens de recherche de dépendances fonctionnelles et un **algorithme de normalisation**. Il en va de même dans notre travail.

# 1.3 COMPARAISON SUR LES METHODES

La méthode 1 propose un algorithme en 5 étapes, cherchant successivement les types d'entités, les types d'entités faibles, les types d'associations many-to-many et one-to-many.

La méthode 2 se compose d'un pré-processing (transformation de certains identifiants candidats en identifiants primaires, pour ne plus prendre en compte par la suite que les identfiants primaires), d'une classification des relations et des attributs, et c'est selon ces classifications que se fait la traduction en structures conceptuelles. On retrouve successivement les types d'entités, les types d'entités faibles, les types d'associations many-to-many, les types d'associations n-aires, les types d'associations one-to-many, et les spécialisations/généralisations.

La méthode 3 se compose d'une classification des relations suivie d'une recherche des structures conceptuelles. Son originalité se situe dans le fait que la classification et la conceptualisation se basent principalement sur les contraintes d'inclusion du schéma relationnel.

La méthode 4 est composée d'un pré-processing de renommage (afin d'assurer l'équivalence des noms des attributs dans tout le schéma et d'éviter des ambiguïtés de noms), et d'une recherche en 5 étapes : spécialisations/généralisations, types d'entités, types d'entités faibles, types d'associations many-to-many, types d'associations one-to-many.

La méthode 5 est assez proche de celle sur laquelle nous nous basons. Elle cherche d'abord des informations sur le schéma relationnel (types d'entités, identifiants, clés étrangères, etc.), et ensuite elle propose des transformations permettant d'obtenir un niveau de conceptualisation plus élevé (recherche des types d'associations many-to-many, des spécialisations/généralisations, etc.). La particularité de cette méthode est qu'elle prend en compte des schémas relationnels obtenus par une méthode de conception non-classique.

La méthode 6 cherche d'abord des informations dans l'extension de la base de données (identifiants, contraintes d'inclusions, dépendances fonctionnelles, etc.), et ensuite elle conceptualise en considérant divers types de configurations (surtout en fonction des contraintes d'inclusion). La particularité de cette méthode est d'inclure une recherche des types d'entités manquants [1].

On peut dire que les 6 méthodes sont plus ou moins interactives.

Cependant, dans les 4 premières méthodes et dans la méthode 6, quand on demande des informations à l'utilisateur, il doit presque à chaque fois se baser sur le schéma relationnel de départ. Sa tâche n'est donc pas très facile, car pour donner des informations, il se base sur son interprétation du schéma, et il n'est pas facile de raisonner sur un schéma physique (par exem-

[1]Un type d'entités manquant est un type d'entités qui a été traduit dans le schéma relationnel en tant qu'attribut sans que cela ne donne lieu à une dépendance fonctionnelle supplémentaire.

ple, puisqu'il dispose d'un schéma sans associations, il doit faire l'effort de réfléchir en termes de liens relationnels, et cet effort sera d'autant plus important que le schéma est grand).

La cause de ces problèmes est le fait que ces 5 méthodes passent d'un schéma physique à un schéma conceptuel sans passer (ou presque pas) par des schémas intermédiaires. Autrement dit, ils n'utilisent pas l'approche transformationnelle[2]. La méthode sur laquelle nous nous basons utilise cette approche. Ainsi, par exemple, on obtient à un certain moment un schéma sur lequel il y a des types d'associations one-to-many avant de passer à des types d'associations complexes. En fait, on passe par des schémas intermédiaires sur lesquels on trouve des concepts de plus en plus riches. Il est donc beaucoup plus facile pour l'utilisateur de s'y retrouver.

La méthode 5 se rapproche de l'approche transformationnelle. Cependant, l'ordre des transformations ne donne pas des schémas de plus en plus riches (par exemple, on cherche les spécialisations/généralisations avant les types d'associations), ce qui peut dérouter l'utilisateur.

# 1.4 COMPARAISON SUR LES RESULTATS

### méthodes 1 à 4

On trouvera dans [CAS93] une critique complète des 4 premières méthodes. Nous allons en voir les points principaux.

- Aucune des 4 méthodes ne retrouve les types d'entités manquants. Cela aboutit à retrouver dans certains cas de mauvaises structures conceptuelles.
- Il y a des problèmes concernant les types d'entités faibles dans les méthodes 1, 2 et 4. Dans les méthodes 1 et 4, c'est, nous semble-t-il, notamment dû à une mauvaise définition du concept de T.E. faible. La méthode 2, dans laquelle un T.E. faible correspond à un T.E. ayant un identifiant hybride, ne retrouve pas les T.E. faibles reliés à des T.E. faibles.
- La méthode 1 ne permet pas de retrouver les types d'associations implémentés par référence vers un identifiant candidat.
- La méthode 2 ne détecte pas les types d'associations récursifs.
- La méthode 2 ne détecte pas l'héritage multiple.
- Dans la méthode 4, il y a une ambiguïté entre la définition des relations qui deviendront des types d'entités normaux et la définition des relations qui deviendront des types d'associations.

---

[2]Cette approche consiste à passer d'un schéma à un autre par une succession de transformations. On peut lire [HAI91] pour plus d'informations.

## méthode 5

La méthode 5 ne retrouve pas les types d'entités manquants, et n'aboutit pas à un schéma normalisé (cf ci-dessus la comparaison sur les hypothèses). De plus, elle ne dé-optimise pas complètement le schéma (notamment, elle ne recherche pas les attributs dérivables).

Un des avantages de cette méthode est qu'elle considère un schéma relationnel qui pourrait être obtenu en utilisant des méthodes de conception non-classiques.

## méthode 6

La méthode 6 ne de-optimise pas le schéma. De plus, elle se base uniquement sur l'extension et elle semble assez rigide. Cela entraîne des risques d'obtenir un schéma conceptuel ne correspondant pas tout à fait au schéma relationnel de départ. En effet, il faut que l'extension soit totalement représentative des contraintes sous-jacentes au schéma (c-à-d que l'extension ne doit pas refléter des contraintes qui en réalité ne sont pas posées sur le schéma). Si ce n'est pas le cas, la rigidité de la méthode empêche l'utilisateur de 'rectifier le tir'.

Voyons maintenant quelles critiques on peut faire par rapport à la méthode utilisée dans notre travail. Tout d'abord, les méthodes 1 à 4 et la méthode 6 ne dé-optimisent pas le schéma. Ensuite, elles ne considèrent pas des schémas obtenus de manière non-classique, ce qui peut aboutir à des schémas conceptuels qui, même s'ils sont corrects par rapport au schéma relationnel de départ, ne correspondent pas à la réalité modélisée dans le schéma relationnel. La méthode sur laquelle nous nous basons comprend une étape de dé-optimisation, et elle prend en compte tant des schémas obtenus par des méthodes classiques que par des méthodes non-classiques (mentionnons par exemple la traduction d'un type d'associations many-to-many par copie d'un ensemble de clés étrangères).

# CHAPITRE 2
# LES UNITES DE COMPORTEMENT

## 2.1 DEFINITION

Une Unité de Comportement (UC) est la description formelle d'une manipulation de données servant à gérer (du moins en partie) une contrainte d'intégrité. Pour définir les UC, nous nous sommes inspirés des Update Protocol Model (UPM) définis par Carlson dans [CAR83], que nous avons redéfinis pour les adapter à nos besoins.

Une UC définit le comportement que doivent suivre les données lors de certaines opérations sur celles-ci (ajout, suppression ou modification de données). Plus précisément, les UC spécifient quelles sont les préconditions ou les opérations à réaliser lors d'une opération sur une ligne d'une table. Les UC permettent d'exprimer les comportements imposés aux données indépendamment des moyens utilisés. Ces moyens sont les triggers et les requêtes du programme d'application. Ils peuvent prendre les formes les plus diverses. En fait, on peut considérer que les UC sont les spécifications des triggers et des configurations de requêtes qui gèrent les contraintes d'intégrité.

Avant de voir la syntaxe et la sémantique des UC, nous allons en donner un exemple.

Soit le schéma relationnel :

```
A (a1 , a2)
B (b1 , b2 , a1)

B.a1 ⊆ A.a1
```

Une des UC de gestion de la contrainte d'inclusion ci-dessus pourrait être :

```
DELETE a FROM A
ou bien PRECOND ¬∃ b ∈ B | b[a1] = a[a1]
ou bien IMPLIES REPLACE b | b[a1] = a[a1] BY b' IN B | b'[a1] = NULL
```

Lors d'une suppression d'une ligne a de la table A, soit on vérifie qu'il n'existe pas de ligne de B ayant les mêmes valeurs pour son attribut a1 que a[a1], soit on va mettre une valeur nulle à l'attribut a1 des lignes de B pour lesquelles a1 = a[a1].

## 2.1.1 Syntaxe des UC

Nous allons donner la syntaxe des UC en utilisant une la notation proche de la notation BNF. Le symbole de disjonction est représenté par le ' / '.

```
UC ::= condition_de_declenchement
       corps

condition_de_declenchement ::=
      {INSERT nom_ligne IN nom_table [|predicat]
      / DELETE nom_ligne FROM nom_table [ | predicat]
      / REPLACE nom_ligne BY nom_ligne' [ | predicat]}


corps ::= { precondition / implication / OU BIEN precondition OU BIEN
             implication }

precondition ::= PRECOND predicat

implication ::= IMPLIES action [OU BIEN implication]

action ::= operation [action]

operation ::= { INSERT nom_ligne IN nom_table |  predicat
              / DELETE nom_ligne FROM nom_table | predicat
              / REPLACE [ALL | ANY | ONE] nom_ligne [ | predicat] BY
                nom_ligne' IN nom_table | predicat }
```

Remarque

Lorsqu'une opération ne donne lieu à aucune gestion de contrainte d'intégrité, on indique la mention suivante :

```
NO PROTOCOL
```

## 2.1.2 Sémantique des UC

Nous allons maintenant voir le sens d'une Unité de Comportement. Nous ne reprendrons pas tous les éléments de la syntaxe, car la définition de certains d'entre eux est triviale.

La **condition de déclenchement** est une opération (ajout, suppression, modification) sur une ligne d'une table qui va déclencher la vérification de la **précondition** ou l'**implication**. Dans certains cas, on étend la condition de déclenchement en ajoutant un **prédicat** (le symbole '|' présent devant le prédicat signifie "tel que").

Le **corps** de l'UC est constitué d'une précondition ou d'une implication, ou encore des deux. Dans ce dernier cas, on trouvera l'un ou l'autre dans l'implémentation de l'UC. On note ce fait par des '**ou bien**'.

La précondition (**PRECOND**)est constituée de **prédicats** qui doivent être vérifiés pour que l'opération de déclenchement de l'UC puisse être réalisée.

L'implication (**IMPLIES**) consiste à effectuer des **opérations** sur des tables en fonction des lignes de la condition de déclenchement. Une opération peut-être l'ajout (**INSERT**), la suppression (**DELETE**) ou la modification (**REPLACE**) d'une (ou de plusieurs) ligne(s). Le prédicat de l'INSERT décrit la ligne insérée. Il en va de même pour le prédicat du DELETE. Dans le REPLACE, on trouve deux prédicats. Le premier sert à définir la (les) ligne(s) que l'on modifie, et le deuxième décrit l'état de la (des) ligne(s) après la modification[4]. Les **ALL**, **ANY**, **ONE** servent à définir le nombre de lignes qui seront modifiées. ALL signifie toutes celles qui vérifient le premier prédicat , ANY signifie au moins une d'entre elles, et ONE signifie une et une seule.

Nous allons maintenant évoquer ce qui est implicite dans les UC. Lorsque, dans le REPLACE de l'implication, la première condition ne permet de trouver qu'une seule ligne, on ne mettra ni ALL, ni ANY, ni ONE. En ce qui concerne l'INSERT de l'implication, les valeurs qui ne sont pas mentionnées sont considérées comme n'ayant aucune importance pour l'UC. Notons que dans l'implémentation de l'UC, il faudra prendre en compte ces valeurs, mais elles n'auront pas d'influence sur la gestion de la contrainte d'intégrité. Pour le REPLACE de l'implication, c'est le même principe : on considère que les valeurs non mentionnées ne sont pas modifiées, ou si elles le sont, leur nouvelle valeur n'influence pas la gestion de la contrainte d'intégrité. C'est le même principe pour les valeurs non mentionnées dans la condition de déclenchement : elles n'ont pas d'influence sur le déclenchement de l'UC.

## 2.2 UTILISATION DES UC ET DE LEURS IMPLEMENTATIONS DANS LES RECHERCHES

La recherche dans les comportements imposés aux données est généralement développée en détail dans notre travail. Nous définissons d'abord les UC, et ensuite nous envisageons une de leurs implémentations possibles sous forme de triggers Sybase (version 4.0) et de configurations de requêtes SQL standard.

On peut faire trois remarques à ce propos :

- Les UC que nous définissons décrivent les comportements qui sont les plus susceptibles d'être imposés aux données pour gérer les contraintes.

---

[4]Les attributs inchangés ne sont pas décrits.

- Les triggers que nous développons ne sont valables que si on prend comme hypothèse que les mises à jour effectuées sur les tables de déclenchement sont toujours effectuées sur une seule ligne. En effet, les triggers Sybase ne sont pas assez puissants que pour prendre en compte des opérations complexes sur plusieurs lignes (dans d'autres SGBD comme par exemple RDB, on a la possibilité de déclarer la clause `FOR EACH ROWS` qui stipule que l'action du trigger doit être exécutée pour chaque ligne faisant l'objet de l'opération de déclenchement) . Nous avons choisi les triggers Sybase en raison de leur simplicité et de leur souplesse.

- Les triggers et les requêtes sont donnés à titre indicatif (nous ne les développons d'ailleurs pas à chaque fois). Rappelons en effet qu'ils peuvent être réalisés de diverses manières. Les UC étant d'un niveau d'abstraction plus élevé, on peut leur accorder une plus grande importance.

# CHAPITRE 3

# EXTRACTION DES STRUCTURES DE DONNEES

## 3.1 LES TYPES D'ENTITES ET LEURS ATTRIBUTS

Dans cette phase on va partir du schéma relationnel et obtenir un premier schéma E/A ne décrivant que la structure des tables et des attributs. Pour ce faire il suffit de parcourir le texte DDL de déclaration de la BD et créer pour chaque opération `CREATE TABLE` une entité qui a le même nom et les mêmes attributs (avec leurs types) que la table ainsi déclarée.

La figure 3.1. montre la traduction d'une table `CLIENT` en type d'entités.

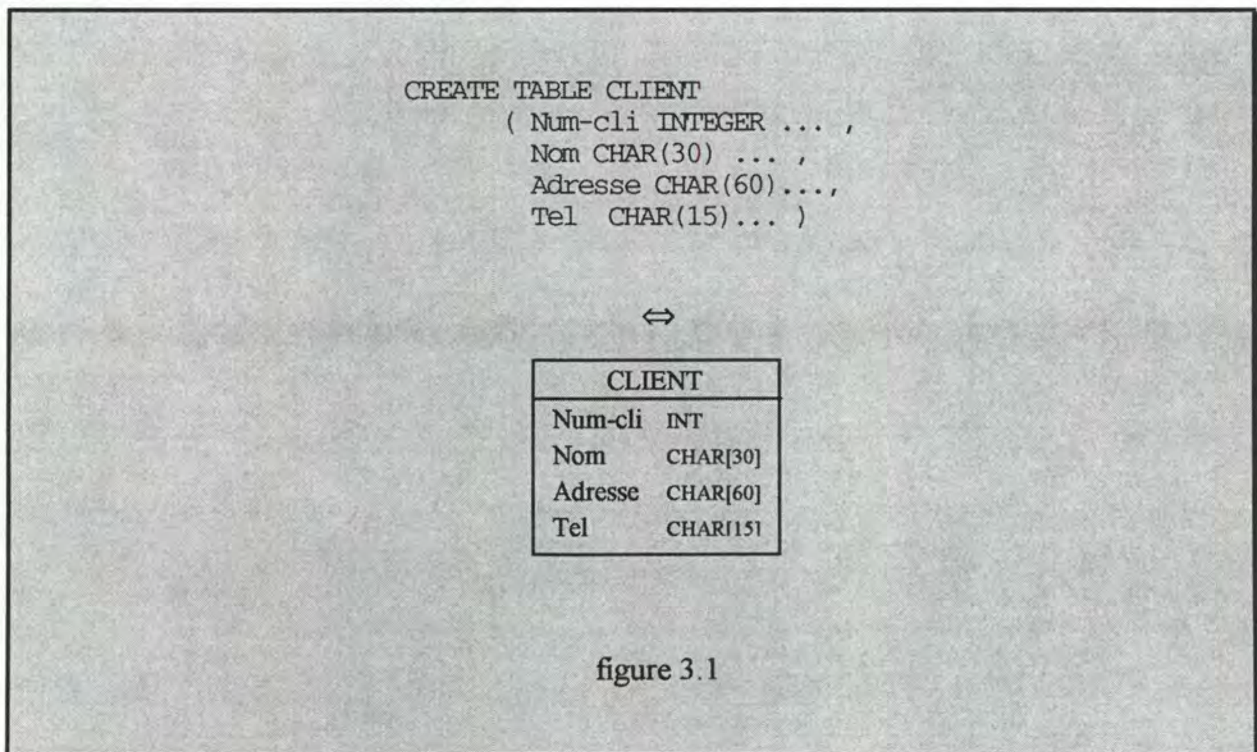



figure 3.1

Dans la suite du travail, nous garderons parfois le graphisme E/A tout en gardant la terminologie du modèle relationnel.

# 3.2 LES IDENTIFIANTS

Nous allons rechercher ici les moyens de retrouver l'(les) identifiant(s)[5] d'une table en se basant sur la déclaration des tables, sur les vues et sur les comportements imposés aux données. Nous considérerons que l'on analyse chaque table seule, sans regarder ses liens avec les autres tables. Nous verrons dans la partie consacrée à la recherche des contraintes référentielles qu'on peut encore retrouver quelques indices sur les identifiants lors de cette phase.

La table dont nous nous servirons pour illustrer ce qui suit est représentée à la figure 3.2. Sur cette figure, a1 est l'identifiant. Notons que a1 pourrait être constitué de plusieurs attributs. On aurait alors un identifiant composé de plusieurs attributs. a2 peut aussi être constitué de plusieurs attributs. Ce qui suit sera aussi valable dans le cas d'une table ayant plusieurs identifiants.

Dans la recherche des identifiants, nous considérerons aussi les identifants *facultatifs*. Un identifiant facultatif est un identifiant dont les valeurs non-nulles[6] sont distinctes.

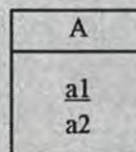



figure 3.2

## *3.2.1 Recherche dans la déclaration du schéma.*

Cette partie est inspirée de [BEL93].

---

[5]Rappelons que les identifiants intéressants sont les identifiants stricts.

[6]Une valeur nulle est une valeur particulière d'attribut employée en relationel pour simuler la présence d'un attribut absent ou dont on ne connaît pas la valeur. En SQL, ce sera le plus souvent 'NULL' qui sera utilisé comme valeur nulle mais on peut aussi se servir d'une valeur conventionnelle.

#### 3.2.1.1 Recherche dans les déclarations d'index.

Une recherche dans les déclarations des index est un moyen simple et sûr de retrouver les identifiants d'une table. En effet, il suffit de chercher des déclarations de la forme :

```
CREATE UNIQUE INDEX nom-index ON A (a1)
```

Le fait de déclarer l'index unique permet de s'assurer qu'il n'y aura jamais deux lignes de la table A qui ont la même valeur pour les attribut de cet index. Ces attributs constituent donc un identifiant.

Remarque

Un identifiant est souvent indexé. Les index "non-uniques" fournissent donc aussi un indice (moins fiable) pour la recherche des identifiants.

#### 3.2.1.2 Recherche dans les déclarations de tables

Il suffit de se baser sur les clauses `UNIQUE` et `PRIMARY KEY`. Celles-ci peuvent se trouver dans la déclaration des colonnes ou dans la déclaration de la table (dans les blocs `PRIMARY KEY` ou `UNIQUE`). Les colonnes auxquelles on a donné une de ces deux propriétés représenteront donc des attributs identifiants. Ici aussi, comme dans le cas des index "uniques", le niveau de certitude est maximal.

### *3.2.2 Recherche sur les noms*

Cette recherche se base sur une analyse des noms des attributs d'une table. Elle n'a pas un niveau de certitude aussi élevé que la précédente.

On constate dans beaucoup de cas que l'attribut identifiant porte le même nom que la table, ou une abréviation, ou encore qu'il est préfixé ou postfixé par "id", "num", "key", etc. Cela permet donc de retrouver aisément un certain nombre d'identifiants qui n'ont pas été déclarés explicitement.

### *3.2.3 Recherche dans les vues*

Supposons qu'on ait déclaré la vue suivante :

```
CREATE VIEW nom-vue (..., att-1,...)
      AS SELECT ...
              CONCAT(att-2, att-3)
              ...
         FROM R1
```

Cette vue permet d'accéder aux attributs att-2 et att-3 en une seule fois. Ce genre de construction peut-être utilisé lorsque (att-2, att-3) forment un identifiant pour faciliter la dénomination de celui-ci. Lors d'une jointure par exemple, il ne faudra utiliser qu'un seul attribut (att-1) pour désigner l'identifiant de la table.

## *3.2.4 Recherche dans les comportements imposés aux données*

### 3.2.4.1 Définition des UC

*A) Ajout d'une ligne dans A*

```
INSERT a IN A
ou bien PRECOND ¬ ∃ a' ∈ A | a'[a1] = a[a1]
ou bien IMPLIES REPLACE a BY a' IN A | ∀ a'' ∈ A : a'[a1] <> a''[a1]
```

Lorsqu'on ajoute une ligne dans A, soit on vérifie qu'il n'existe pas d'autre ligne ayant la même valeur pour l'identifiant, soit on affecte une valeur correcte à l'identifiant de la ligne insérée.

*B) Suppression d'une ligne de A*

```
NO PROTOCOL
```

*C) Modification d'une ligne de A*

```
REPLACE a BY a' IN A | a[a1] <> a'[a1]
PRECOND ¬∃ a'' ∈ A | a''.a1 = a'.a1
```

Remarquons que ces UC sont aussi valables pour un identifiant facultatif, excepté le fait qu'il ne faudra prendre en compte que les valeurs non-nulles lors des comparaisons.

### 3.2.4.2 Recherche dans les triggers

*A) Ajout d'une ligne dans A*

La précondition pourrait être implémentée par un trigger du genre :

```
CREATE TRIGGER nom-trigger on A
FOR INSERT
AS IF (SELECT COUNT(*) FROM A, inserted WHERE A.a1 = inserted.a1)
     <> 1

BEGIN
   gestion-erreur
END
```

Ce trigger vérifie que le nombre de lignes qui ont un identifiant égal à l'identifiant inséré est bien égal à 1 (rappelons que les triggers Sybase sont évalués après l'insertion).

L'`IMPLIES` pourrait correspondre à un trigger qui affecte lui même la valeur d'un identifiant inséré.

```
CREATE TRIGGER nom-trigger ON A
FOR INSERT
AS

BEGIN

        UPDATE A SET a1 = SELECT max(a1)+ 1
                          FROM A

        FROM inserted
        WHERE A.a1 = inserted.a1
END
```

Ce trigger ne pourra être présent que dans le cas où l'identifiant est un numéro, et où il n'a pas de "sens sémantique", c-à-d que sa valeur peut-être déterminée automatiquement (il peut s'agir par exemple du numéro de dossier d'un client). Dans ce trigger, on voit que la valeur de l'identifiant de l'élément nouvellement inséré aura la valeur maximale de tous les identifiants présents dans `A` incrémentée de 1.

Remarque

On peut se demander quelle était la valeur de l'attribut `a1` de l'élément que l'on a inséré dans `A` avant le déclenchement du trigger. En effet, un tel trigger pourrait être implémenté sur un schéma dans lequel on a défini `A.a1` comme étant un index unique. Il faut donc être prudent et ne pas se dire que l'on peut insérer n'importe quoi dans `A.a1`, puisque de toutes façons le trigger lui donne la bonne valeur. Une solution serait par exemple de mettre `A.a1` à 0 quand on insère une ligne dans `A`, tout en s'assurant que la numérotation des identifiants commence à 1 et pas à 0.

*B) Modification d'une ligne de A*

On pourrait trouver un trigger qui effectue le même genre de vérification que lors de l'ajout d'une ligne dans A.

```
CREATE TRIGGER nom-trigger on A
FOR UPDATE
AS IF UPDATE (a1) AND
   (SELECT COUNT(*) FROM A, inserted WHERE A.a1 =  inserted.a1)
    <> 1

BEGIN
    gestion-erreur
END
```

Remarque

Le `IF UPDATE (a1)` sert à vérifier qu'il y a bien eu une modification sur `A.a1`.

### 3.2.4.3 Recherche dans les requêtes

Voici maintenant comment on peut s'assurer de l'unicité d'un identifiant par des constructions de requêtes dans les programmes d'application.

*A) Ajout d'une ligne dans A*

Soit l'ajout de la ligne `(var-a1 , var-a2)` dans A.

La précondition pourrait être implémentée comme suit. Avant l'ajout, on vérifiera que `var-a1` n'est pas déjà présent dans A. On aura la requête :

```
EXEC SQL SELECT a1 FROM A WHERE a1 = :var-a1 END-EXEC
```

On peut ensuite analyser le SQLCODE et si on trouve 100, c'est que `var-a1` n'était pas présente dans A, et on peut faire l'insertion. Dans le cas contraire, il y aura une gestion d'erreur. On aura donc la structure suivante :

```
IF trouve-100
THEN EXEC SQL INSERT INTO A VALUES (:var-a1 , :var-a2) END-EXEC
ELSE gestion-erreur
```

En ce qui concerne l'`IMPLIES`, nous allons voir comment on peut, au sein du programme d'application, affecter automatiquement une valeur à l'identifiant .

Il y a deux manières de procéder. Soit on fait comme pour le trigger d'action, c-à-d d'abord l'insertion et puis la mise à jour de l'élément inséré, soit on cherche d'abord la bonne valeur de `A.a1` et ensuite on insère. Nous allons développer cette deuxième solution.

On va donc d'abord chercher la valeur maximale parmi les valeurs de l'identifiant de A.

```
EXEC SQL SELECT max (a1) INTO :var-a1 FROM A END-EXEC
```

Ensuite, on incrémente cette valeur de 1.

```
var-a1 = var-a1 + 1
```

On peut alors faire l'insertion.

```
EXEC SQL INSERT INTO A VALUES (:var-a1 , :var-a2) END-EXEC
```

*B) Modification d'une ligne de A*

On aura la même structure que pour l'ajout. Avant l'insertion, on regarde si la nouvelle valeur de l'identifiant est présente dans A. Si elle est présente, il y aura une gestion d'erreur. Sinon, on fera le UPDATE.

### 3.2.5 Recherche dans les consultations

On peut aussi trouver des traces d'un identifiant dans les consultations de la table. En effet, si on trouve une requête du type :

```
EXEC SQL   SELECT ... FROM A WHERE A.a1 = :var-a1   END-EXEC
```

et que cette requête n'est jamais accompagnée d'un curseur, c'est qu'on est certain à chaque fois de n'obtenir qu'une seule ligne. On peut donc faire l'hypothèse que l'attribut du WHERE est identifiant. Ceci n'est bien entendu valable que s'il n'y a pas utilisation d'une fonction d'agrégation ou de la clause DISTINCT.

### 3.2.6 Recherche dans l'extension de la base de données

Pour retrouver les identifiants d'une table, on peut aussi consulter l'extension de cette table, ou tout au moins un échantillon.

Il suffit de consulter chaque colonne, et examiner si toutes ses valeurs sont distinctes. Si tel est le cas, on a un identifiant. La même recherche doit aussi être faite sur des ensembles de colonnes, pour détecter aussi les identifiants composés de plusieurs attributs. Cette méthode

est donc assez coûteuse en temps. Un moyen de l'optimiser serait de d'abord faire la recherche sur toutes les colonnes prises séparément. Ensuite, on cherche les identifiants constitués de plus d'un attribut. Pour cela, on fait un examen sur des ensembles de colonnes. On ne considère que des ensembles ne contenant pas de colonne(s) reconnues en tant qu'identifiantes. En effet, de tels ensembles donneraient lieu à des identifiants non stricts.

Pour mener à bien cette recherche il suffit d'appliquer le type de requêtes suivantes :

```
EXEC SQL SELECT COUNT (*) FROM A INTO :var-2  END-EXEC

EXEC SQL SELECT COUNT (A.a1) FROM A INTO :var-1  END-EXEC
```

On compare alors `var-1` et `var-2`, et si ces deux variables sont égales, alors `A.a1` est probablement identifiant.

Un autre moyen serait d'utiliser le curseur suivant :

```
EXEC SQL

        DECLARE CURSOR  nom-curseur FOR SELECT COUNT (A.a1)
                                        FROM A GROUP BY A.a1

END-EXEC
```

Après avoir ouvert ce curseur, on fait une boucle sur un FETCH et on regarde si on obtient à chaque fois 1. Cette méthode sera plus rapide, puisque dès qu'on trouve une valeur différente de 1, on peut passer à un autre attribut. Dans le cas où un attribut est identifiant, on parcourera tout le résultat de la requête (mais uniquement dans ce cas là).

# 3.3 LES ATTRIBUTS FACULTATIFS

On trouve dans [BEL93] diverses façons de traduire un schéma E/A contenant des attributs facultatifs en un schéma conforme au modèle relationnel. Le modèle relationnel n'acceptant pas les attributs facultatifs, il faut soit rendre ceux-ci obligatoires, soit en faire des types d'entités. Nous allons nous intéresser ici à la manière de retrouver un attribut facultatif qui a été rendu obligatoire. Nous nous intéresserons à l'autre méthode lors de la conceptualisation.

Considérons le type d'entités de la figure 3.3 dans lequel `a3` est un attribut facultatif.

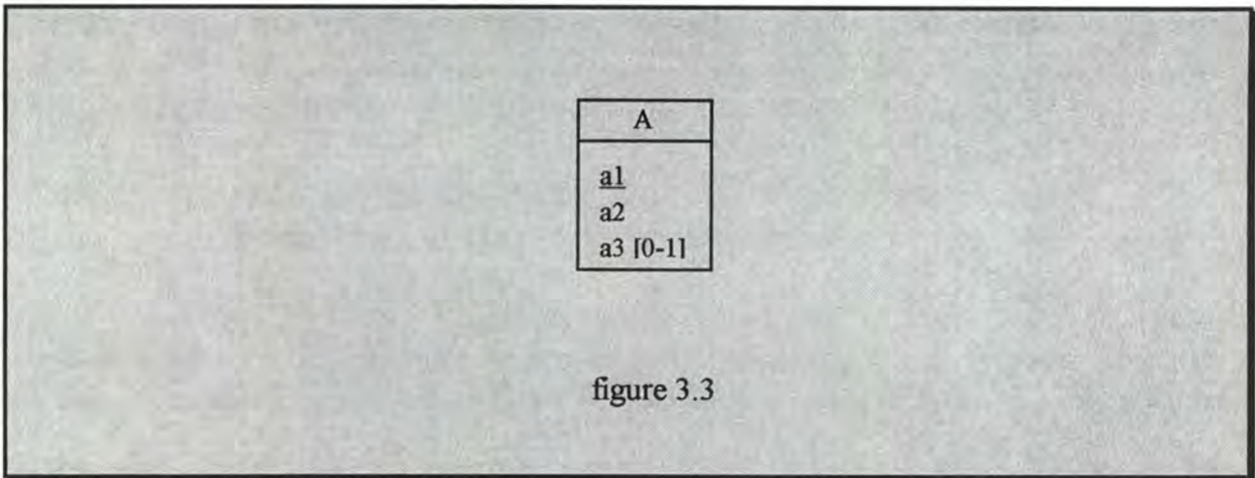



figure 3.3

En SQL, on peut simuler le caractère facultatif d'un attribut de deux façons : soit en utilisant la valeur `NULL`, soit en utilisant une valeur conventionnelle, du même domaine que les autres valeurs que peut prendre l'attribut facultatif. Ces deux valeurs n'ont pas d'autres signification que de représenter une valeur d'attribut absente.

L'utilisation de la clause `NOT NULL` lors de la déclaration des attributs fournit un indice permettant de repérer les attributs facultatifs: on peut considérer que tout attribut qui n'est pas déclaré `NOT NULL` est facultatif (les autres étant obligatoires).

Cette méthode n'est pas sûre. En effet, le concepteur a pu ne pas ajouter la clause `NOT NULL` par habitude ou parce qu'il ne connaissait pas cette déclaration ou encore parce qu'il prévoit de l'information incomplète pour cette attribut. Cependant la probabilité qu'il s'agisse bien d'un attribut facultatif augmente si la valeur `NULL` est utilisée dans les requêtes ou les triggers utilisant l'attribut en question, ou est présente dans l'extension de la base de données.

L'utilisation d'une valeur conventionnelle est plus difficile à détecter. En effet, on ne sait pas ce que sera cette valeur, puisque c'est le concepteur qui la choisit. Il peut s'agir de 0 pour le prix d'un produit, ou "*****" pour un libellé, etc...

Une façon de trouver un indice de la présence d'une valeur conventionnelle pour un attribut est d'examiner un échantillon de l'extension de la table. Si on remarque que la même valeur revient souvent, on peut dégager l'heuristique qu'il s'agit d'une valeur conventionnelle. Cette heuristique pourra éventuellement être renforcée si cette valeur est 0 pour un attribut numérique, ou si cette valeur est constituée de "X" ou de "*" dans le cas d'un alphanumérique, etc.

On pourrait aussi retrouver des traces d'une gestion de valeur conventionnelle dans les requêtes. Si on trouve plusieurs requêtes du type :

```
EXEC SQL  INSERT INTO A VALUES ( :var-a1 , :var-a2 , "*****" )  END-EXEC
```

alors on peut supposer que "*****" est une valeur conventionnelle.

Dans la suite du texte, quand nous utiliserons une valeur représentant un attribut absent (nous appellerons une telle valeur une valeur nulle), nous ne prendrons que le cas où elle a été implémentée par utilisation de `NULL`. Les autres possibilité d'implémentation (utilisation de valeurs conventionnelles) ne seront pas prises en compte, bien qu'elles soient très proches.

Etant donné le peu de certitude que l'on peut avoir à propos des attribut facultatif à ce niveau de l'analyse, l'utilisateur devra reconsidérer le caractère obligatoire ou facultatif des attributs lorsqu'il aura un schéma de niveau d'abstraction plus élevé.

# 3.4 LES DEPENDANCES FONCTIONNELLES

Les dépendances fonctionnelles (DF) fournissent, en tant que contraintes d'intégrité particulières, des informations importantes sur la sémantique d'un schéma relationnel. Nous verrons (cf 4.3.1) qu'une table a pu, lors du processus de conception, être obtenue suite à une dénormalisation. Le problème de ces dénormalisations est qu'une table peut alors représenter deux concepts du monde réel (à savoir deux entités conceptuelles). Dans ce cadre, les DF fournissent des moyens permettant de normaliser le schéma.

Le problème avec les DF est qu'aucun SGBD SQL ne fournit de moyen pour les déclarer. Pour les retrouver nous devrons donc limiter l'analyse aux manipultations des données et à l'extension de la base de données.

## *3.4.1 Définition*

Considérons le schéma de la figure 3.4.

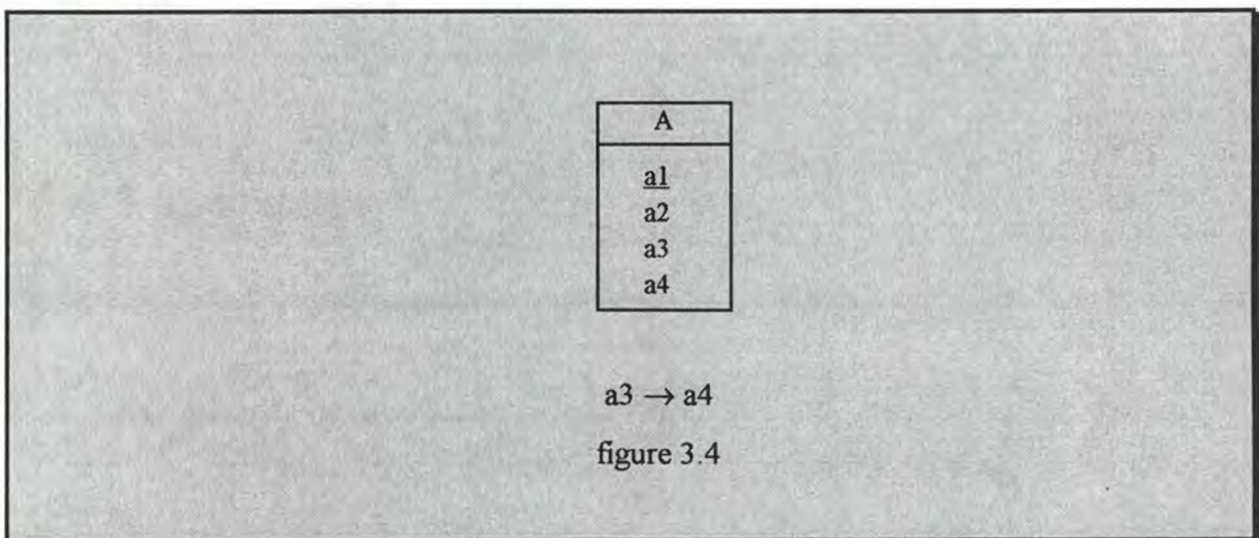


figure 3.4

Une DF (notée →) est une contrainte qui spécifie qu'une valeur d'un attribut a3 (ou d'un ensemble d'attributs a3) détermine univoquement la valeur d'un autre attribut(ou ensemble d'attributs a4). a3 est appelé le déterminant, a4 le determiné. Comme pour les identifiants nous considérons aussi les DF dont le déterminant est 'facultatif'[7].

[7]Dans ce cas on ne considère que les lignes pour lesquelles le déterminant est non-nul.

```
a3 -> a4 ⇔ ∀paire de ligne t1, t2 ∈ relation R
              t1.a3 = t2.a3 ⇒ t1.a4 = t2.a4
```

Les seules DF intéressantes sont celle appelées DF 'élémentaires'.

On peut déduire de la définition qu'il y a une DF de l'identifiant d'une table vers tous les autres attributs de la table[8].

**Définition : dépendance fonctionnelle élémentaire**

```
a3 → a4 est élémentaire ⇔ a4 est un attribut simple,
                         a3 est minimal,
                       et a3 → a4 est non triviale

a3 → a4 est triviale si a3 ⊆ a4
```

Pour faciliter la découverte des DF, il faut être particulièrement attentif à certaines dépendances qui ne doivent pas être prises en compte :

- Les DF non-élémentaires où a4 est composite car elles peuvent être dérivées de DF élémentaires grâce à la règle d'additivité :

```
a3 → a4 ,  a3' → a4' ⇒  (a3,a3') → (a4,a4')
en particulier : a3 → a4, a3 → a4' ⇔ a3 → (a4,a4')
```

- Les DF non-élémentaires où a3 n'est pas minimal car elles peuvent être dérivées par augmentation :

```
∀ a3' ⊇ a3 : a3 → a4 ⇒ a3' → a4
```

- Les DF transitives car elle peuvent être dérivées par transitivité :

```
a3 → a4  , a4 → a4' ⇒ a3 → a4'
```

[8]Les dépendances fonctionnelles pourraient donc servir à une recherche des identifiants. On trouve dans [HAI89b] une recherche des identifiants basée sur les dépendances fonctionnelles.

## 3.4.2 Recherche dans les vues

On peut obtenir un indice de la présence d'une DF si on retrouve des vues semblables aux suivantes :

```
CREATE VIEW A1 (...)
AS SELECT a1, a2 , a3
   FROM A

CREATE VIEW A2 (...)
AS SELECT DISTINCT a3, a4
   FROM A
```

Ces deux vues représentent les tables qu'ont aurait obtenu en normalisant A (cf 4.3.1). Le concepteur peut décider de faire deux vues car il considère qu'elles représentent mieux le réel perçus qu'une simple table.

Si on retrouve ces deux vues, on ne peut faire qu'une supposition, car les mêmes vues peuvent être utilisées pour d'autres raisons.

Remarque

La vue A2 est intéressante car elle permet par la suite d'utiliser l'attribut a3 comme s'il était identifiant. Certaines méthodes présentées pour la recherche des identifiant pourraient donc, si elles étaient appliquées aux vues, permettre de retrouver certains déterminants d'une DF.

## 3.4.3 Recherche dans les comportements imposés aux données

### 3.4.3.1 Définition des UC

*Ajout d'une ligne dans A*

```
INSERT a IN A
ou bien PRECOND ¬∃ a' ∈ A | a'[a3] = a[a3] AND a'[a4] <> a[a4]
ou bien IMPLIES REPLACE a | (∃ a'' ∈ A | a''[a3] = a[a3])
        BY a' IN A | a'[a4] = a''[a4]
```

La précondition correspond à la vérification avant l'ajout d'un élément dans A qu'il n'existe pas de lignes de A qui ont une même valeur pour a3 et une valeur différente pour a4. L'IMPLIES modifie a.a4 en lui affectant la valeur de l'attribut a4 d'une ligne qui a le même a3 (que a.a3).

*Suppression d'une ligne de A*

```
NO PROTOCOL
```

*Modification d'une ligne de A*

```
REPLACE a BY a' IN A
ou bien PRECOND ¬ ∃ a'' ∈ A | a''[a3] = a'[a3] AND a''[a4] <> a[a4]
ou bien IMPLIES REPLACE a' | (∃ a''' ∈ A | a'''[a3] = a'[a3]) BY a'' IN
A | a''[a4] = a'''[a4]
```

Cette UC est quasiment identique à l'UC de l'ajout.

### 3.4.3.2 Recherche dans les triggers

Examinons les triggers concernant l'ajout dans A.

La précondition correspond à un trigger de vérification de la forme :

```
CREATE TRIGGER nom-trigger ON A
FOR INSERT
AS IF EXISTS (SELECT *
              FROM A, inserted
              WHERE A.a3 = inserted.a3 AND A.a4 <> inserted.a4)

BEGIN
      gestion-erreur
END
```

Si on trouve une ligne de `A` qui a la même valeur pour `a3` et une valeur différente pour `a4` que la ligne insérée, alors il y a une erreur : la dépendance fonctionnelle n'est plus respectée.

Le trigger de vérification concernant la modification a la même forme.

L'`IMPLIES` correspond à un trigger d'action du genre

```
CREATE TRIGGER nom-trigger ON A
FOR INSERT AS
BEGIN
      UPDATE A
      SET A.a4 = SELECT DISTINCT a4 FROM A WHERE A.a3 = inserted.a3
      FROM inserted
      WHERE A.a3 = inserted.a3
END
```

L'implémentation de ce trigger suppose qu'il y a déjà dans A une ligne qui a une valeur de a3 égale à la valeur de a3 de la ligne insérée.

### 3.4.3.3 Recherche dans les requêtes

Examinons les requêtes concernant l'ajout dans A.

Supposons que l'on ajoute dans A la ligne (var-a1 , var-a2 , var-a3 , var-a4).

Pour la précondition de l'UC, on pourrait avoir la requête suivante :

```
EXEC SQL

        SELECT DISTINCT a4 FROM A INTO :var-a4-bis
        WHERE A.a3 = :var-a3

END-EXEC
```

Si cette requête ne donne pas de résultat, c'est qu'il n'y a pas de ligne de A qui a la même valeur de a3, et on peut insérer sans problème.

Si cette requête donne un résultat, on compare var-a4 et var-a4-bis, et si ces deux variables sont égales, on peut faire l'insertion. Dans le cas contraire, on ne fera pas l'insertion, et il y aura probablement un traitement d'erreur.

L'IMPLIES pourra se traduire de deux façons. Ou bien on insère d'abord dans A, avec une valeur nulle pour a4, et ensuite il y a une requête pour aller voir quelle sera la valeur de a4 en fonction de la valeur pour a3 de la ligne insérée. Ou bien, avant l'insertion, on va chercher la valeur de a4 et on insère ensuite. La première solution correspond strictement à l'implies, et c'est ainsi qu'à été implémenté le trigger d'action. La solution qui a le plus de chances d'être implémentée nous semble être la deuxième. On aurait alors les requêtes suivantes.

```
EXEC SQL
        SELECT DISTINCT a4 INTO :var-a4 FROM A WHERE a3 = :var-a3
END-EXEC

EXEC SQL
        INSERT INTO A VALUES (:var-a1, :var-a2, :var-a3, :var-a4)
END-EXEC
```

Les requêtes correspondantes à l'UC de modification seront très proches de celles concernant l'ajout.

### 3.4.4 Recherche dans les consultations

Nous allons voir ici de quelle façon on peut retrouver des indices de la présence d'une dépendance fonctionnelle en s'intéressant aux consultations de la BD.

Si on trouve une requête du type :

```
EXEC SQL SELECT DISTINCT a4 FROM A WHERE a3 = :var-a3 END-EXEC
```

et que cette requête n'est pas accompagnée de la définition d'un curseur, c'est que le concepteur est certain de ne retrouver qu'une seule valeur distincte de `a4`. `a3` est donc le déterminant et `a4` le déterminé. Notons que nous avions déjà fait le même type de recherche dans la partie consacrée à la recherche des identifiants (en fait la différence se situe dans le fait qu'ici on utilise un `DISTINCT`).

### 3.4.5 Recherche dans l'extension.

Il y a de nombreux algorithmes pour rechercher les DF dans l'extension d'une BD. Nous allons en décrire un, simple mais peu efficace et nous donnerons ensuite quelques améliorations possibles. Il s'agit de l'algorithme donné dans [CAS93].

#### Algorithme de recherche

Dans l'algorithme, toute combinaison possible d'attributs est un déterminant possible. Pour éviter une certaine redondance dans le travail on va analyser les DF par ordre croissant du nombre d'attributs présents dans le déterminant. Pour une table R(a, b, c, d) l'odre serait donc a, b, c, d, ab, ac, ad, bc, bd, cd ,abc, abd, acd, bcd. Cet ordre évite qu'un déterminant soit analysé avant que ses composants ne soient déjà analysés.

Pour chaque déterminant X, chaque attribut appartenant à {R • X} est un déterminé candidat. Ces attributs candidats ne sont utilisés comme possibles déterminés d'une DF `A` que si on n'a pas déjà retrouvé une DF `B` dont le déterminant est contenu dans le déterminant de la DF `A` et dont le déterminé est le même. Sinon cette paire déterminant - déterminé est une DF redondante (par augmentation). De cette façon seules les DF minimales sont obtenues. En outre, une fois qu'une DF est retrouvée, on peut calculer les DF transitives de façons à les éliminer et à ne pas les inclure dans la recherche.

Pour chacune des DF minimales ainsi formées, on analyse l'extension pour tester si elle satisfait à la condition. La condition peut être testée simultanément pour toutes les DF ayant le même déterminant. Il faut retrouver les lignes qui ont la même valeur pour le déterminant et voir si la valeur du déterminé est toujours la même.

Cette technique est bien sûr extrêmement coûteuse en temps d'accès puisqu'il faut pour chaque valeur disctincte du déterminant rechercher tous les tuples qui ont cette valeur. La table est donc examinée beaucoup de fois.

## Possibilités d'amélioration

Nous allons envisager quatre façons d'améliorer l'algorithme que nous venons de voir.

1. Puisque la plupart des identifiants peuvent être retrouvés sans une recherche dans l'extension, et qu'un identifiant détermine fonctionnellement tous les attributs d'une table, on n'a pas besoin d'inclure les DF dues à des identifiants déja retrouvés.
2. Si le dictionnaire des données du SGBD contient des méta-informations sur la cardinalité de chaque attribut, on peut l'utiliser pour déterminer si un certain attribut est un déterminé possible pour un certain déterminant. En effet si la cardinalité du déterminant est plus grande ou égale à la cardinalité du déterminé, alors il est possible qu'il y ait une DF, sinon c'est impossible.
3. On peut pré-ordonner la relation en utilisant comme clé de tri le déterminant. Ainsi lors de la recherche des tuples ayant le même déterminant, les temps d'accès seront plus faibles car il seront consécutifs. Le problème ici est évidemment le coût du tri. Malgré tout, le coût total devrait être moins élevé.
4. Plutôt que de trier la table pour chaque déterminant, on peut ne trier que sur des clés de tri 'maximales'. Par exemple, si la table est `R(a, b, c, d)`, la clé de tri sera abc, bcd, cda, dab, ac, bd. Ces clés de tri fournissent en même temps un tri pour les autre déterminants à savoir a, b, c, d, ab, bc, cd, ad. Remarquons qu'avec cette méthode, l'ordre de vérification ne sera pas déterminé par le besoin d'éviter les déterminants non minimaux mais par la clé de tri.

# 3.5 LES CONTRAINTES REFERENTIELLES.

Un attribut `b1` (ou un ensemble d'attributs `b1`) appartenant à une table `A` est apellé attribut de référence (ou clé étrangère) lorsque ses valeurs sont utilisées pour référencer une ligne d'une table `B`. Pour être plus précis considérons que la table `B` a comme identifiant `B.b1`. Si on a une contrainte d'inclusion `A.b1` $\subseteq$ `B.b1` et donc que chaque valeur de `A.b1` (non-nulle) trouve sont équivalent dans `B.b1`, on peut dire que `A.a1` permet de 'désigner' une ligne de `B` (la ligne désignée étant la ligne de `B` qui a comme valeur d'identifiant la valeur de `A.b1`). Cette propriété est appelée contrainte référentielle. Elle consiste donc en une contrainte d'inclusion dont le côté droit est un identifiant.

Un cas particulier de contrainte référentielle est la contrainte d'égalité qui consiste en deux contrainte d'inclusion réciproques (dont au moins un des attributs mis en cause est identifiant).

Les contraintes référentielles vont nous permettre lors de la conceptualisation de retrouver des types d'associations one-to-many. Elle sont en effet issues de la suppression d'une association comme le montre la transformation de la figure 3.5.

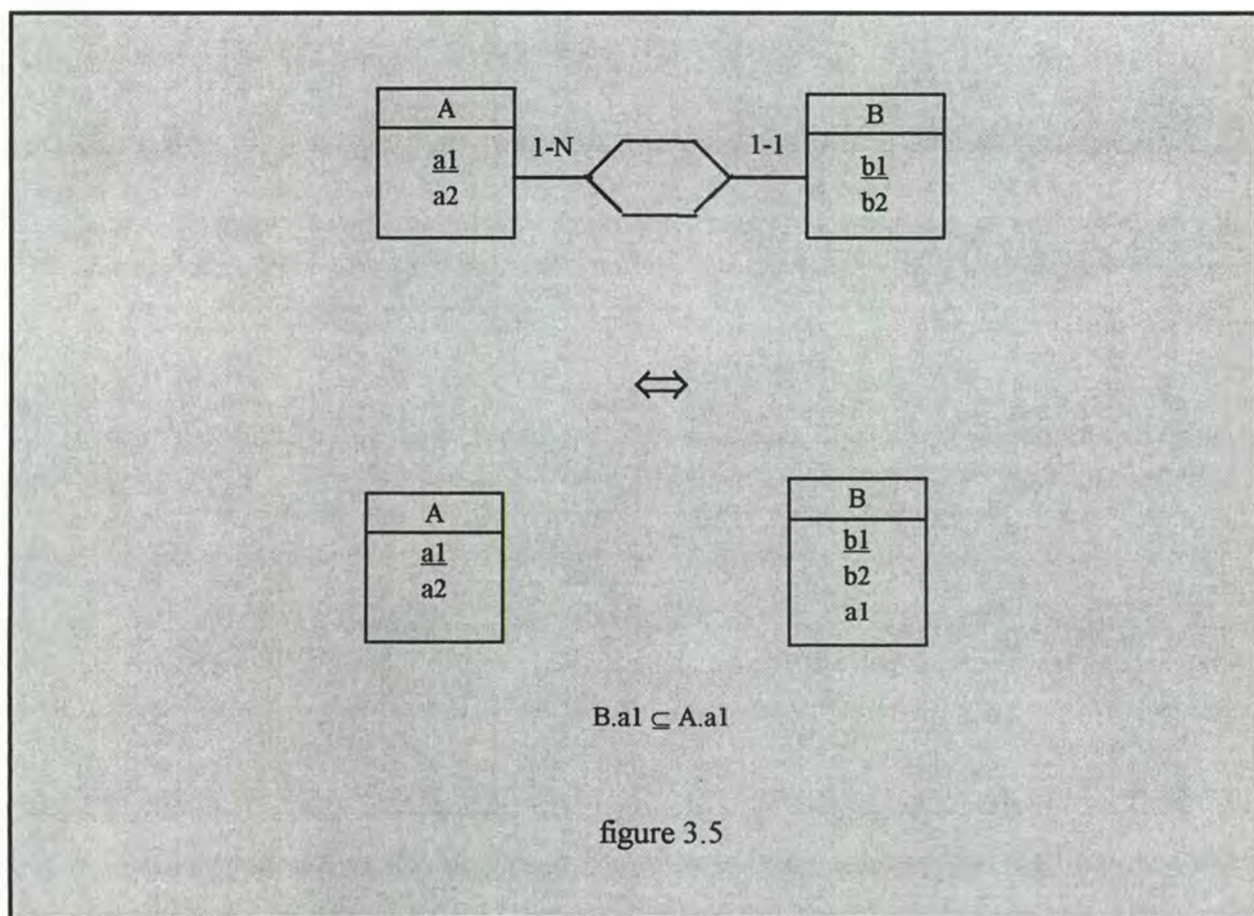



figure 3.5

Nous verrons d'abord comment une clé étrangère peut être déclarée explicitement (dans les SGBD récents). Ensuite nous expliquerons comment les noms des attributs peuvent refléter la présence d'une telle clé. Nous examinerons ensuite les vues et les checks. Nous passerons alors aux comportements imposés aux données : nous verrons la gestion des clés étrangères en distinguant le cas d'une contrainte d'inclusion du cas d'une contrainte d'égalité. Enfin, nous envisagerons une recherche dans l'extrension et dans les consultations : nous verrons que les jointures réalisées à partir des instructions de sélection peuvent aussi nous fournir des indices sur la présence d'une contrainte référentielle.

Remarque

Les recherches dans les checks et dans l'extension vont nous permettre d'émettre des hypothèses au sujet des contraintes d'inclusions (et pas nécessairement au sujet des contraintes référentielles). Notons que les contraintes d'inclusion qui ne sont pas des contraintes référentielles sont également importantes à retrouver. D'une part, nous nous en servirons dans la recherche des types d'entités manquants. D'autre part, elles sont nécessaires à la recherche des contraintes d'égalité. En effet, une contrainte d'égalité (qui consiste en deux contraintes d'inclusion réciproques) peut ne pas impliquer deux identifiants (c-à-d qu'une des deux contraintes d'inclusion qui la composent peut ne pas être une contrainte référentielle).

Les développements qui suivent se baseront sur le schéma inférieur de la figure 3.5[9].

## *3.5.1 Recherche dans la déclaration du schéma*

Dans les SGBD récents, il est possible de définir les clés étrangères explicitement. On aura par exemple dans les déclarations attachées à la table B :

```
FOREIGN KEY (a1) REFERENCES A (a1)
```

Cela signifie qu'il y a une contrainte d'inclusion `B.b1` $\subseteq$ `A.a1`. Cette déclaration n'est valable que si `A.a1` a été déclaré `PRIMARY KEY`.

Remarque

Une clé étrangère est souvent indexée. Les index fournissent donc aussi un indice pour la recherche des clés étrangères.

[9]Nous considérerons en outre que l'identifiant `A.a1` représente un attribut obligatoire.

### 3.5.2 Recherche sur les noms

Il est possible de retrouver des contraintes référentielles par une analyse sur les noms. Souvent une partie ou l'entièreté du nom de l'identifiant peut se retrouver dans la clé étrangère, de même que le nom de la table contenant l'identifiant référencé. Il se peut aussi parfois que la clé étrangère soit préfixée ou postfixée par "FK" ou "REF",...

### 3.5.3 Recherche dans les vues

On peut faire le même genre de recherche que pour les identifiants.

Supposons qu'on ait déclaré la vue suivante :

```
CREATE VIEW nom-vue (..., att-1,...)
       AS SELECT ...
                 CONCAT(att-2, att-3)
                 ...
          FROM R1
```

Cette vue permet d'accéder aux attributs `att-2` et `att-3` en une seule fois. Ce genre de construction peut-être utilisé lorsque (`att-2, att-3`) forment une clé étrangère pour faciliter la dénomination de celle-ci. Lors d'une jointure par exemple, il ne faudra utiliser qu'un seul attribut (`att-1`) pour désigner la clé étrangère de la table.

### 3.5.4 Recherche dans les checks

On pourrait avoir un check défini sur la table `B`. Il serait de la forme :

```
CHECK (a1 IN (SELECT a1 FROM A))
```

Ce check exprime directement la contrainte d'inclusion entre `B.a1` et `A.a1`.

### 3.5.5 Recherche dans les comportements imposés aux données

Nous allons examiner ici comment le concepteur peut s'assurer du respect d'une contrainte référentielle à l'aide de triggers et de configurations de requêtes. Nous évoquerons d'abord le cas où la clé étrangère et l'identifiant référencé sont impliqués dans une contrainte d'inclusion et nous présenterons ensuite ce qu'il faut ajouter s'il y a une contrainte d'égalité.

## 3.5.5.1 Contraintes d'inclusion

### 3.5.5.1.1 Définition des UC

*A) Ajout d'une ligne dans A*

```
NO PROTOCOL
```

*B) Suppression d'une ligne de A*

```
DELETE a FROM A
ou bien PRECOND ¬∃ b ∈ B | b[a1] = a[a1]
ou bien IMPLIES
        ou bien REPLACE ALL b | b[a1] = a[a1] BY b' IN B | b'[a1] = NULL
        ou bien DELETE b FROM B | b[a1] = a[a1]
```

Lorsqu'on supprime une ligne de A, il y a une valeur de A.a1 en moins. Dès lors :

- ou bien on vérifie qu'il n'y avait pas de B.a1 qui lui correspondait (précondition);
- ou bien on affecte une valeur nulle aux B.a1 qui lui correspondaient (cette alternative n'est possible que si B.a1 représente un attribut facultatif)
- ou bien on supprime les lignes de B pour lesquelles l'attribut a1 référencait la ligne de A supprimée (cette alternative est surtout utilisée lorsque B.a1 représente un attribut obligatoire).

*C)Modification d'une ligne de A*

```
REPLACE a BY a' IN A | a[a1] <> a'[a1]
ou bien PRECOND ¬∃ b ∈ B | b[a1] = a[a1]
ou bien IMPLIES
        ou bien REPLACE ALL b | b[a1] = a[a1]
                BY b' IN B | b'[a1] = a'[a1]
        ou bien REPLACE ALL b | b[a1] = a[a1] BY b' IN B | b'[a1] = NULL
        ou bien DELETE b FROM B | b[a1] = a[a1]
```

Comme les valeurs de B.a1 sont inclues dans les valeurs de A.a1, si on modifie une valeur de A.a1, il faut :

- ou bien vérifier qu'elle n'avait aucun correspondant dans B.a1 (précondition);
- ou bien modifier les B.a1 correspondant à l'ancienne valeur de a.a1 en leurs affectant la nouvelle valeur de a.a1 (a'.a1);
- ou bien affecter la valeur nulle aux B.a1 correspondant à l'ancienne valeur de a.a1 (cette alternative n'est possible que si B.a1 représente un attribut facultatif);

- ou bien supprimer les lignes de `B` dont l'attribut `a1` est égal à la nouvelle valeur de `a.a1` (cette alternative est surtout utilisée lorsque `B.a1` représente un attribut obligatoire).

*D) Ajout d'une ligne dans B*

```
INSERT b IN B | b[a1] <> NULL
PRECOND ∃ a ∈ A | a[a1] = b[a1]
```

Lorsqu'on ajoute une ligne dans `B` pour laquelle la clé étrangère est non-nulle, il faut vérifier qu'elle a un correspondant dans `A.a1`.

*E) Suppression d'une ligne de B*

```
NO PROTOCOL
```

*F) Modification d'une ligne de B*

```
REPLACE b BY b' IN B | b'[a1] <> NULL
PRECOND  ∃ a ∈ A | a[a1] = b'[a1]
```

### 3.5.5.1.2 Recherche dans les triggers

*A) Suppression d'une ligne de A*

La précondition pourrait être implémentée par un trigger de vérification du genre :

```
CREATE TRIGGER nom-trigger ON A
FOR DELETE
AS IF EXISTS (SELECT * FROM B, deleted WHERE B.a1 = deleted.a1)

BEGIN
     gestion-erreur
END
```

Ce trigger vérifie que le `A.a1` supprimé n'avait pas de correspondant dans `B`.

La première alternative de l'`IMPLIES` sera implémentée par un trigger d'action affectant la valeur nulle aux attributs `B.a1` référençant la ligne de `A` supprimée

```
CREATE TRIGGER nom-trigger ON A
FOR DELETE
AS UPDATE B
   SET B.a1 = NULL
   FROM deleted
   WHERE B.a1 = deleted.a1
```

A la deuxième alternative de l'`IMPLIES` correspondra un trigger d'action effectuant une suppression des ligne de `B` dont l'attribut `B.a1` référençait la ligne de `A` supprimée.

```
CREATE TRIGGER nom-trigger ON A
FOR DELETE
AS DELETE B
   FROM B, deleted
   WHERE B.a1 = deleted.a1
```

*B) Modification d'une ligne de A*

La précondition pourrait être implémentée par le trigger de vérification suivant :

```
CREATE TRIGGER nom-trigger ON A
FOR UPDATE
AS IF UPDATE (a1) AND
     EXISTS (SELECT * FROM B, deleted
             WHERE B.a1 = deleted.a1 )

BEGIN
     gestion-erreur
END
```

Ce trigger vérifie que l'ancienne valeur de `A.a1` n'avait pas de correspondant dans `B`.

Remarque

Le `IF UPDATE (a1)` sert à vérifier que `A.a1` a fait l'objet d'un `UPDATE`. Cependant, il n'est pas certain que `A.a1` a été effectivement modifié. Pour simplifier les écritures, nous considérons qu'il a été modifié. Autrement, le `IF` aurait la forme suivante :

```
IF UPDATE (a1) AND ((SELECT a1 FROM inserted) <> (SELECT a1 FROM deleted))
```

Le trigger d'action correspondant à la première alternative `IMPLIES` sera de la forme :

```
CREATE TRIGGER nom-trigger ON A
FOR UPDATE
AS IF UPDATE (a1)

BEGIN
   UPDATE B
   SET B.a1 = (SELECT a1 FROM inserted)
   FROM deleted
   WHERE B.a1 = deleted.a1
END
```

Le trigger d'action correspondant à la mise à nulle des attributs `B.a1` (deuxième alternative de l'`IMPLIES`) sera de la même forme, mais `inserted.a1` sera remplacé par `NULL`.

Le trigger correspondant à la suppression de ligne de `B` dont l'attribut `B.a1` référençait la ligne modifiée (troisième alternative de l'`IMPLIES`) sera similaire à celui de la suppression dans A :

```
CREATE TRIGGER nom-trigger ON A
FOR UPDATE
AS IF UPDATE(a1)

BEGIN
   DELETE B
   FROM B, deleted
   WHERE B.a1 = deleted.a1
END
```

*C) Ajout d'une ligne dans B*

On aura un trigger de vérification du genre :

```
CREATE TRIGGER nom-trigger ON B
FOR INSERT
AS IF UPDATE(a1)
   AND NOT EXISTS (SELECT * FROM A, inserted
                   WHERE A.a1 = inserted.a1)

BEGIN
     gestion-erreur
END
```

Ce trigger vérifie que lors de l'ajout d'une ligne de `B` (avec une valeur non-nulle pour `B.a1`), il doit exister une ligne de `A` ayant une valeur pour `A.a1` équivalente à la valeur de l'attribut `a1` de la ligne insérée.

*D) Modification d'une ligne de B*

On aura le même trigger de vérification que pour l'ajout dans B, à la condition de déclenchement près.

*Conclusion sur les triggers*

- Les **triggers de vérification** attachés à la table contenant l'identifiant sont des triggers déclenchés par une modification ou une suppression dans cette table. Ils vérifient qu'il n'y aura pas de clé étrangère qui référence un élément inexistant. Les triggers de vérification attachés à la table contenant la clé étrangère sont déclenchés par une insertion ou une modification dans cette table. Ils vérifient qu'il y a bien un identifiant d'une autre table qui sera référencé.
- On constate que les **triggers d'action** ne sont attachés qu'à la table contenant l'identifiant, et qu'ils sont déclenchés par une suppression ou une modification sur cette table. Ils ont comme action de mettre à jour les clé étrangères ou de supprimer des lignes de l'autre table.

### 3.5.5.1.3 Recherche dans les requêtes

*A) Suppression d'une ligne de A*

Supposons que l'on supprime la ligne de A identifiée par le contenu de la variable `var-a1`.

La précondition se traduira par la requête :

```
EXEC SQL (SELECT * FROM B WHERE B.a1 = :var-a1)  END-EXEC
```

Si cette requête donne un résultat (`SQLCODE = 0`), il y aura un traitement d'erreur (la valeur de l'identifiant de la ligne supprimée se trouvait dans `B.a1`). Sinon, la suppression pourra se faire.

La première alternative de l'`IMPLIES` pourra être réalisé à l'aide des deux requêtes suivantes (ces deux requêtes pouvant être permutées) :

```
EXEC SQL DELETE FROM A WHERE a1 = :var-a1 END-EXEC

EXEC SQL  UPDATE B SET a1 = NULL WHERE a1 = :var-a1  END-EXEC
```

La deuxième alternative de l'`IMPLIES` pourra être réalisée de la façon suivante :

```
EXEC SQL DELETE FROM A WHERE a1 = :var-a1 END-EXEC

EXEC SQL DELETE FROM B WHERE a1 = :var-a1 END-EXEC
```

Ces deux requêtes peuvent également être permutées.

*B) Modification d'une ligne de A*

Supposons que l'on modifie la ligne de A identifiée par la variable var-a1, et que l'on remplace la valeur de l'attribut a1 par var-new-a1 (var-a1 étant différent de var-new-a1).

Comme dans le cas de la suppression, la précondition de l'UC peut donner lieu à la requête:

```
EXEC SQL (SELECT * FROM B WHERE B.a1 = :var-a1)  END-EXEC
```

Si cette requête donne un résultat, il y aura un traitement d'erreur (l'ancienne valeur de A.a1 se trouvait dans B.a1). Sinon, on fera la modification.

La première alternative de l'IMPLIES pourra être réalisée par une requête de modification dans B accompagnant la requête de modification dans A.

```
EXEC SQL UPDATE A SET A.a1 = :var-new-a1 WHERE A.a1 = :var-a1 END-EXEC

EXEC SQL  UPDATE B SET B.a1 = :var-new-a1 WHERE B.a1 = :var-a1  END-EXEC
```

Les deux requêtes peuvent être permutées.

Pour la deuxième alternative de l'IMPLIES, il y aura les mêmes requêtes, si ce n'est que NULL sera mis dans B.a1 (et non var-new-a1).

Pour la troisième alternative de l'IMPLIES, la requête de modification dans A sera accompagnée d'une requête de suppression dans B.

```
EXEC SQL UPDATE A SET A.a1 = :var-new-a1 WHERE A.a1 = :var-a1 END-EXEC

EXEC SQL DELETE FROM B WHERE B.a1 = :var-a1 END-EXEC
```

Ces requêtes peuvent être permutées.

*C) Ajout d'une ligne dans B*

Supposons que l'on veuille ajouter la ligne `(var-b1 , var-b2 , var-a1)`.

Une des façons de traduire l'UC sous forme de requêtes serait :

```
EXEC SQL  SELECT A.a1 FROM A WHERE A.a1 = :var-a1  END-EXEC
```

Si le `SELECT` n'a rien trouvé (`SQLCODE = 100`), c'est qu'il n'y a aucun `A.a1` correspondant à `var-a1`, et il y aura alors un traitement d'erreur. Sinon, on peut faire l'insertion :

```
EXEC SQL  INSERT INTO B VALUES (:var-b1 , :var-b2 , :var-a1)  END-EXEC
```

*D) Modification d'une ligne de B*

C'est le même principe que pour l'insertion.

## 3.5.5.2 Contraintes d'égalité

Les développements qui suivent se feront sur base du schéma inférieur de la figure 3.5 dans lequel la contrainte d'inclusion a été remplacée par les contraintes

`B.a1` $\subseteq$ `A.a1 (1)`

`A.a1` $\subseteq$ `B.a1 (2)`

La contrainte (2) a donc été rajoutée par rapport au schéma initial. Ces deux contraintes correspondent à la contrainte

`A.a1 = B.a1`

Nous allons nous contenter de voir les UC de gestion de la contrainte (2). Ceux-ci sont assez semblables à ceux présentés ci-dessus. Nous ne développerons pas les triggers et les requêtes.

*A) Ajout d'une ligne dans A*

```
INSERT a IN A
ou bien PRECOND ∃ b ∈ B | b[a1] = a[a1]
ou bien IMPLIES ou bien INSERT b IN B | b[a1] = a[a1]
                ou bien REPLACE ANY b | b[a1] = NULL
                        BY b' IN B | b'[a1] = a[a1]
```

Lorsque l'on ajoute une ligne a dans A, il faut s'assurer que la contrainte (2) est respectée. Pour ce faire :

- ou bien on vérifie que la valeur de a.a1 existe bien dans B.a1 (précondition) ;
- ou bien on ajoute une ligne b dans B. La clé étrangère (b.a1) de cette ligne référence a ;
- ou bien on modifie une (ou plusieurs) lignes b de B (dont l'attribut B.a1 est nul) en affectant la valeur de a.a1 à b.a1.

*B) Suppression d'une ligne de A*

```
NO PROTOCOL
```

*C) Modification d'une ligne de A*

```
REPLACE a BY a' IN A
ou bien PRECOND ∃ b ∈ B | b[a1] = a'[a1]
ou bien IMPLIES REPLACE ANY b
                BY b' IN B | b'[a1] = a[a1]
```

Lorsqu'on modifie une ligne a de A, pour que la contrainte (2) soit respectée il faut :

- ou bien vérifier que la nouvelle valeur de a.a1 existe bien dans B.a1 (précondition);
- ou bien modifier une (ou plusieurs) lignes b de B en affectant la nouvelle valeur de a.a1 à b.a1.

*D) Ajout d'une ligne dans B*

```
NO PROTOCOL
```

*E) Suppression d'une ligne de B*

```
DELETE b FROM B
ou bien PRECOND ∃ b' ∈ B | b' <> b AND b'[a1] = b[a1]
ou bien IMPLIES DELETE a FROM A | a[a1] = b[a1]
```

<u>Remarque</u>

L'IMPLIES aura normalement lieu si

```
¬∃ b' ∈ B | b' <> b AND b'[a1] = b[a1]
```

Si on supprime une ligne de B, il faut qu'il y aie une autre ligne de B qui a la même valeur de clé étrangère. Si ce n'est pas le cas il faut supprimer la ligne de A qui était référencée par la valeur de clé étrangère supprimée.

*E) Modification d'une ligne de B.*

```
REPLACE b BY b' IN B
ou bien PRECOND ∃ b'' ∈ B | b'' <> b AND b''[a1] = b[a1]
ou bien IMPLIES DELETE a FROM A | a[a1] = b[a1]
```

Remarque

L'`IMPLIES` aura normalement lieu si

```
¬∃ b'' ∈ B | b'' <> b AND b''[a1] = b[a1]
```

Cette UC est très proche de celle d'une suppression dans B.

### 3.5.6 Recherche dans l'extension de la base de données

Pour rechercher les contraintes d'inclusion dans l'extension de la BD, on peut utiliser une requête du type :

```
SELECT * FROM B WHERE B.a1 NOT IN (SELECT a1 FROM A)
```

Si cette requête ne donne pas de réponse, alors on est sûr que tous les `B.a1` sont inclus dans les `A.a1`.

Le coût de cette méthode est très élevé, car il faut faire cette requête pour toutes les combinaisons d'attributs. Voici quelques moyens d'optimisation, tirés de [CAS93] :

- prendre en compte le fait que les attributs présents des 2 côtés de la contrainte d'inclusion doivent être du même type ;
- ne considérer pour le côté droit que les identifiants ;
- ne considérer que les contraintes d'inclusion entre au plus 3 attributs composants semble raisonnable ;
- certains types de données ne devraient pas être inclus dans la recherche. Par exemple, les réels, les booléens, etc. ;

- commencer la recherche par des contraintes d'inclusion entre deux attributs. Ensuite, considérer les contraintes concernant des compositions d'atributs. En effet, si A⊆B et C⊆D alors AC⊆BD. Cela permet d'éviter certaines recherches ;
- ordonner les tables sur les arguments respectifs de la contrainte avant la recherche pour minimiser les temps d'accès ensuite.

**Le problème des contraintes transitives**

Les contraintes d'inclusion possèdent la propriété de transitivité. En effet :

Si A ⊆ B et B ⊆ C alors A ⊆ C

Les contraintes d'inclusion transitives ne doivent pas être prises en compte lors de l'analyse ultérieure (elles sont en quelque sorte redondantes).

Après une recherche dans l'extension , il faut donc les éliminer. Cette élimination ne peut pas toujours être automatisée. Ainsi par exemple, si on trouve les contraintes suivantes :

```
(1) att1 ⊆ att2
(2) att1 = att3
(3) att3 ⊆ att2
```

On aura :

```
(1) + (2) --> (3)
(2) + (3) --> (1)
```

Le problème est que l'on ne sait pas quelle contrainte transitive ( (1) ou (3) ) il faut supprimer. Il faut alors demander l'avis de l'utilisateur.

## 3.5.7 Recherche dans les consultations.

Nous allons nous intéresser ici à l'analyse des requêtes de sélection présentes dans le programme d'application, dans la déclaration de triggers, de checks, ou de vues. Les requêtes intéressantes sont celles dans lesquelles une jointure entre deux ou plusieurs tables est effectuée. [PET94] et [AND94] présentent une méthode de Reverse Enginneering basée sur l'analyse des conditions de jointure.

Supposons que $A.a_k$ ( ou $A.a_{k_1}$ , . . . ,$A.a_{k_i}$) est un identifiant et que $B.b_l$ (ou $B.b_{l_1}$ , . . . , $B.b_{l_i}$) est une clé étrangère qui référence cet identifiant. Examinons le genre de requêtes qui nous intéresse.

- 

```
SELECT ... FROM A , B WHERE ... AND (A. ak = B.bl ) ...
```

ou

```
SELECT ... FROM  A WHERE A.ak  = (SELECT B.bl  FROM B WHERE ...)
```

Ces requêtes effectuent la jointure $A(A_k) * B(b_l)$

•

```
SELECT ... FROM A, B WHERE ... AND (A.ak1 = B.bl1) ...
                                AND (A.aki = B.bli)
```

ou

```
SELECT ... FROM A
WHERE CONCAT(A.Ak1 ,...,B.bki) = SELECT CONCAT (B.bl1,...,B.bli)
                                 FROM B WHERE ...)
```

Ces requêtes effectuent la jointure $A(a_{k_1}, ..., a_{k_i}) * B(b_{l_1}, ..., b_{l_i})$

De telles jointures ont pour effet de 'coupler' chaque ligne de B avec la ligne de A que la clé étrangère référence. Si on prend comme hypothèse qu'une ligne correspond conceptuellement à une entité, ce mécanisme permet d'extraire de l'information concernant plusieurs entités (qui sont liées par un lien de clé étrangère). Ce mécanisme sera bien évidemment très souvent utilisé. Dès lors si on retrouve une jointure dans une requête, il y a de fortes chances pour qu'elle compare une clé étrangère et l'identifiant qu'elle référence. Ceci nous fournit un indice pour la recherche des contraintes référentielles. Observons toutefois que cette jointure ne nous indique pas le sens de la contrainte (autrement dit on ne peut pas différencier a priori l'identifiant de la clé étrangère). De plus il ne s'agit que d'un indice car une jointure peut être réalisée sur d'autres arguments qu'une clé étrangère et son identifiant. Voici quelques exemples de jointures de ce type :

- Soit le schéma relationnel :

A(<u>a1</u>,a2) , B(b1,<u>c1</u>) , C(<u>c1</u>, c2)
A.a2 ⊆ B.c1 et B.c1 ⊆ C.c1

Cet exemple est particulier car on voit que la clé étrangère `A.a2` référence l'identifiant `B.c1` qui référence lui même l'identifiant `C.c1`. Par transitivité on a `A.a2` ⊆ `C.c1`. On peut donc utiliser directement `A.a2,C.c1` pour faire la jointure `A(a2) * C(c1)`. Une telle jointure permet de 'coupler' les ligne de A et de C sans passer par B. Retrouver une telle jointure nous fournirait donc une indication sur une contrainte d'inclusion transitive qui représente, en quelque sorte, un lien indirect entre A et B. Ce genre de contrainte transitive est redondante avec les deux autres et ne doit pas être prise en compte lors de l'extraction.

- Soit le schéma relationnel :

```
A(a1,a2) , B(b1,a1) , C(c1, a1)
B.a1 ⊆ A.a1 et C.a1 ⊆ A.a1
```

On remarque que `B.a1` et `C.a1` référencent le même identifiant `A.a1`. Lorsqu'on veut mettre en correspondance les lignes de B et de C, on peut faire une jointure entre B et C sur `a1` sans passer par A. On aurait alors `B(a1) * C (a1)`. Ce genre de jointure met en cause deux clés étrangères.

- Soit le schéma relationnel :

```
VOITURE(Id-voit , Cylindrée , Couleur)
PEINTURE(No-type , Couleur)
```

et soit la requête :

```
SELECT DISTINCT Id-voit
FROM VOITURE, PEINTURE
WHERE VOITURE.Couleur = PEINTURE.Couleur
```

Cette requête effectue une jointure de `PEINTURE` et `VOITURE` sur `Couleur`. L'utilité de cette requête est de retrouver les voitures dont la couleur est présente **dans** les couleurs des peintures. C'est en fait un `IN` qu'on veut implémenter entre deux attributs qui ont le même domaine. Cette jointure a comme arguments deux attributs qui ne sont ni identifiants, ni clés étrangères. Il est probable que ce genre de jointure soit assez rare mais il peut exister et doit être pris en compte.

### 3.5.7.1 Recherche basée sur les jointures

Nous venons de voir que sauf exceptions, une jointure a de bonnes chances de concerner une clé étrangère et l'identifant qu'elle référence. De telles jointures peuvent servir à une recherche dans l'extension :

Supposons qu'une jointure aie comme arguments les attributs `a1` et `a2`. On peut examiner dans l'extension si

- `a1` $\subseteq$ `a2`
- `a2` $\subseteq$ `a1`
- `a1 et/ou a2` sont identifiants

Comme il ne faut plus appliquer cette recherche que sur tous les couples d'attributs arguments d'une jointure (et non plus sur toutes les configurations possibles d'attributs), cette recherche sera beaucoup moins coûteuse que celle présentée ci-dessus (cf 3.5.6).

Cette méthode permet de retrouver non seulement des clés étrangères mais aussi les identifiants qu'elles référencent. Elle permet de retrouver seulement les liens qui donnent lieu à une jointure.

### 3.5.7.2 Extension de la méthode.

Plutôt que de se concentrer uniquement sur les jointures effectuées explicitement dans des requêtes, on peut également examiner les liens qui seraient faits entre deux tables par deux ou plusieurs requêtes sans utilisation explicite de jointure.

Voyons cela par un exemple. Soit le schéma relationnel

```
CLIENT (Num-cli, Nom-cli, Adresse-cli)
PRODUIT (Num-pro, Qte-stock, Libelle, Prix)
COMMANDE (Num-cli, Num-pro)

COMMANDE.Num-cli ⊆ CLIENT.Num-cli
COMMANDE.Num-pro ⊆ PRODUIT.Num-pro
```

Supposons qu'on cherche le libellé des produits commandés par un client dont on connaît le nom `var-nom` et l'adresse `var-ad`.

D'abord on recherche la valeur de l'identifiant `Num-cli` du client correspondant aux conditions.

```
EXEC SQL
    SELECT Num-cli INTO :var-num-cli
    FROM CLIENT
    WHERE Nom-cli = :var-nom AND Adresse-cli = :var-ad
END-EXEC
```

Ensuite, on recherche dans COMMANDE les Num-pro en correspondance avec le Num-cli retrouvé ci-dessus. Comme on obtiendra plusieurs Num-pro, il faut définir et ouvrir le curseur suivant :

```
EXEC SQL
   DECLARE CURSOR nom-curseur FOR
   SELECT Num-pro FROM COMMANDE WHERE Num-cli = :var-Num-cli
END-EXEC
```

On effectue alors une boucle qui pour chaque Num-pro fournit son libellé.

On aura par exemple

```
FETCH nom-curseur INTO :var-num-pro

WHILE   (...)
BEGIN
    ...
    EXEC SQL
      SELECT libelle INTO :var-libelle
      FROM Produit
      WHERE num-pro = :var-num-pro
    END-EXEC;
    FETCH nom-curseur INTO :var-Num-pro

END
```

Nous déduisons de tout ceci qu'un lien est effectué entre CLIENT et COMMANDE grâce à l'utilisation de la variable[10] résultant de la requête sur CLIENT comme argument du WHERE du curseur sur COMMANDE. Quant au lien entre PRODUIT et COMMANDE, il est effectué grâce à l'utilisation de la variable résultant du FETCH sur COMMANDE comme argument de la requête sur PRODUIT.

[10]ou d'une copie de cette variable.

Si on considère qu'un `FETCH` correspond à l'execution d'une requête, on peut dire qu'une jointure *implicite* est faite entre deux tables `A` et `B` lorsque la valeur d'un attribut `A.a` tirée d'une requête sur la table `A` est utilisée en tant qu'argument du `WHERE` (égalité avec un attribut `B.a`) d'une requête sur la table `B`.

Les jointures implicites offrent les même possibilité au concepteur que les jointures explicites dans une seule requête. Elles fournissent un indice pour la recherche des contraintes référentielles et sont un support à une recherche dans l'extension équivalente à celle des jointures explicites.

# CHAPITRE 4
# CONCEPTUALISATION DES STRUCTURES DE DONNEES

## 4.1 NETTOYAGE ET RENOMMAGE DU SCHEMA

Nous avons déjà évoqué cette étape dans le tome 1. Il ne nous a pas semblé utile de la développer en détail ici.

## 4.2 DE-TRADUCTION DU SCHEMA CONFORME

### *4.2.1 Les types d'associations one-to-many*

#### 4.2.1.1 Transformation classique

##### 4.2.1.1.1 Principe

Voyons comment traduire de façon classique les types d'associations one-to-many d'un schéma E/A en un schéma conforme au modèle relationnel. On applique la transformation de la figure 4.1. Par cette transformation, on supprime le type d'associations one-to-many, et on ajoute l'attribut identifiant du type d'entités assumant le rôle de connectivité maximale N dans le type d'entités assumant le rôle de connectivité 1.

Sur la figure 4.1, on voit qu'on a recopié `A.a1` dans `B` car la connectivité maximale de `ro-1` est N et la connectivité maximale de `ro-2` est 1. Si la connectivité maximale de `ro-1` avait été 1, `B.a1` serait identifiant.

On voit aussi que `B.a1` est facultatif. C'est dû au fait que la connectivité minimale de `ro-2` est 0. Si elle avait été 1, cet attribut aurait été obligatoire.

Comme la connectivité minimale de `ro-1` est 0, on a une contrainte référentielle d'inclusion. Si cette connectivité avait été 1, on aurait eu une contrainte référentielle d'égalité.

A
a1
a2
0-N
ro-1
0-1
ro-2
B
b1
b2

⇔

A
a1
a2

B
b1
b2
a1[0-1]

B.a1 ⊆ A.a1

figure 4.1

### 4.2.1.1.2 Recherche

Considérons le schéma conforme au modèle relationnel de la figure 4.2.

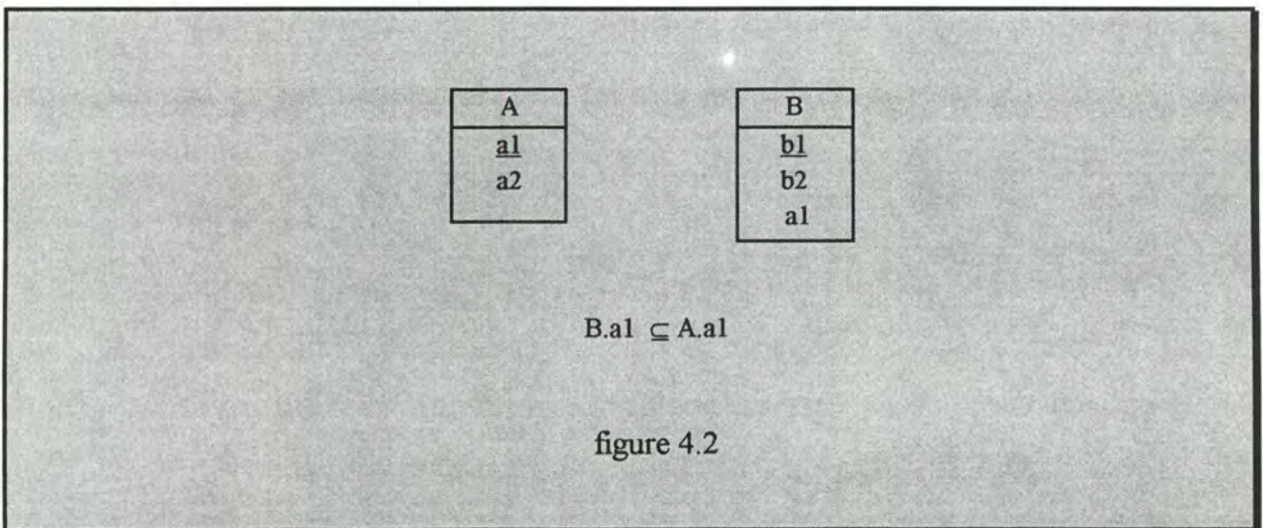



figure 4.2

De par la contrainte d'inclusion et le fait que `A.a1` est un identifiant, on sait que `B.a1` est une clé étrangère. On peut donc relier `A` et `B` par un type d'associations (figure 4.3).

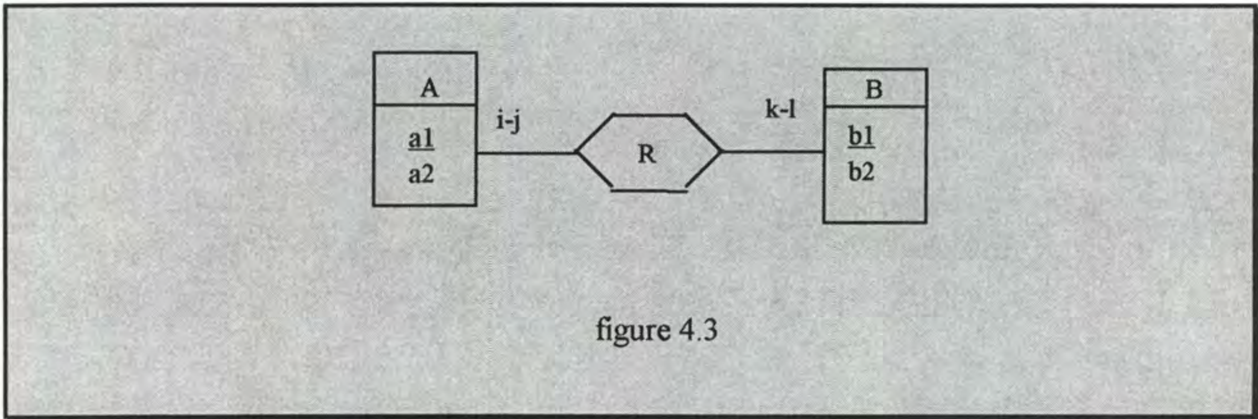



figure 4.3

Il reste maintenant à déterminer les connectivités i,j,k et l.

*Connectivités minimales*

**i = 0** si `B.a1` $\subseteq$ `A.a1`

**i = 1** si `B.a1` = `A.a1`

**k = 0** si `B.a1` est facultatif

**k = 1** si `B.a1` est obligatoire

*Connectivités maximales*

**l = 1** (de par le fait qu'il y a dans `B` une référence vers `A`)

**j = 1** si `B.a1` est identifiant

**j = N** si `B.a1` n'est pas identifiant

Pour dé-traduire un tel schéma relationnel, il suffit donc de retrouver les attributs de référence, de les éliminer, et de traduire les contraintes référentielles en types d'associations one-to-many ou one-to-one, avec les bonnes connectivités minimales. Cependant, dans le cas d'une

configuration de connectivités 1-1 - 1-1, on a une parfaite symétrie des indices. On ne peut arbitrairement supprimer l'un des 2 attributs clé étrangère. Il est possible que l'attribut clé étrangère ait été déclaré explicitement. Sinon, il faudra faire une recherche complémentaire pour retrouver la clé étrangère qu'a voulu implémenter le concepteur.

On peut déja se dire que si une des 2 tables possède 2 identifiants et l'autre un seul, il y a de fortes chances pour que la clé étrangère se situe dans la deuxième table. De même si l'un deux 2 attributs est référencé par une autre clé étrangère, il s'agit probablement d'un identifiant propre. On peut aussi bien entendu faire appel à l'utilisateur.

## 4.2.1.2 Transformation non-classique

### 4.2.1.2.1 Principe

On peut envisager plusieurs cas de transformations non-classiques d'un type d'associations one-to-many.

*Cas 1*

Considérons le schéma de la figure 4.4.

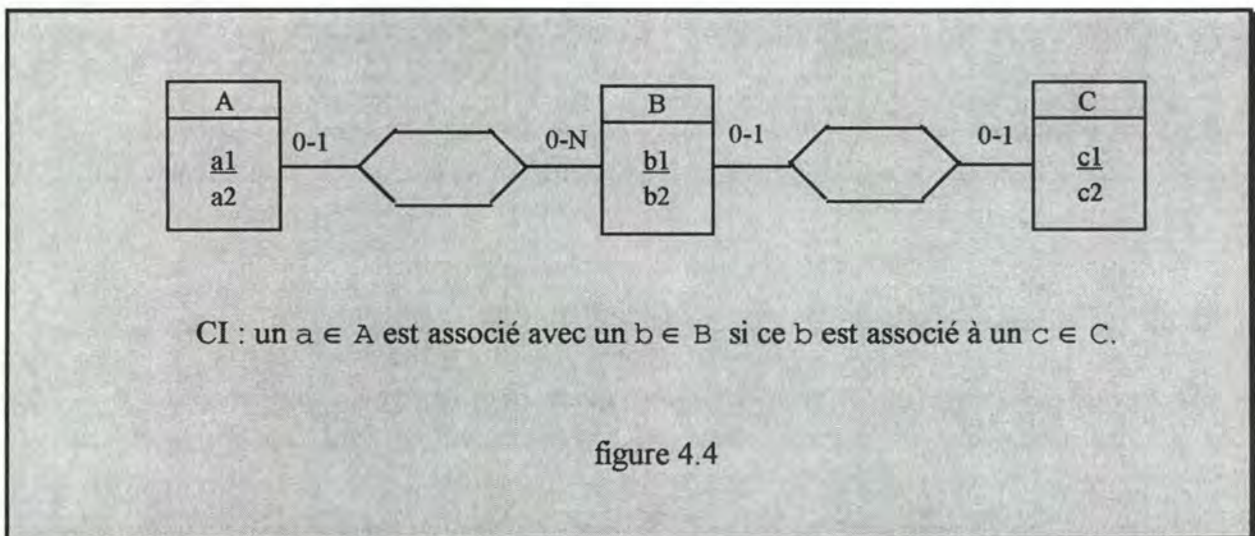



figure 4.4

Supposons que l'on aie transformé l'association entre B et C en mettant c1 dans B. Si ensuite on veut supprimer l'association entre A et B, on peut, plutôt que de recopier b1 dans A, recopier B.c1 dans A en vertu de la contrainte d'intégrité. L'originalité de cette méthode est qu'on n'utilise pas un identifiant propre de B pour la transformation.

On obtiendrait alors le schéma équivalent de la figure 4.5.

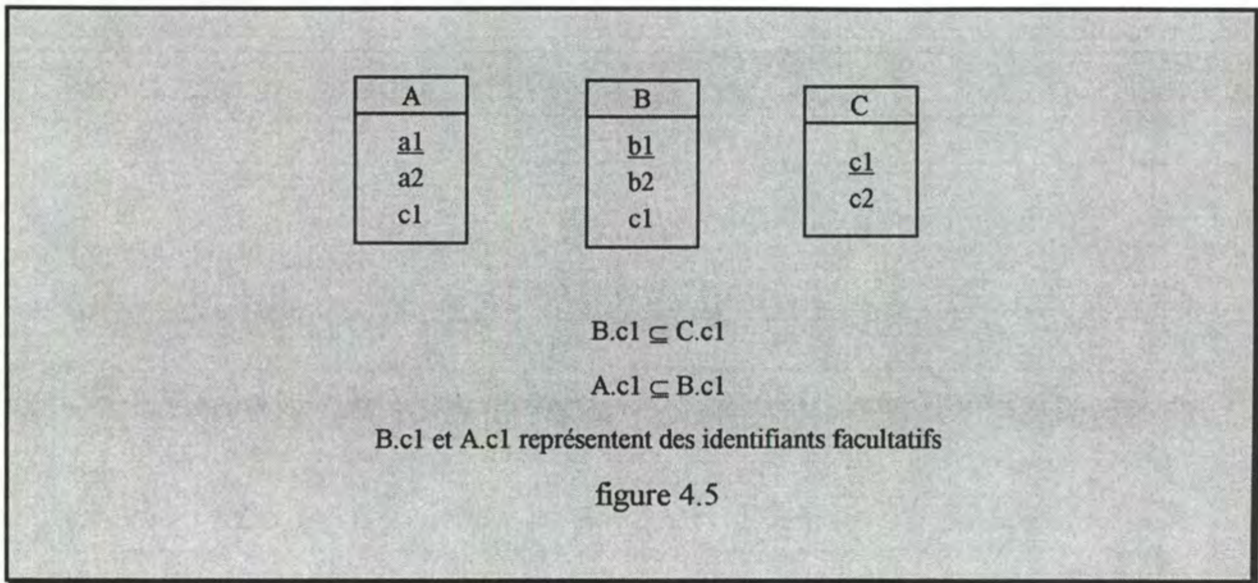


figure 4.5

La contrainte d'intégrité devient inutile puisqu'on ne peut accéder de A qu'a des éléments de B qui ont un lien avec C.

### *Cas 2*

Considérons le schéma de la figure 4.6.

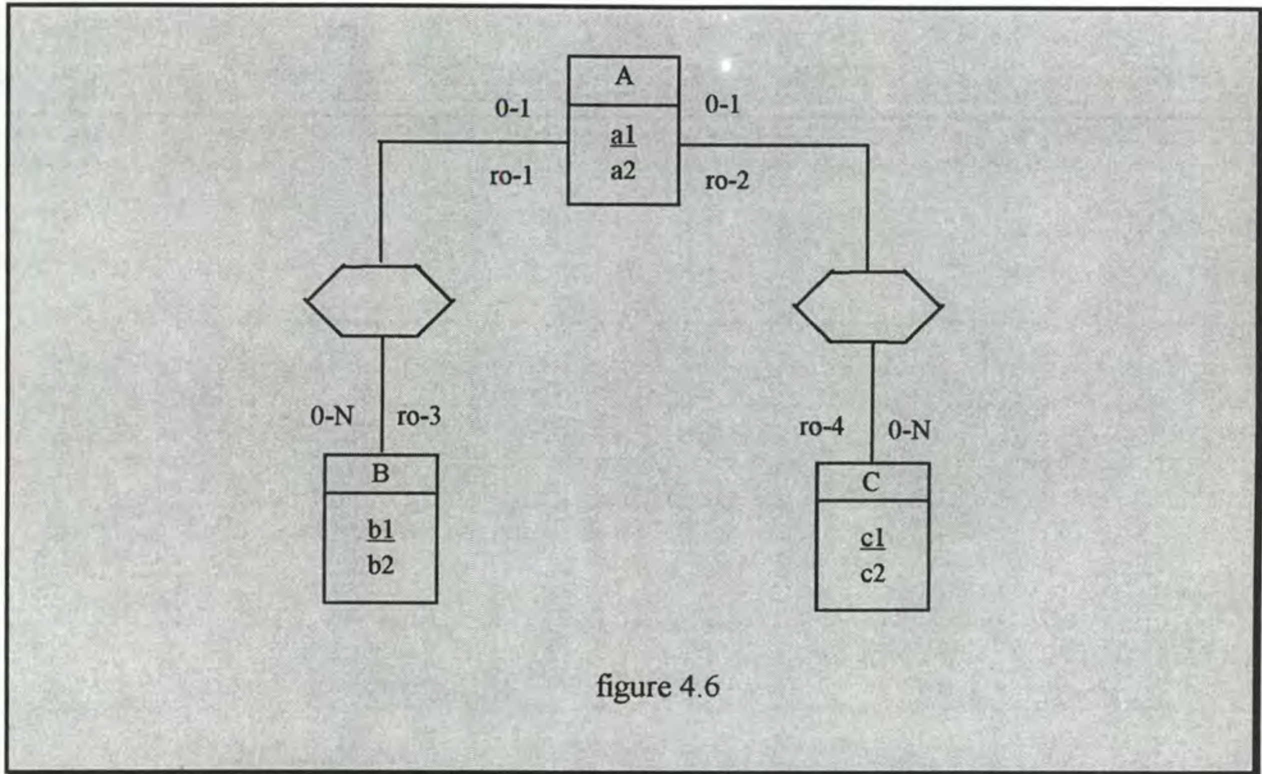


figure 4.6

Supposons tout d'abord qu'il y a une **contrainte d'exclusion** entre les rôles `ro-1` et `ro-2` et que les domaines de `b1` et `c1` sont de même type et ont une intersection vide.

Plutôt que de recopier dans `A` l'attribut identifiant de `B` et de `C`, on peut prendre comme attribut de référence un attribut `A.ref` dont le domaine = `{dom(b1) ∩ dom(c1)}`. De cette façon, lorsque `A.ref` appartiendra au domaine de `b1` (`c1`), c'est que l'élément sera lié a un élément de `B` (`C`). On peut faire le même genre de transformation même si les deux domaines ne sont pas disjoints. Il suffirait alors d'utiliser un attribut "flag" de `A` pour déterminer si l'élément est lié à un `B` ou à un `C`.

On obtiendra la contrainte d'inclusion suivante :

`A.ref ⊆ (B.b1 ∩ C.c1)`

L'originalité de cette méthode se situe dans le fait qu'un attribut clé étrangère référence plusieurs identifiants et qu'il n'y a qu'une contrainte d'inclusion pour deux liens.

Supposons maintenant qu'il y a une **contrainte d'égalité** entre les rôles `ro-1` et `ro-2` et que les deux identifiants `b1` et `c1` ont le même domaine. Supposons aussi qu'une entité de `A` est reliée à une entité de `B` et à une entité de `C` qui ont la même valeur d'identifiant.

On peut alors également ne recopier qu'un seul attribut de référence dans `A` pour supprimer les 2 types d'associations.

On aura alors les contraintes de référence suivantes :

`A.ref ⊆ B.b1`

A.ref ⊆ C.c1

Remarquons que bien qu'il y aie une seule clé étrangère pour les deux liens, il y a toujours deux contraintes d'inclusion distinctes.

Remarque

Ces deux constructions seront normalement aussi assez exceptionnelles.

### *Cas 3*

Ce cas est repris de [HAI93b] . Nous le développerons brièvement ici.

Considérons le schéma de la figure 4.7.

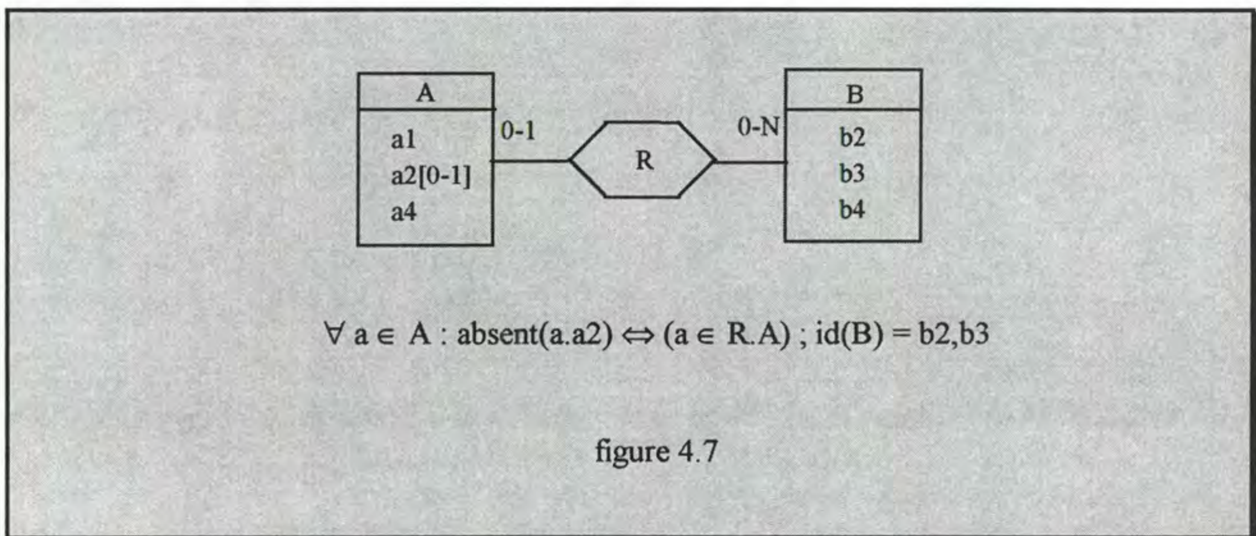



figure 4.7

La contrainte signifie que A.a2 ne prend pas de valeur ssi A joue un rôle dans R.

On peut enlever le type d'associations pour obtenir le schéma de la figure 4.8. On trouve dans [HAI93b] la démonstration de l'équivalence des deux schémas.

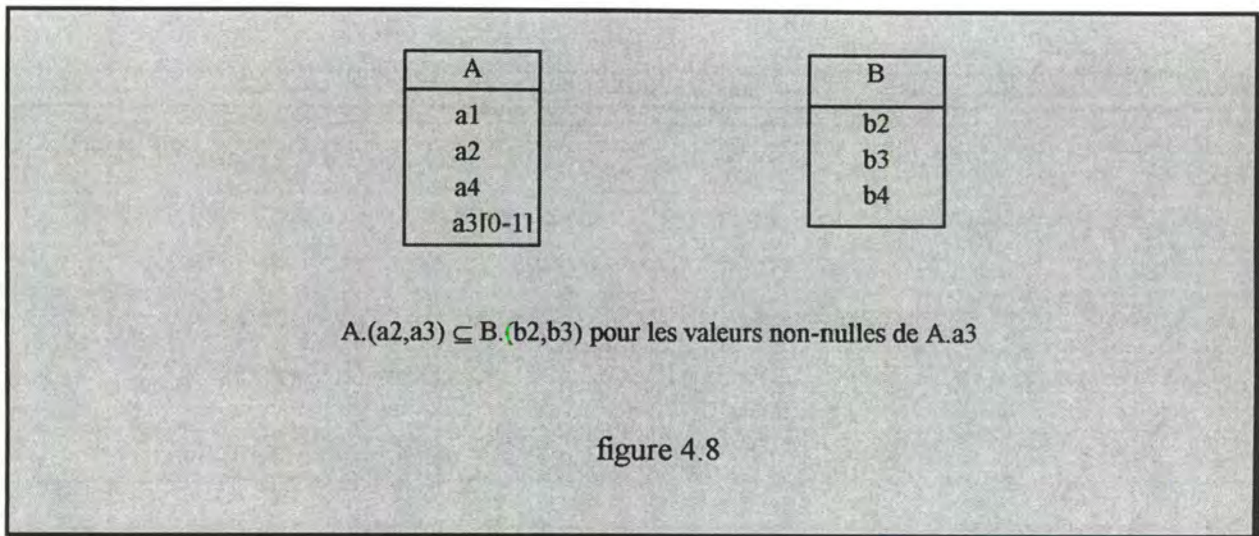


figure 4.8

On le voit ici a2 est devenu obligatoire car pour certaines entités a ∈ A il représente une valeur existante du premier schéma, et pour les autres entités a' ∈ A il représente un des composants de la clé étrangère a2,a3.

Ce qui est remarquable dans ce schéma c'est que l'un des composants (en l'occurrence a3) est optionnel. C'est comme si les A étaient "à moitié liés" aux B.

### *Cas 4*

Ce cas vient également de [HAI93b].

Considérons le schéma de la figure 4.9.

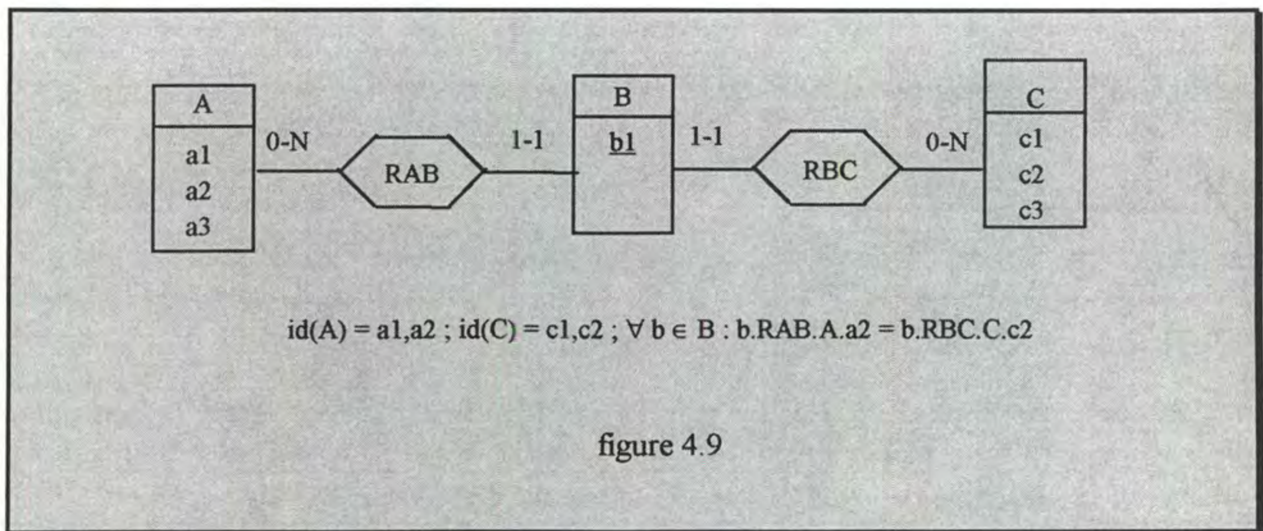


figure 4.9

On peut supprimer les types d'associations de telle façon qu'on obtienne le schéma (sémantiquement équivalent) de la figure 4.10.

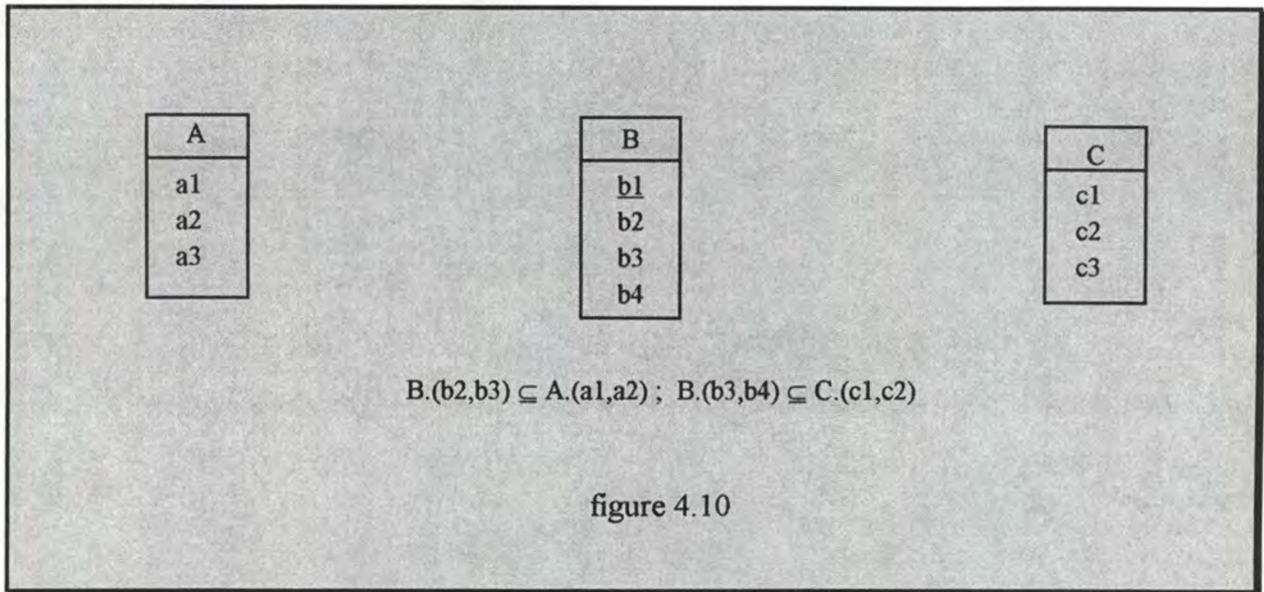



figure 4.10

Ici, la particularité c'est que les clé étrangères se recouvrent partiellement.

## *Cas 5*

Considérons le schéma de la figure 4.11.

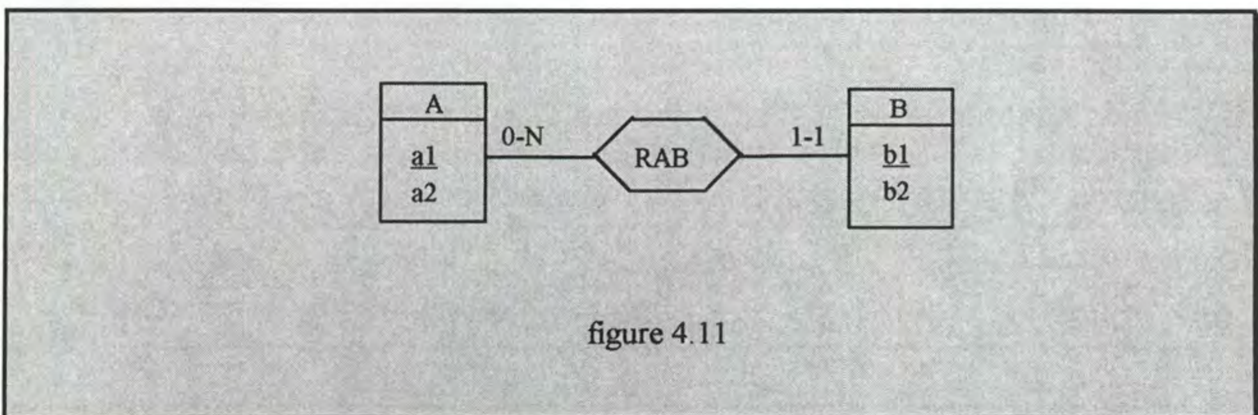



figure 4.11

Si la connectivité maximale de A-RAB est fixée (soit m), on peut remplacer le type d'associations RAB par m attributs dans A qui référenceront chacun une occurence de B. Cela permet de ne pas recopier une clé étrangère dans B.

On aurait alors le schéma relationnel de la figure 4.12.

$A(a1,\ a2,\ ref_1\ ,\ ref_2\ ,\ldots,\ ref_n)$
B(b1, b2)

$A.\ ref_i \subseteq B.b1\ ,\ i = 1..N$

figure 4.12

Ce genre de structure sera très rarement implémenté pour un type d'associations one-to-many car il devrait être plus difficile à gérer qu'une structure obtenue par la méthode classique. De plus une telle transformation impose de rajouter autant d'attributs que le nombre maximal d'associations. Or il se peut que certains $a \in A$ soient reliés à peu de $b \in B$. Cela se traduira par une perte d'efficacité tant au niveau des temps d'accès qu'au niveau de la place sur le disque.

Cependant, nous verrons qu'une telle construction peut être utilisée avangateusement pour une many-to-many. L'étude de ce genre de transformation sera donc faite dans la partie y consacrée.

### *Cas 6*

Le schéma de la figure 4.11 pourrait être traduit en relationnel par une transformation du type d'associations RAB en un type d'entités. On obtiendrait alors le schéma de la figure 4.13.

A (a1 , a2)
B (b1, b2)
RAB (a1 , b1)

$RAB.a1 \subseteq A.a1$
$RAB.b1 \subseteq B.b1$

figure 4.13

Cette transformation, peu intéressante dans le cas d'un type d'association one-to-many, sera par contre plus intéressante pour la traduction de types d'associations complexes. Nous y reviendrons plus loin.

#### 4.2.1.2.2 Recherche

Nous allons reprendre les 6 cas évoqués ci-dessus.

Pour le **cas 1**, on peut retrouver l'association entre A et B de la même façon que dans la manière classique, avec un identifiant référencé facultatif.

Le **cas 2** ne pose pas de problème si l'on considère une contrainte d'inclusion dont le côté droit est l'union de deux attributs comme deux contraintes d'inclusions.

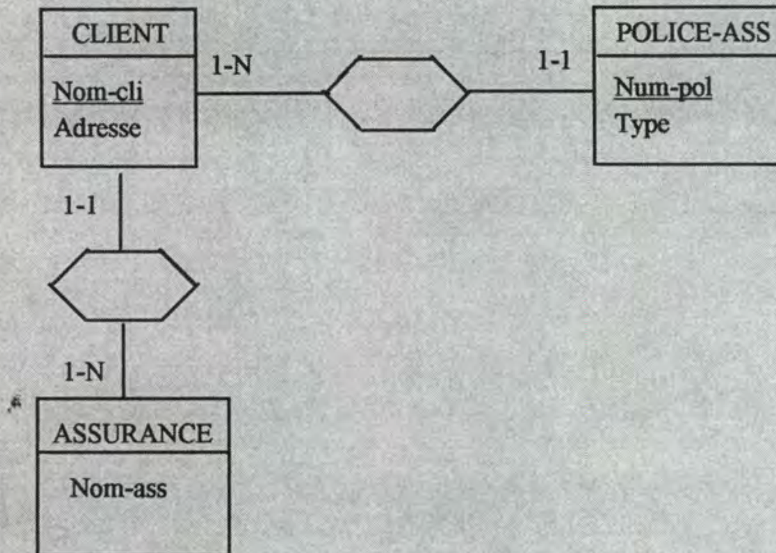
Pour le **cas 3**, il ne faudra pas éliminer tous les attributs de référence lors de la traduction en E/A mais ne garder que les attributs obligatoires de la clé étrangère qui deviendront facultatifs. Notons que la transformation évoquée au cas 3 peut prendre des formes plus complexes. Les recherches seront alors différentes.

En ce qui concerne le **cas 4**, il faudra faire attention lors de la traduction en E/A à ne pas supprimer les deux attributs formant une clé étrangère sans garder une trace de la présence d'un des deux attributs dans l'autre clé étrangère.

Les **cas 5 et 6** seront traités lors de l'étude des types d'associations complexes.

## *4.2.2 Les identifiants dont l'un des composants est un rôle.*

Considérons le schéma conceptuel de la figure 4.14.




id (POLICE-ASS) = Type, CLIENT

figure 4.14

On remarque que POLICE-ASS possède 2 identifiants, un identifiant simple et un identifiant **hybride** composé d'un attribut et d'un rôle. Pour avoir un identifiant hybride pour un type d'entités, il faut que le rôle de celui-ci soit de connectivités 1-1.

On pourrait également avoir un type d'entités identifié entièrement par un autre. Ce serait le cas ci-dessus si le deuxième identifiant de POLICE-ASS était CLIENT (uniquement). Ceci ne serait possible que si POLICE-ASS et CLIENT étaient associés par un type d'associations de connectivités 1-1 - 1-1 ou 1-1 - 0-1.

Rendons le schéma conceptuel conforme au modèle relationnel de manière classique. On obtient le schéma de la figure 4.15.

Pour retrouver l'identifiant hybride, il suffit de voir que POLICE-ASS possède un identifiant composé d'une clé étrangère et d'un attribut non clé. Lors de la traduction en E/A, on va supprimer la clé étrangère. Il faudra également traduire l'identifiant de départ.

Un type d'entités identifié entièrement par une autre aura dans le schéma relationnel un identifiant qui sera aussi clé étrangère. Cela se présentera à chaque configuration de connectivité 1-1 - 1-1 ou 1-1 - 0-1.

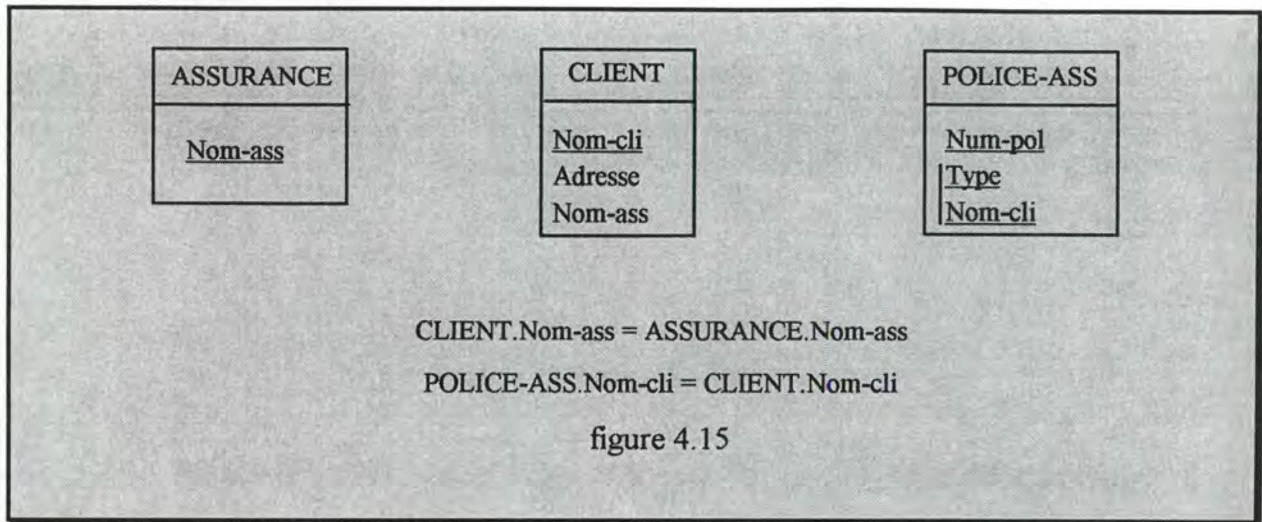


figure 4.15

## *4.2.3 Les types d'entités manquants*

On trouve une recherche des types d'entités manquants dans [CAS93]. Pour cette partie du travail, nous allons résumer cette recherche et l'adapter à la méthode utilisée dans ce travail.

Un type d'entités manquant est un type d'entités qui a été modélisé comme un attribut ou un groupe d'attributs dans une ou plusieurs tables dans le schéma physique. Voyons comment peuvent apparaître ces attributs et comment retrouver un type d'entités manquant à l'aide d'une série d'exemples.

**Exemple 1**

Considérons le schéma de la figure 4.16. La figure 4.17 donne une traduction possible de ce schéma en relationnel.

Le type d'entités `UNIVERSITE` n'est pas représenté par une table dans ce schéma. Il est présent en tant qu'attribut composant (`Nom-unif`) de l'identifiant de `FACULTE` et de `BATIMENT`, (c-à-d les tables correspondantes aux T.E. auxquels il était lié), et dans `PROFESSEUR` (qui a comme clé étrangère l'identifiant de `FACULTE`).

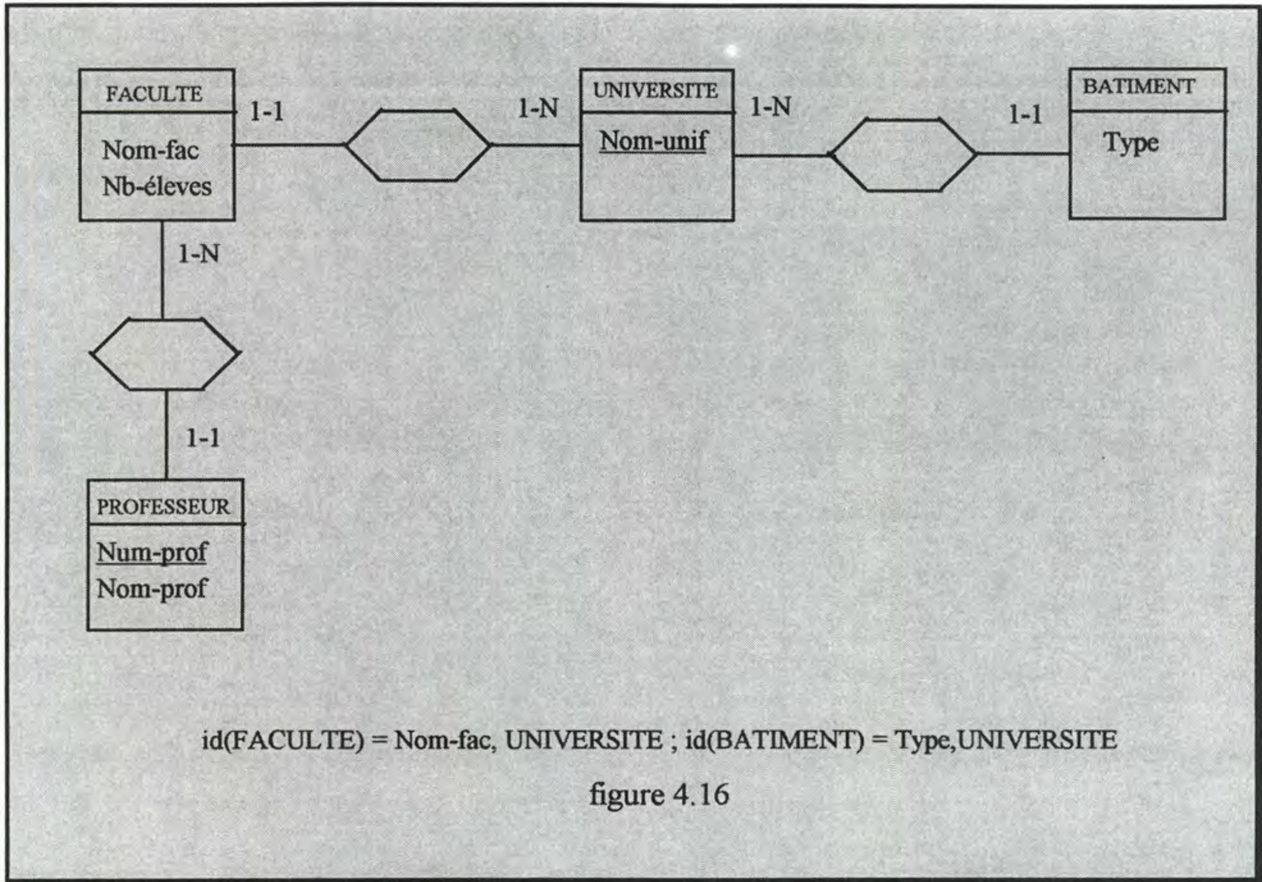
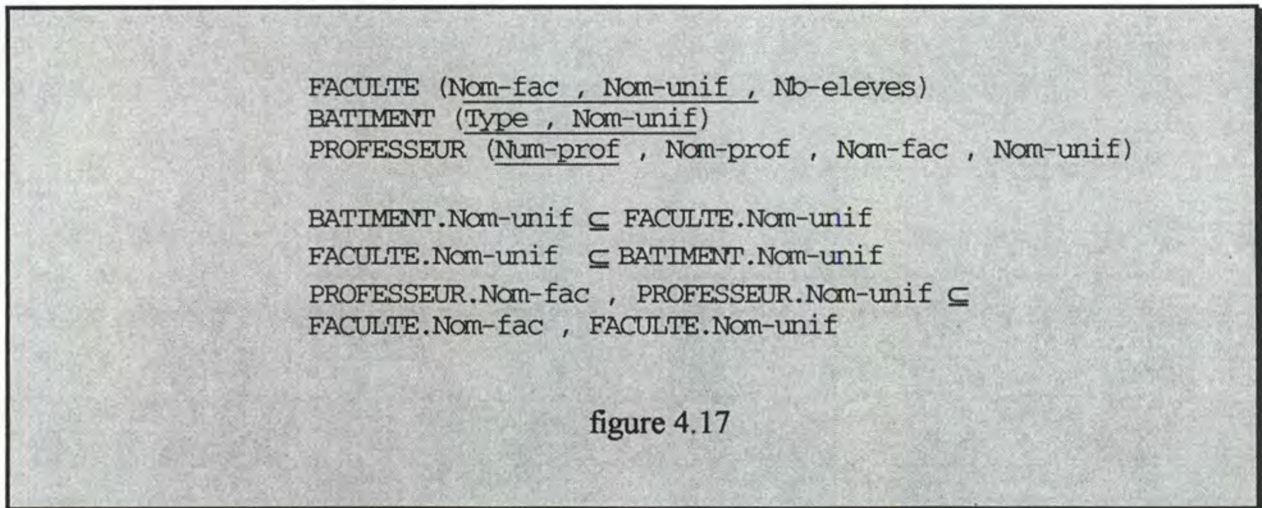



figure 4.16



```
FACULTE (Nom-fac , Nom-unif , Nb-eleves)
BATIMENT (Type , Nom-unif)
PROFESSEUR (Num-prof , Nom-prof , Nom-fac , Nom-unif)

BATIMENT.Nom-unif ⊆ FACULTE.Nom-unif
FACULTE.Nom-unif ⊆ BATIMENT.Nom-unif
PROFESSEUR.Nom-fac , PROFESSEUR.Nom-unif ⊆
FACULTE.Nom-fac , FACULTE.Nom-unif
```

figure 4.17

Nous allons maintenant essayer de revenir au schéma de départ. En appliquant au schéma de la figure 4.17 la recherche des types d'associations one-to-many on obtient le schéma de la figure 4.18.

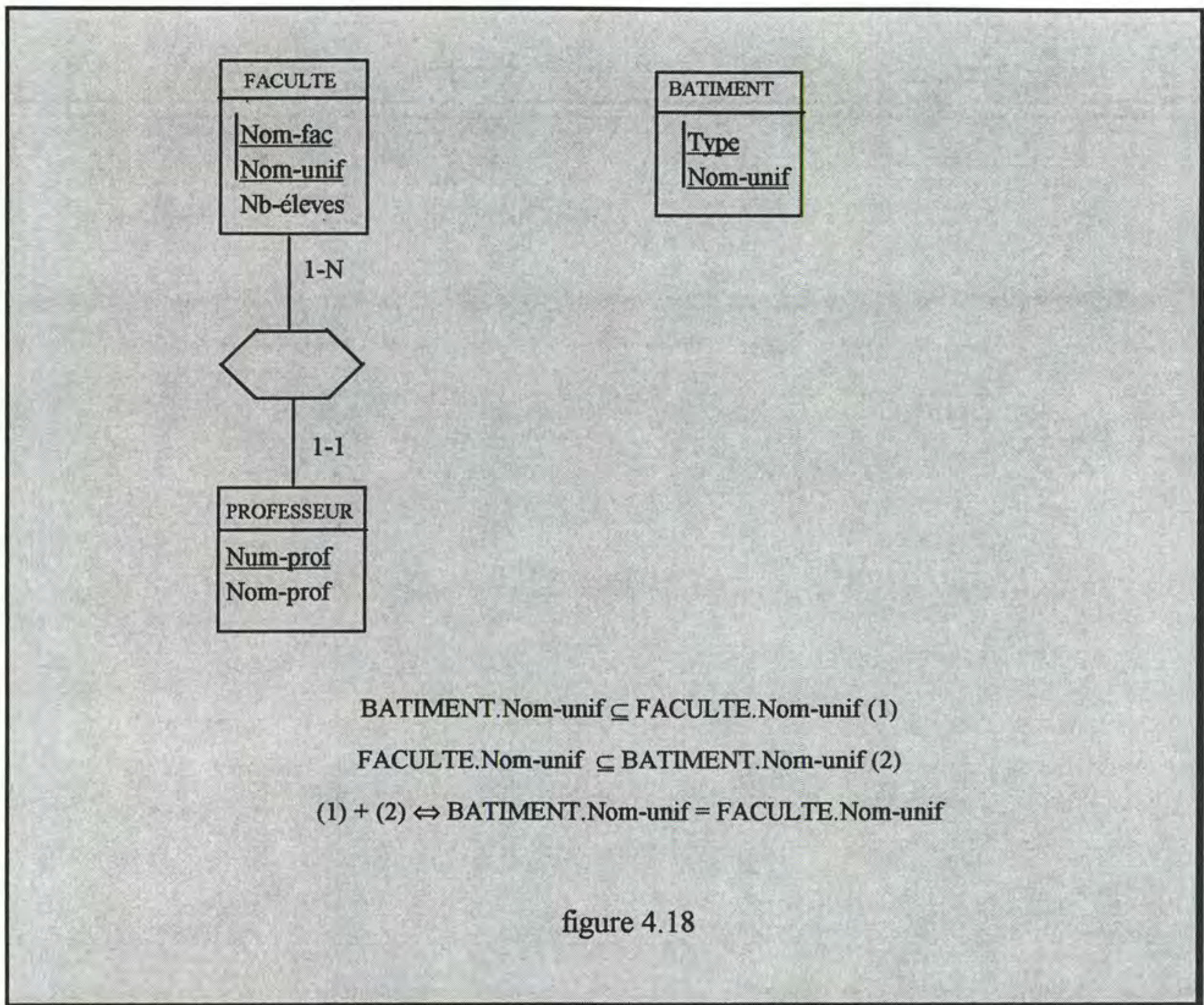



figure 4.18

On remarque dans ce schéma qu'il y a des attributs (`FACULTE.Nom-unif` et `BATIMENT.Nom-unif`) qui font partie d'un identifiant (mais qui ne sont pas des identifiants), et qui sont présents du côté droit d'une contrainte d'inclusion. Il n'est donc pas possible de traduire cela en tant que type d'associations one-to-many, car ces contraintes d'inclusion ne sont pas des contraintes référentielles. Cette forme assez particulière de contrainte fournit un indice pour retrouver un T.E. manquant.

Une fois que l'attribut représentant le T.E. manquant a été retrouvé, il suffit de créer un T.E. dans le schéma conceptuel, comprenant un attribut (il s'agit de l'attribut qui représentait le T.E. manquant). Il est forcément identifiant. Toutes les contraintes d'inclusion dont le côté droit contenait un attribut représentant le T.E. manquant doivent maintenant porter sur l'attribut du nouveau T.E..

On va donc créer un T.E. `UNIVERSITE(Nom-unif)` et les contraintes deviennent :

```
BATIMENT.Nom-unif ⊆ UNIVERSITE.Nom-unif
FACULTE.Nom-unif  ⊆ UNIVERSITE.Nom-unif
```

Elles vont alors être transformées en types d'associations one-to-many. Il reste à retrouver les connectivités exactes. Du côté de FACULTE et de BATIMENT, la connectivité minimale est forçément 1, car ces deux T.E. sont identifiés en partie par Nom-unif. Les connectivités maximales sont 1, à cause des contraintes référentielles ci-dessus. En ce qui concerne les rôles joués par UNIVERSITE, pour les connectivités minimales, on peut considérer qu'au moins une des deux connectivités est 1 (sinon, on aurait probablement eu une table pour UNIVERSITE dans le schéma de départ). Si dans les contraintes sur le schéma de départ on trouve une égalité entre FACULTE.Nom-unif et BATIMENT.Nom-unif (comme c'est le cas dans le schéma ci-dessus), on peut supposer que les deux connectivités minimales sont 1. Si on a une contrainte d'inclusion, le rôle joué par le T.E. se trouvant du côté droit de cette contrainte aura une connectivité minimale 1, et l'autre rôle aura une connectivité minimale 0.

**Exemple 2**

Considérons le schéma de la figure 4.19.

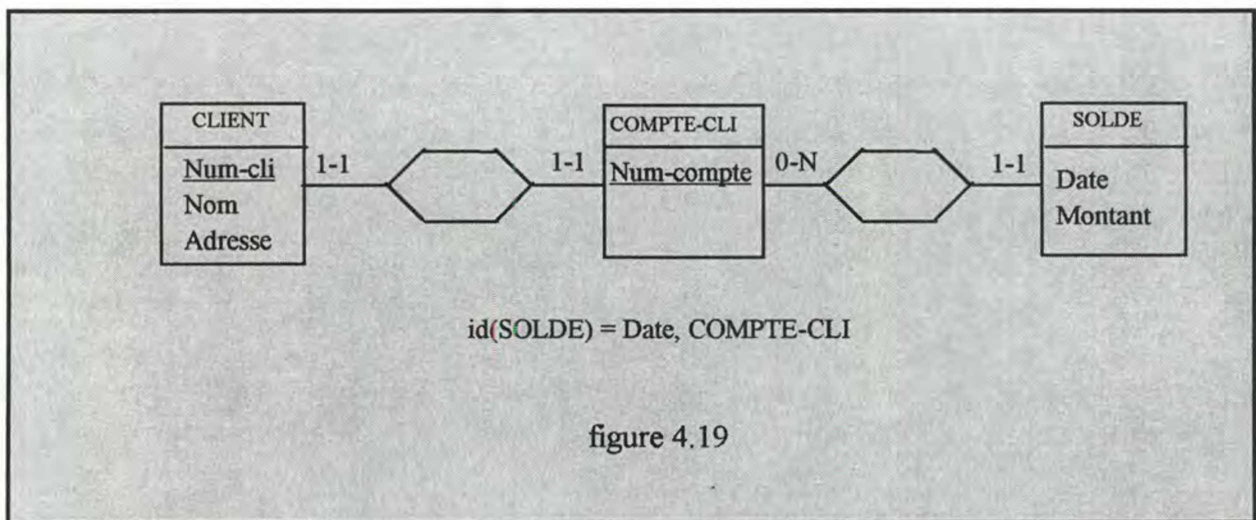
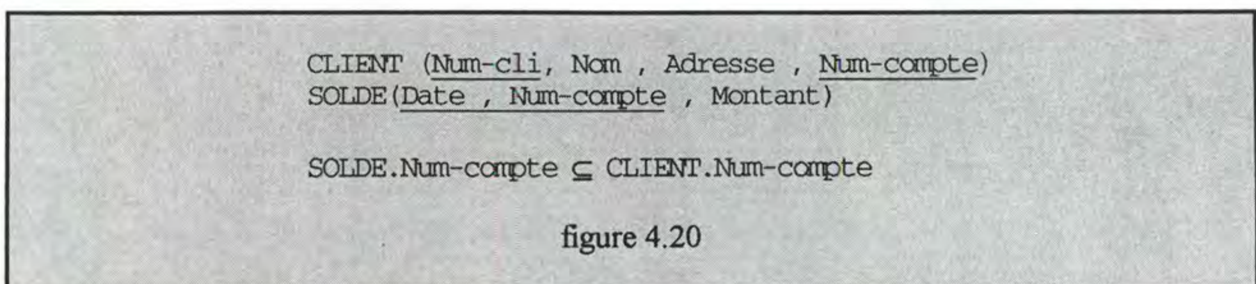



figure 4.19

Une traduction possible de ce schéma en relationnel est donnée à la figure 4.20.



```
CLIENT (Num-cli, Nom , Adresse , Num-compte)
SOLDE(Date , Num-compte , Montant)

SOLDE.Num-compte ⊆ CLIENT.Num-compte
```

figure 4.20

On remarque que COMPTE-CLI a disparu au profit de Num-compte dans CLIENT et SOLDE. On remarque aussi que CLIENT possède deux identifiants : Num-cli, qui était l'identifiant de CLIENT dans le schéma conceptuel, et Num-compte qui joue un rôle d'identification suite à la transformation en relationnel.

Essayons maintenant de retrouver le schéma de départ (figure 4.19). En appliquant à ce schéma la recherche des types d'associations one-to-many, on obtient le schéma de la figure 4.21.

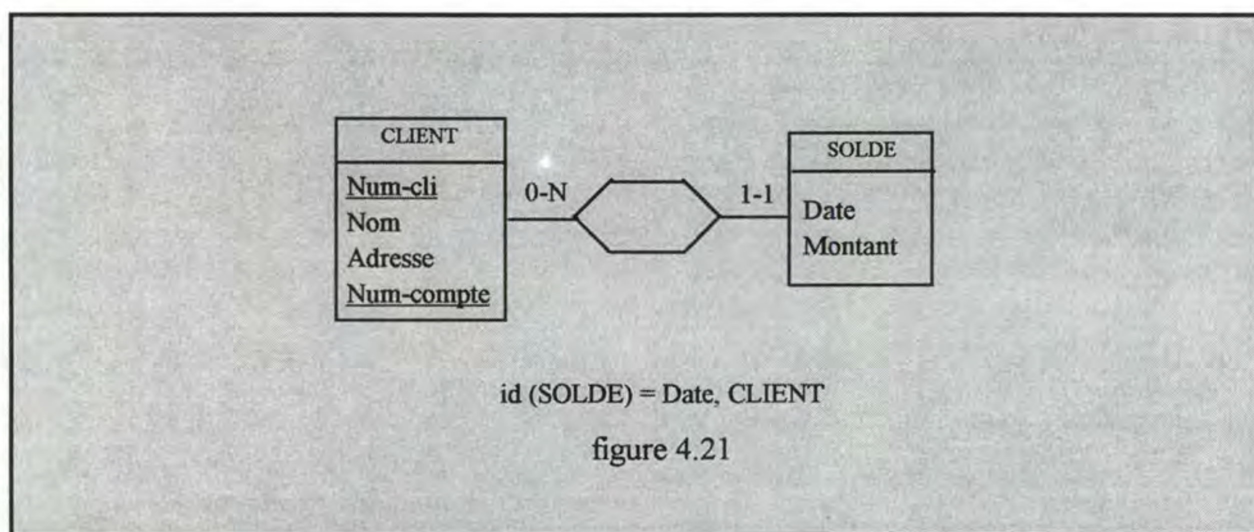


figure 4.21

Ayant retrouvé ce schéma, et sachant que dans le schéma relationnel (figure 4.20) SOLDE.Num-compte faisait partie d'un identifiant et était inclus dans CLIENT.Num-compte on peut se demander si Num-compte ne représente pas un T.E. manquant. On peut par exemple demander l'avis de l'utilisateur.

La façon de retrouver le schéma de départ est proche de celle évoquée à l'exemple 1.

**Exemple 3**

Considérons le schéma de la figure 4.22. Une traduction possible de ce schéma en relationnel est donnée à la figure 4.23.

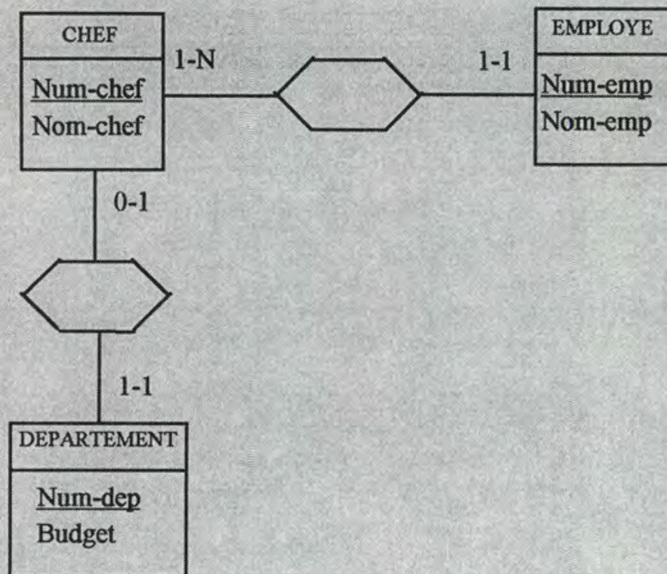



figure 4.22

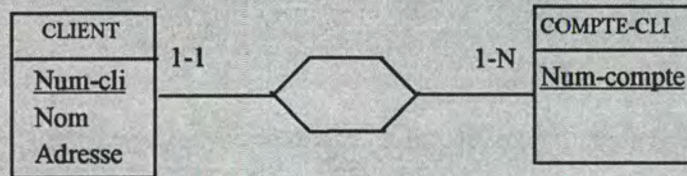
```
DEPARTEMENT (Num-dep, Budget, Nom-chef)
EMPLOYE (Num-emp, Nom-emp, Nom-chef)
DEPARTEMENT.Nom-chef ⊆EMPLOYE.Nom-chef
```

figure 4.23

Puisque `EMPLOYE.Nom-chef` n'est pas identifiant, il ne s'agit pas d'une contrainte référentielle. On a un identifiant qui est inclus dans un attribut non clé. Cela fournit un indice permettant de retrouver le T.E. manquant `Nom-chef`. On peut retrouver le schéma de départ de la même façon qu'à l'exemple 1.

**Exemple 4**

Considérons le schéma de la figure 4.24. Une traduction possible de ce schéma en relationnel est donnée à la figure 4.25.




id (CLIENT) = Num-cli , COMPTE-CLI

figure 4.24

```
CLIENT (Num-cli, Num-compte , nom , adresse)
```

figure 4.25

Ici il n'y plus d'indice pour pouvoir retrouver le T.E. manquant. Seul l'utilisateur peut le repérer. Il faudra s'intéresser particulièrement aux attributs d'identifiants composites.

# 4.3 DE-OPTIMISATION D'UN SCHEMA RELATIONNEL

Lors de la conception d'une base de données, on peut être amené à optimiser le schéma relationnel, pour deux raisons : diminuer la taille des structures de données et diminuer le temps d'accès. Le schéma conceptuel ne doit pas prendre en compte les optimisations : en effet, il décrit la sémantique de la base de données, indépendamment des implémentations possibles.

Il y a 3 techniques principales d'optimisation d'un schéma : la dénormalisation, la redondance structurelle, et la restructuration. Nous allons les développer en montrant d'abord pourquoi et comment on les utilise, et nous proposerons des moyens pour les retrouver.

## *4.3.1 La redondance de dénormalisation*

### 4.3.1.1 Définition

Lors de la conception, il peut arriver que l'on regroupe 2 types d'entités liés par un lien one-to-many en un seul. Cela introduit de la redondance, et fait apparaître des dépendances fonctionnelles dans le schéma conceptuel. Considérons le schéma de la figure 4.26.

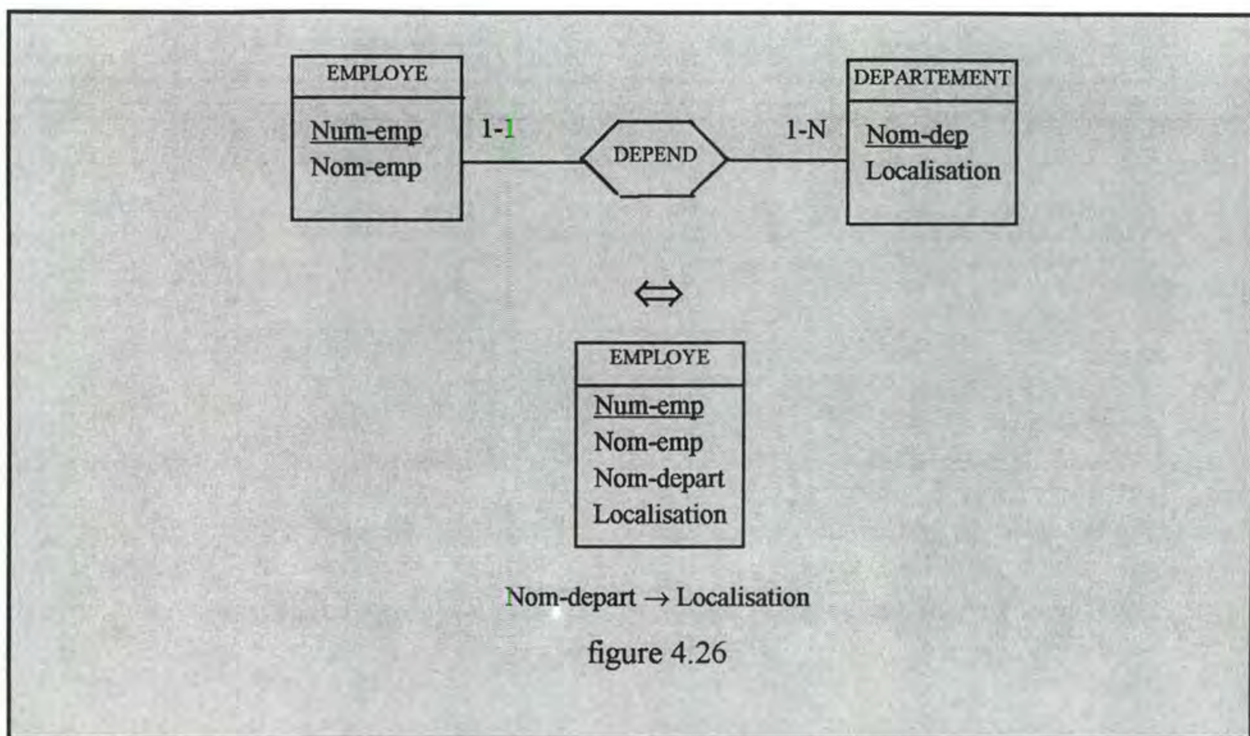


figure 4.26

Sur ce schéma on voit que les types d'entité EMPLOYE et DEPARTEMENT ont été fusionnés. On obtient le type d'entité EMPL, reprenant les attributs de EMPLOYE et de DEPARTEMENT. L'identifiant est Num-emp, c-à-d le même identifiant que EMPLOYE (qui jouait le rôle de connectivité 1-1 dans le schéma de départ). De plus, il y a une dépendance fonctionnelle de nom-depart vers LOCALISATION, exprimant le fait que Nom-depart était identifiant dans DEPARTEMENT.

On pourrait aussi regrouper deux types d'entités reliés par des rôles dont les connectivités minimales sont 0. On aurait alors dans EMPL des lignes correspondant à des employés dans lesquelles les attributs venant de département n'auraient aucun sens sématique (ils correspondraient à des attributs absents), des lignes correspondant à des départements dans lesquelles les attributs venant de EMPLOYE sont inutiles, et des lignes concernant un employé lié à un département (donc des lignes où ces deux types d'attributs seraient nécessaires). Pour implémenter ces absences, on peut utiliser la valeur NULL ou une valeur conventionnelle. Un autre moyen serait d'avoir un attribut flag permettant de savoir à quel type de ligne on a à faire.

Si les deux connectivités maximales étaient 1, on aurait alors deux identifiants dans EMPL (num-emp et nom-depart), et donc il n'y aurait plus de dépendance fontionnelle explicite (il y aura bien entendu les dépendances implicites entre un identifiant et les attributs identifiés par ce dernier). Cette technique s'apparente à la fusion verticale (cf transformations de restructuration, tome1) et n'entraîne aucune dénormalisation.

La dénormalisation que nous venons de voir sera généralement effectuée pour des raisons d'optimisation. En effet, cela permet de ne plus accéder qu'à une seule table. Il peut arriver

aussi que le concepteur d'une base de données ait construit un schéma dénormalisé pour d'autres raisons (par exemple, par suite d'une mauvaise interprétation du réel perçu).

Il est important de normaliser le schéma pour obtenir un schéma décrivant des concepts uniques. Grâce aux DF les tables seront décomposées de façons à ce que chaque concept soit représenté séparément.

On obtient ce résultat en normalisant le schéma en troisième forme normale (3NF), car les transformations pour y arriver garantissent la préservation des données et des DF.

## 4.3.1.2 Normalisation en 3NF

Le point de départ de cette partie est tirée de [HAI89b].

### 4.3.1.2.1 Définitions

**Première forme normale (1NF)**

Une relation est sous première forme normale (1NF) si elle est définie sur des domaines simples (C'est le cas de toute table SQL).

**Deuxième forme normale (2NF)**

Une relation est sous deuxième forme normale (2NF) si elle est sous 1NF et si tout attribut non identifiant dépend strictement de chacun des identifiants stricts

**Troisième forme normale (3NF)**

Une relation est sous troisième forme normale (3NF) si elle est sous 2NF et s'il n'existe aucune dépendance transitive entre un identifiant et un attribut non identifiant via des attributs non identifiants.

### 4.3.1.2.2 Théorème de décomposition

Considérons le schéma relationnel de la figure 4.27.

```
A (a1 , a2 , a3 , a4 )
a3 -> a4
```

figure 4.27

On remarque une dépendance fonctionnelle de a3 vers a4. Pour normaliser, on peut regrouper a3 et a4 dans une table à part, avec a3 comme identifiant.

Ceci revient à appliquer le théorème de décomposition.

```
A(a1, a2, a3, a4)

a3 -> a4

⇔

A'(a1,a2,a3)
A''(a3,a4)

A'[a3] = A''[a3]
```

**Théorème de décomposition**

#### 4.3.1.2.3 Algorithme classique de normalisation en 3NF

1. Recherche des identifiants et des DF.
2. Recherche d'une couverture minimale des DF.
3. Recherche des DF non triviales[11] et dont le déterminant n'est pas un identifiant.
4. Décomposition successive de la relation en se basant sur les DF du point 3 dont le déterminé n'entre dans la composition du déterminant d'aucune DF.

A chaque décomposition, le déterminant va devenir clé étrangère dans l'une des relations, et identifiant référencé par cette clé dans l'autre. On aura en outre une contrainte d'égalité entre ces deux attributs.

[11]Une DF est triviale si lorsqu'on retire un attribut du déterminant, il n'y a plus de DF.

Après la décomposition, il faut modifier certaines contraintes référentielles obtenues lors de l'extraction. Pour montrer cela supposons qu'on ait obtenu après décomposition les tables de la figure 4.28.

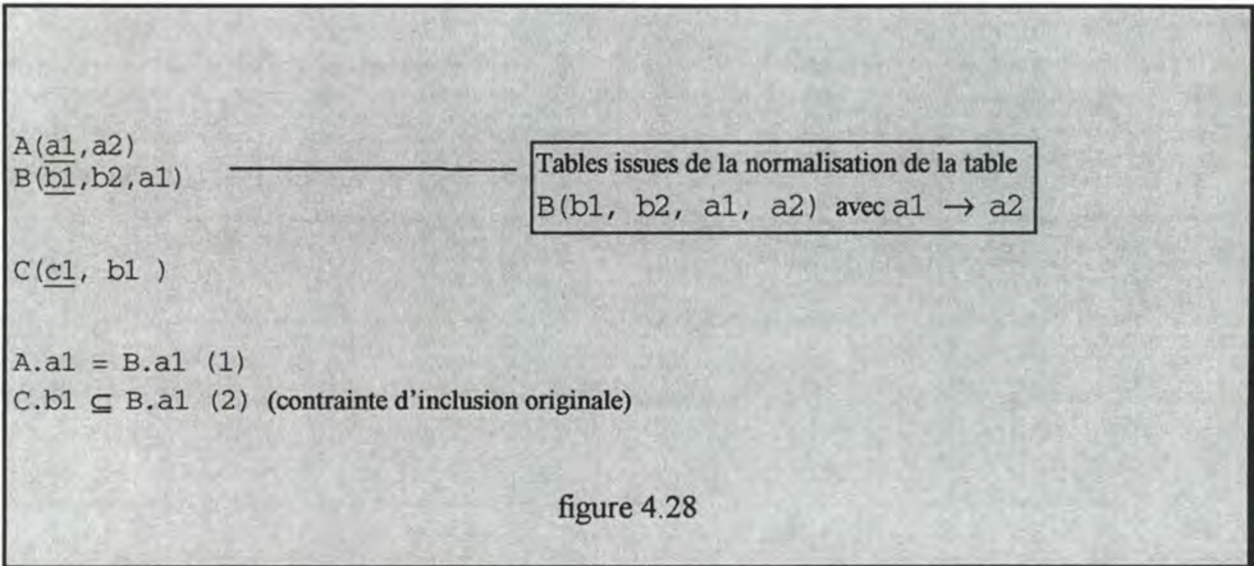



figure 4.28

De (1) et (2) on peut déduire que `C.b1` ⊆ `A.a1` (3). Puisque A.a1 est identifiant cette contrainte représente une contrainte référentielle et doit être conservée. De plus la contrainte (2) peut être supprimée car elle peut être obtenue par transitivité à partir de (1) et (3).

### 4.3.1.2.4 Révision de l'algorithme classique

- Nous avons vu que les contraintes d'égalité donnent lieu à un type d'association one-to-many dont les connectivités minimales sont 1.

  Pourtant comme on l'a vu ci-dessus, il est possible que le concepteur ait fait subir à ses tables des modifications pour pouvoir regrouper des tables liées par des associations dont les connectivités minimales sont 0. Dans ce cas l'algorithme tel qu'il est présenté ci-dessus ne nous nous permettra pas de retrouver le schéma de départ. Voici comment il faut procéder pour retrouver les bonnes connectivités (en supposant qu'on traduit les clé étrangères par des types d'associations).

  On peut examiner l'extension de la façon suivante.

  Considérons le schéma de la figure 4.29, et supposons que `A` et `B` sont deux tables obtenues après la décomposition de `AB (a1, a2, b1, b2)` avec `b1` → `b2`.

```
A(a1 , a2 , b1 )
B(b1 , b2 )
```

figure 4.29

Nous allons distinguer deux hypothèses sur la représentation des attributs absents.

- Supposons tout d'abord que lors de la conception, les attributs absents ont été représenté par une valeur nulle.

  Si les attributs de A (sans A.b1) et de B étaient toujours nuls et non-nuls en même temps dans chaque ligne de AB, alors on a très probablement deux connectivités minimales 1. Si parfois les attributs de A (resp. B) étaient nuls alors que ceux de B (resp. A) étaient présents dans la table de départ, alors on devrait avoir une connectivité minimale 0 du coté de A (resp. B)

- Supposons maintenant que le concepteur a utilisé un ou plusieurs attributs flags pour indiquer a quelle type de ligne on a à faire.

  Il faut examiner chaque ligne d'une façon semblable. D'abord, on retrouve le ou les attributs flags (il devrait s'agir d'attributs admettant peu de valeur distinctes). Ensuite, on détermine la signification de chaque valeur de cet (ces) attribut(s) et on en tire des conclusions sur les connectivités.

## *4.3.2 La redondance structurelle*

La redondance structurelle consiste à ajouter des structures à un schéma de telle façon que leurs instances puissent être dérivées à partir d'instances d'autres structures. Nous allons envisager deux types de redondances : les attributs dérivables et la redondance entre types d'associations.

### 4.3.2.1 Les attributs dérivables

Un attribut dérivable est un attribut présent dans un type d'entités ou dans un type d'associations et qui peut être calculé à partir d'un ou plusieurs autre attributs et/ou de rôles de types d'entités ou de types d'associations. Cette technique peut être utilisée pour des raisons d'optimisations mais aussi pour augmenter la lisibilité et la représentativité du réel perçu d'un schéma.

Plutôt que de faire une étude permettant de retrouver n'importe quel attribut dérivable, nous préférons d'abord analyser une série d'attributs dérivables parmi les plus courants. Nous verrons lors de l'analyse des cas qu'il y a un certain nombre d'invariants. Cela nous permettra de donner en conclusion une série d'heuristiques permettant de retrouver des attributs dérivables. Le premier cas sera traité de façon complète et rigoureuse, et les cas suivants seront discutés plus rapidement.

#### 4.3.2.1.1 Cas 1 : redondance entre un attribut et la somme sur un attribut

**Définition**

Considérons le schéma de la figure 4.30.

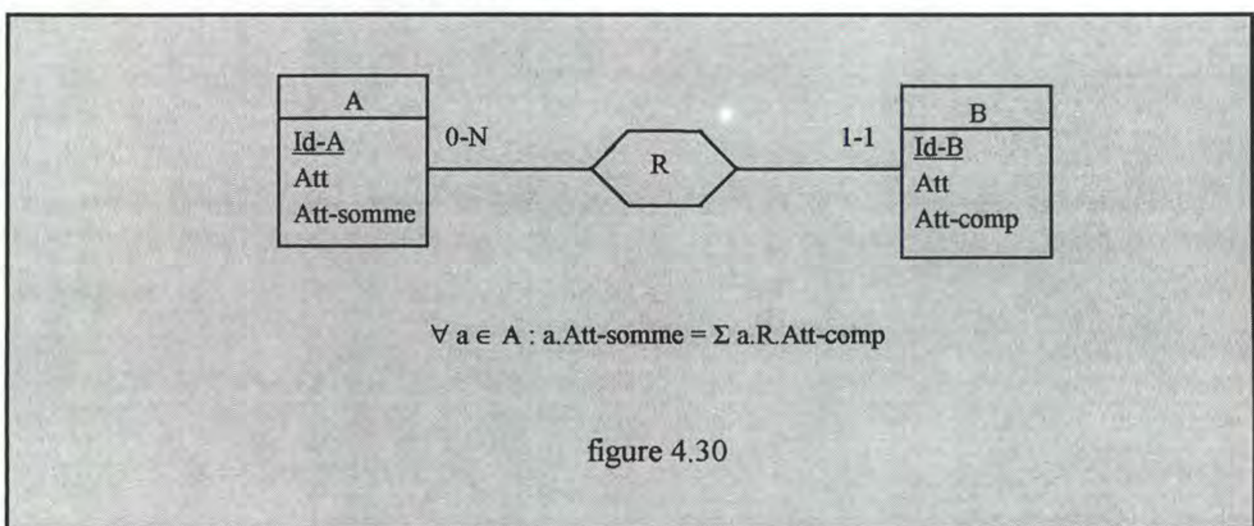



figure 4.30

On voit qu'il y a une contrainte de redondance qui exprime que `a.Att-somme` est redondant avec la somme des valeurs des attributs `Att-comp` des entités `B` associées à l'entité a.

La traduction de ce schéma en un schéma conforme au modèle relationnel est donnée à la figure 4.31. L'opérateur `SELECT` présent dans cette figure permet de sélectionner certaines lignes d'une table. Le premier argument est la table dans laquelle on sélectionne, et le deuxième argument est la condition de sélection.

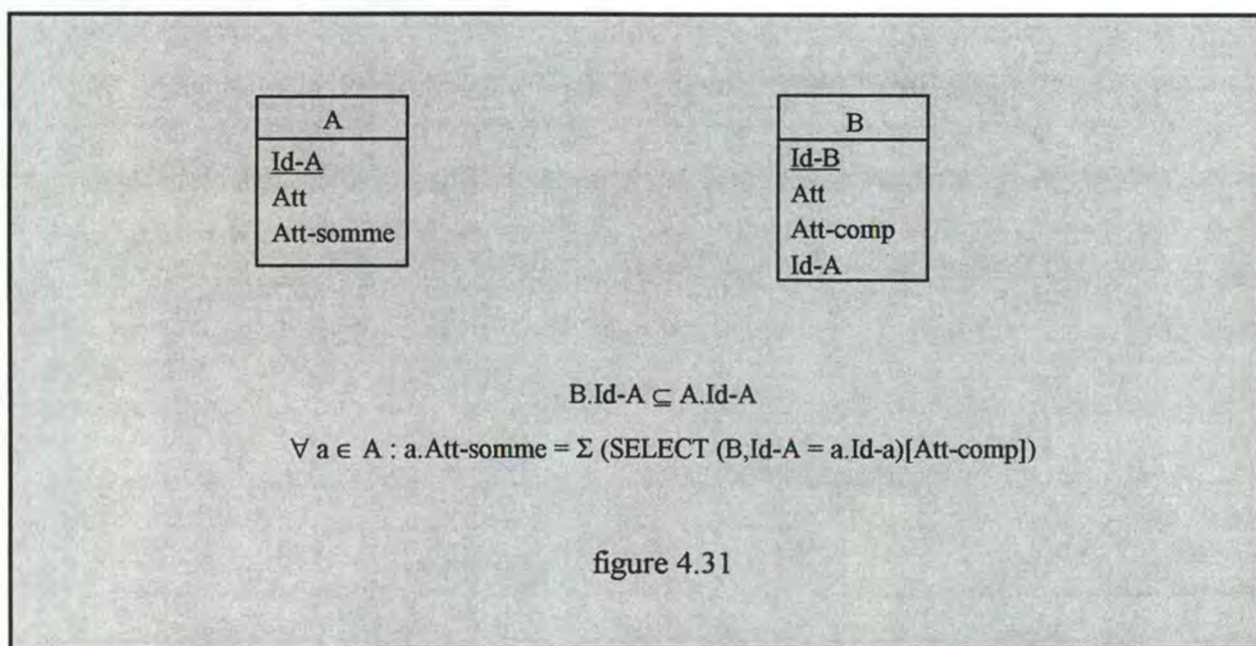



figure 4.31

**Recherche sur les noms**

Il y a de fortes chances pour que l'attribut dérivé porte un nom proche de l'attribut "source" postfixé ou préfixé par des caractères tels que 'SUM', 'SOMME', etc.

**Recherche dans les comportements imposés aux données**

***Définition des UC***

*A) Ajout d'une ligne dans A*

```
INSERT a in A
ou bien PRECOND a[Att-somme] = Σ (SELECT (B , Id-A = a[Id-A])[Att-comp])
ou bien IMPLIES REPLACE a BY a' IN A |
        a'[Att-somme] = Σ(SELECT (B, Id-A = a[Id-A])[Att-comp])
```

Lors d'un ajout dans A, soit on vérifie que l'attribut Att-somme est déjà à la bonne valeur, soit on calcule celle-ci.

*B) Suppression d'une ligne de A*

```
NO PROTOCOL
```

Une suppression d'une ligne de A n'entraîne aucune gestion de la contrainte de redondance

*C) Modification d'une ligne de A*

```
REPLACE a BY a' IN A | a[Att-somme] <> a'[Att-somme]
PRECOND a'[Att-somme] = Σ (SELECT (B,Id-A = a'[Id-A])[Att-comp])
```

Lors d'une modification de l'attribut Att-somme d'une ligne de A, on vérifie qu'il a la bonne valeur (c-à-d celle qui respecte la contrainte de redondance).

*E) Ajout d'une ligne dans B*

```
INSERT b IN B
IMPLIES REPLACE a | a[Id-A] = b[Id-A] BY a' IN A |
        a'[Att-somme] = Σ (SELECT(B , Id-A = a[Id-A])[Att-comp])
```

Un ajout d'une ligne dans B entraîne la mise à jour des attributs dérivables des lignes de A liées à la ligne de B modifiée.

*D) Supression d'une ligne de B*

```
DELETE b FROM B
IMPLIES  REPLACE a | a[Id-A] = b[Id-A] BY a' in A |
         a'[Att-somme] = Σ (SELECT(B, Id-A = a[Id-a])[Att-comp])
```

L'attribut Att-somme de a ∈ A est recalculé suite à la suppression d'une ligne de B reliée à a.

*F) Modification d'une ligne de B*

```
REPLACE b by b' in B | b[Att-comp] <> b'[Att-comp]
IMPLIES REPLACE a | a[Id-A] = b'[Id-a] BY a' IN A |
        a'[Att-somme] = Σ(SELECT(B , Id-a = a[id-a])[att-comp])
```

Lorsqu'on modife l'attribut composant de la somme dans une ligne de B, on modifie l'attribut Att-somme correspondant.

Cette UC ne considère que le changement de Att-comp. Une UC qui considérerait le changement de la clé étrangère Id-A comprendrait un recalcul de l'attribut Att-somme correspondant à l'ancienne ligne reliée et un recalcul de l'attribut Att-somme correspondant à la nouvelle ligne reliée.

### *Recherche dans les triggers*

Nous n'allons envisager que les triggers concernant l'ajout dans A.

Le trigger implémentant la précondition sera de la forme

```
CREATE TRIGGER nom-trigger
ON  A
FOR INSERT
AS IF (SELECT Att-somme
        FROM inserted)
       <>
       (SELECT SUM(Att-comp)
        FROM B , inserted
        WHERE B.Id-A = inserted.Id-A)

BEGIN
    gestion-erreur
 END
```

Le trigger implémentant l'IMPLIES sera de la forme

```
CREATE TRIGGER nom-trigger
ON A
FOR INSERT
AS

BEGIN

      UPDATE A SET Att-somme = (SELECT SUM(Att-comp)
                                 FROM B
                                 WHERE B.Id-A = inserted.Id-A)
               FROM inserted
               WHERE A.Id-a = inserted.Id-A

END
```

### *Recherche dans les requêtes*

*A) Ajout d'une ligne dans A*

Supposons que l'on insère dans A la ligne (var-Id-A , var-Att , var-Att-somme).

La précondition pourrait être implémentée comme suit.

Avant l'ajout, on vérifiera que l'attribut dérivable est à la bonne valeur. Pour cela, on fera d'abord la requête :

```
EXEC SQL

        SELECT SUM(Att-somme) INTO :var-somme
        FROM B
        WHERE B.Id-A = :var-Id-A

END-EXEC
```

suivie d'une comparaison avec `var-att-somme`.

En ce qui concerne l'`IMPLIES`, il y a de fortes chances qu'on retrouve avant `l'INSERT` un appel à une ou plusieurs requêtes SQL qui vont calculer la valeur de l'attribut dérivable.

```
EXEC SQL

        SELECT  SUM(Att-comp) INTO :var-somme
        FROM B
        WHERE Id-A=:var-Id-A

END-EXEC
```

On aura ensuite l'insertion dans A

```
EXEC SQL

        INSERT INTO A VALUES (:var-Id-A,...,:var-somme)

END-EXEC
```

Remarque

C'est le même principe pour une modifcation d'une ligne de `A`.

*B) Suppression d'une ligne de B*

Deux stratégies sont possibles pour implémenter cette UC.

- soit calculer la nouvelle valeur de l'attribut dérivé par un calcul incrémental[12];
- soit calculer la nouvelle valeur de l'attribut dérivé par un calcul complet.

---

[12]Nous appelons calcul incrémental un calcul qui utilise l'ancienne valeur de l'attribut dérivé pour trouver la nouvelle valeur.

Le calcul incrémental est plus rapide. On peut donc penser que ce moyen de gestion de la contrainte sera plus souvent choisi. Nous allons le développer.

D'abord on aura une requête sur la ligne à modifier dans B recherchant la valeur de la clé étrangère et l'ancienne valeur de l'attribut Att-comp. Ensuite, on aura une recheche de l'ancienne valeur de l'attribut dérivé dans A, suivie d'un opération dans le programme d'application calculant la nouvelle valeur de Att-somme. On aura enfin la modification de l'attribut dérivé et la suppression dans B.

Supposons que l'on supprime l'élément de B identifié par var-Id-B.

On aura des déclarations successives semblables à celles ci-dessous:

(1.1)

```
EXEC SQL

        SELECT B.Id-A, B.Att-comp INTO :var-Id-A, :var-comp
        FROM B WHERE Id-B = :var-Id-B

END-EXEC
```

(1.2)

```
EXEC-SQL

        SELECT Att-somme INTO :var-old-somme
        FROM A WHERE Id-A = :var-Id-A

end-exec
```

(1.3)

```
var-new-somme = var-old-somme  - var-comp
```

(1.4)

```
EXEC SQL

        UPDATE A SET Att-somme = :var-new-somme
        WHERE Id-A = :var-Id-A

END-EXEC
```

(1.5)

```
EXEC SQL

        DELETE from B
        WHERE Id-B = :var-Id-B

END-EXEC
```

Remarques

- (1.5) n'étant pas influencé (et n'influencant pas) par (1.4), (1.3), (1.2) il pourrait également se situer entre ces parties là.
- Les autres parties étant dépendantes les unes des autres, on ne peut pas les permuter.
- C'est le même principe pour l'ajout et la modification dans `B`.

**Généralisation**

On peut généraliser le cas présenté ci-dessus à une redondance entre un attribut d'une table et la somme sur n'importe quelles opérations concernant un ou plusieurs attributs d'une autre table associée. L'opération peut être par exemple la multiplication (figure 4.32).

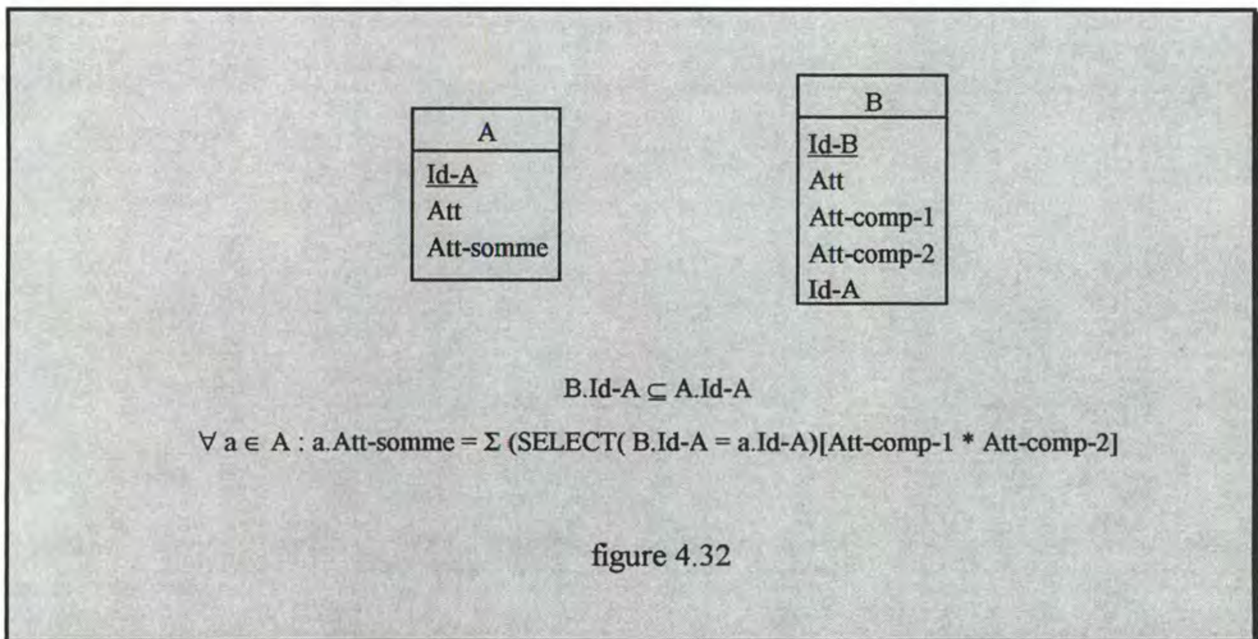



figure 4.32

En fait les principaux changements par rapport au cas décrit ci-dessus vont se situer aux endroits où il y avait un `SELECT` avec un `SUM(Att-comp)`. Il devront être remplacés par un `SUM (opération(s) sur attributs de R2)`.

Par exemple, le trigger de vérification pour une insertion dans A s'écrirait comme suit :

```
CREATE TRIGGER nom-trigger
ON A
FOR INSERT
AS IF
       (SELECT Att-somme
        FROM inserted)
       <>
       (SELECT SUM(Att-comp-1 * Att-comp-2)
        FROM B
        WHERE Id-A = inserted.Id-A)

BEGIN
   gestion-erreur
END
```

En plus de ces changements, les calculs effectués dans le programme d'application pour obtenir la valeur de l'attribut dérivé par une méthode incrémentale devront être modifiés en fonction de l'(des) opération(s) sur les attributs sources

### 4.3.2.1.2 Cas 2 : redondance rôle / attribut

**Définition**

Considérons le schéma de la figure 4.33.

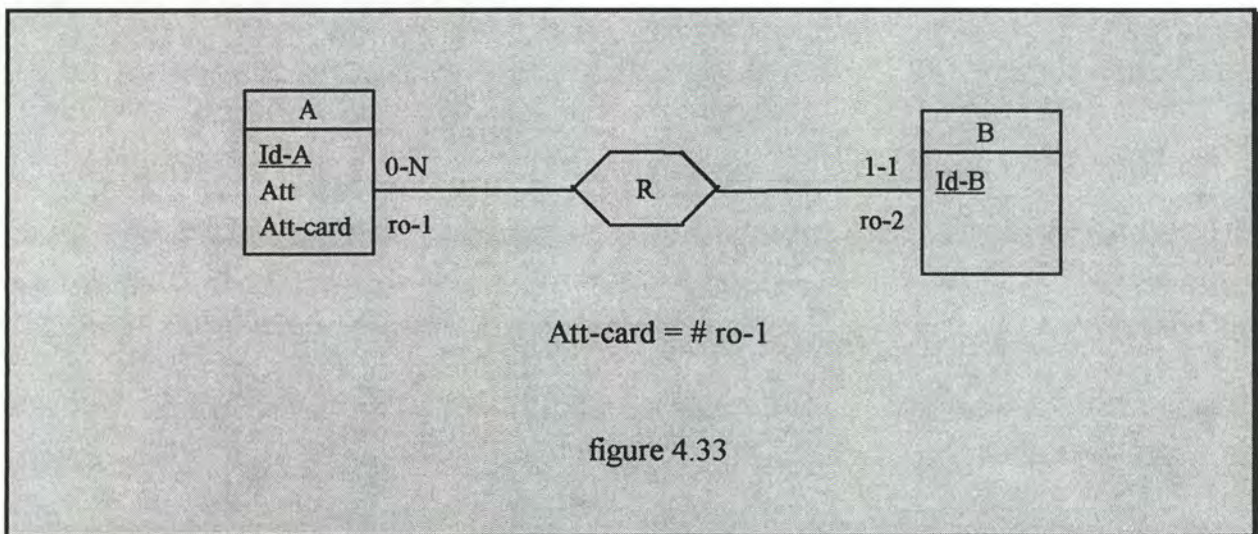



figure 4.33

`Att-card` représente le nombre de `b` ∈ `B` associés à un `a` ∈ `A` . On obtient ainsi un accès rapide à la cardinalité de ro-1.

**Recherche**

La traduction du schéma de la figure 4.33 en structures conformes au relationnel est donnée à la figure 4.34.

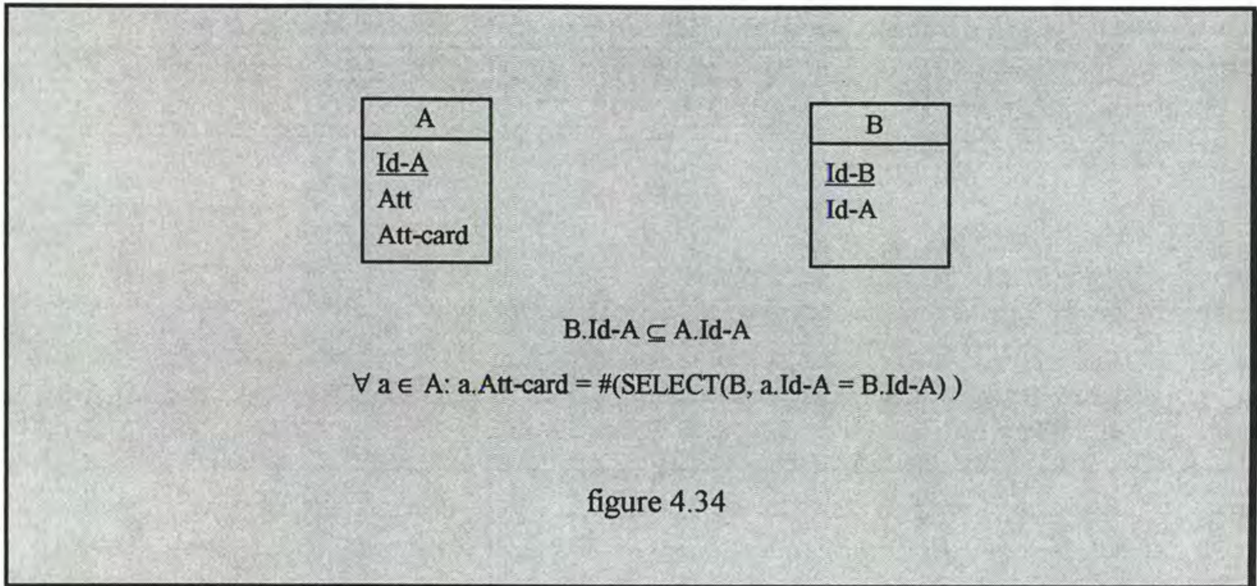



figure 4.34

Voyons l'un des triggers de gestion de cette redondance. il s'agit du trigger de vérification pour l'ajout d'une ligne dans A.

```
CREATE TRIGGER nom-trigger
ON  A
FOR INSERT
AS IF
       (SELECT Att-card
        FROM inserted)
       <>
       (SELECT COUNT(*)
        FROM B , inserted
        WHERE B.Id-A = inserted.Id-A)

BEGIN
    gestion-erreur
 END
```

Par rapport au cas 1, le `SUM` est devenu un `COUNT(*)`. Il en va de même pour les autres triggers et pour les constructions du programme d'application.

En ce qui concerne les calculs incrémentaux il n'y aura evidemment plus besoin d'aller rechercher l'ancienne valeur de `Att-comp`, ni de traiter le cas d'une modification dans `B`. Les calculs effectués dans le programme d'application se résumeront à des incrémentations ou des décrémentations de 1.

On peut également retrouver ce genre d'attribut dérivé en analysant les noms. Il faut être attentif aux libellés comportant 'card', 'nombre', 'number', 'role', '#', etc. Si de plus, on retrouve le nom de l'autre table, l'heuristique est plus forte.

#### 4.3.2.1.3 Cas 3 : redondance entre un attribut d'une table et un attribut d'une autre table

**Définition**

Il peut parfois être utile de recopier un attribut dans un autre type d'entités, afin d'y accéder en une seule fois. Pour pouvoir faire cela, il faut un type d'associations one-to-many entre les deux types d'entités. Dans le schéma de la figure 4.35, on a recopié `Att-source` dans `B`.

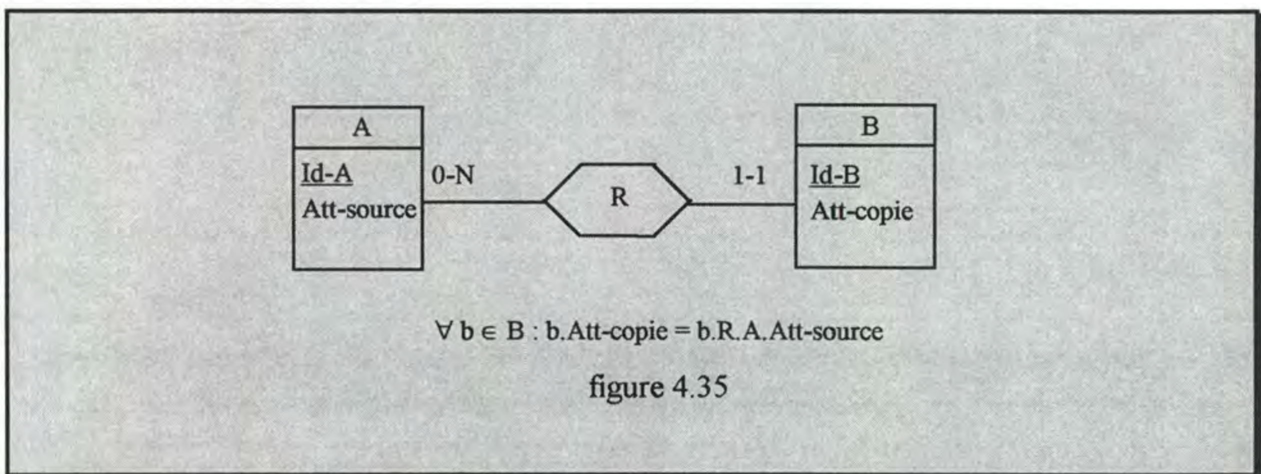


figure 4.35

La traduction du schéma de la figure 4.35 en schéma conforme au modèle relationnel est donnée à la figure 4.36.

| A |
| --- |
| Id-A<br>Att-source |

| B |
| --- |
| Id-B<br>Id-A<br>Att-copie |

$$B.Id\text{-}A \subseteq A.Id\text{-}B$$

$$\forall\ b \in B: b.Att\text{-}copie = SELECT\ (A\ ,\ Id\text{-}A = b.Id\text{-}A)\ [Att\text{-}source]$$

$$B.Id\text{-}A \rightarrow A.Att\text{-}copie$$

figure 4.36

Ce schéma nous donne un premier indice de la présence de la redondance. En effet, on trouve une DF dont le déterminant est une clé étrangère[13].

**Recherche sur les noms**

Il y a de fortes chances pour que l'attribut dérivé porte un nom proche de l'attribut source postfixé ou préfixé par des caractères tels que 'copie', etc.

**Recherche dans les comportements imposés aux données**

***Définition des UC***

*A) Ajout d'une ligne dans A*

```
NO PROTOCOL
```

*B) Suppression d'une ligne de A*

```
DELETE a FROM A
IMPLIES REPLACE b IN B | b[a1] = a[a1] BY b' | b'[Att-copie] = NULL
```

Lorsqu'on supprime une ligne de `A`, on va affecter la valeur nulle à l'attribut `Att-copie` des lignes de B reliées.

[13]Cet indice ne peut bien entendu être retrouvé que sur un schéma non normalisé.

*C) Modification d'une ligne de A*

Supposons que l'on ne modifie que l'attribut `Att-source` de A.

```
REPLACE a BY a' IN A
IMPLIES REPLACE b IN B | b[a1] = a'[a1] BY b' | b'[Att-copie] = a'[Att-source]
```

Une modification de l'attribut `Att-source` d'une ligne de `A` donne lieu à une modification de l'attribut `Att-copie` des lignes de `B` reliées.

*D) Ajout d'une ligne dans B*

```
INSERT b IN B
ou bien PRECOND (∃ a ∈ A | a[a1] = b[a1]) ⇒ a[Att-source] = b[Att-copie]
ou bien IMPLIES REPLACE b | ( ∃ a ∈ A | a[a1] = b[a1])
                BY b' IN B | b'[Att-copie] = a[Att-source]
```

Quand on ajoute une ligne dans `B`, soit on vérifie que l'attribut `Att-copie` a la bonne valeur, soit on affecte la bonne valeur à cet attribut.

*E) Suppression d'une ligne de B*

```
NO PROTOCOL
```

*F) Modification d'une ligne de B*

Supposons que l'on modifie l'attribut `Att-copie` de la ligne de `B`. On aura :

```
REPLACE b BY b' IN B
PRECOND (∃ a ∈ A | a[a1] = b'[a1]) ⇒ b'[Att-copie] = a'[Att-source]
```

***Recherche dans les triggers***

Examinons deux exemples de triggers.

Le trigger implémentant l'`IMPLIES` pour une suppression dans `A` sera de la forme :

```
CREATE TRIGGER nom-trigger
ON A
FOR DELETE
AS UPDATE B
   SET Att-copie = NULL
   FROM deleted
   WHERE B.Id-A = deleted.Id-A
```

Le trigger implémentant la précondition pour un ajout dans B sera du genre :

```
CREATE TRIGGER nom-trigger
ON B
FOR INSERT
AS IF
       (SELECT Att-copie
        FROM inserted)
       <>
       (SELECT Att-source
        FROM A , inserted
        WHERE A.Id-A = inserted.Id-A)
BEGIN
      gestion-erreur
END
```

***Recherche dans les requêtes du programme d'application***

Voyons le cas de la modification d'une ligne de A. Soit la modification de la ligne de A identifiée par `var-Id-A`. Supposons que l'on remplace l'attribut `Att-source` par `var-att-source`. L'`IMPLIES` sera implémenté par les requêtes suivantes.

On aura la modification dans `A` :

```
EXEC SQL

       UPDATE A
       SET Att-source = :var-Att-source
       WHERE Id-A = :var-Id-A

END-EXEC
```

Et la mise à jour dans `B` :

```
EXEC SQL

     UPDATE B
     SET Att-copie = :var-Att-source
     WHERE Id-A = :var-Id-A

END-EXEC
```

Les deux requêtes peuvent être permutées.

Examinons maintenant les requêtes pour une modification dans B. Supposons que l'on modifie la ligne de B identifiée par :var-b1, et que l'on remplace Att-copie par var-new-Att-copie. La vérification sera effectuée de la façon suivante :

On ira d'abord chercher la valeur de clé étrangère pour la ligne de B modifiée :

```
EXEC SQL
       SELECT a1 INTO :var-a1
       FROM B
       WHERE b1 = :var-b1
END-EXEC
```

Ensuite, on vérifiera que :var-new-Att-copie a la bonne valeur :

```
EXEC SQL
       SELECT *
       FROM A
       WHERE a1 = :var-a1 AND Att-source = :var-new-Att-copie
END-EXEC
```

Si la requête ne donne rien, il y aura une gestion d'erreur. Sinon, on fera la modification dans B.

## 4.3.2.1.4 Cas 4 : redondance entre attributs d'une même table

**Définition**

Considérons le schéma de la figure 4.37.

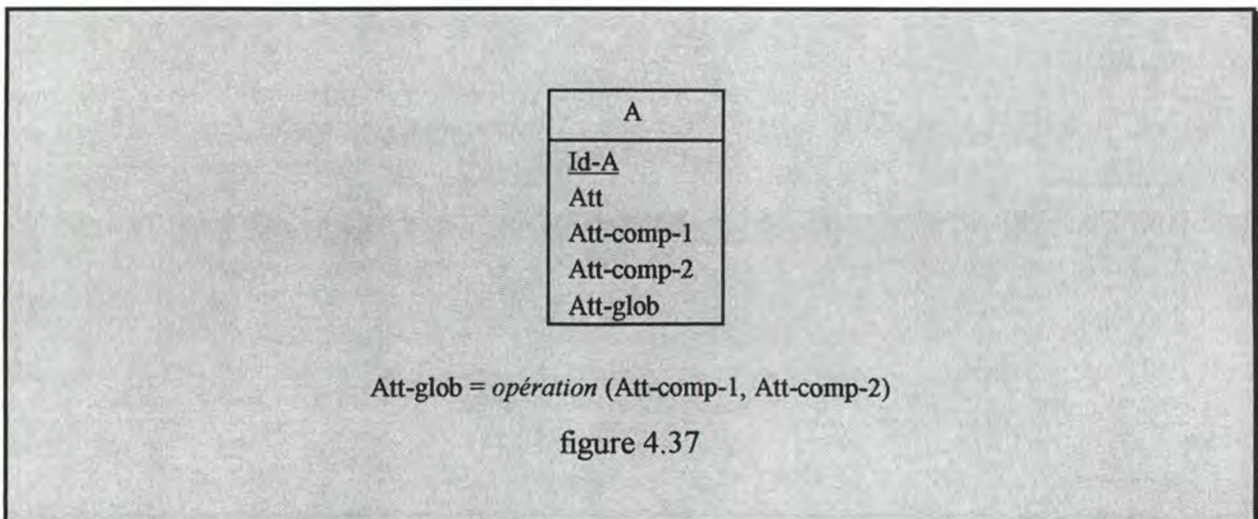


figure 4.37

Remarques

- Une opération est pris au sens large, c-à-d qu'elle comprend toutes les fonctions réalisables sur des valeurs en SQL.
- Il est facile de généraliser ce qui suit à plus de deux attributs composants.

Dans la suite des recherches, nous considérerons que l'opération est une somme.

**Recherche dans les checks**

On pourrait trouver un check de la forme

```
CHECK (Att-glob = Att-comp-1 + Att-comp-2)
```

**Recherche dans les comportements imposés aux données**

Nous ne considérerons que les UC.

*A) Ajout d'une ligne dans A*

```
INSERT a IN A
ou bien PRECOND a[Att-glob] = a[Att-comp-1] + a[Att-comp-2]
ou bien IMPLIES REPLACE a BY a' IN A | a'[Att-glob] = a[Att-comp-1] +
                                                      a[Att-comp-2]
```

Lorsqu'on ajoute une ligne dans A, soit on vérifie que Att-glob est à la bonne valeur, soit on lui affecte cette valeur.

*B) Suppression d'une ligne de A*

```
NO PROTOCOL
```

*C) Modification d'une ligne de A*

Considérons d'abord une modification de Att-comp-1 (on aura le même type d'UC pour Att-comp-2).

```
REPLACE a BY a' IN A | a[Att-comp-1] <> a'[Att-comp-1]
ou bien PRECOND a[Att-glob] = a'[Att-comp-1] + a'[Att-comp-2]
ou bien IMPLIES REPLACE a BY a' IN A | a'[Att-glob] = a'[Att-comp-1] +
                                                      a'[Att-comp-2]
```

Considérons maintenant une modification de `Att-glob`.

```
REPLACE a BY a' IN A | a[Att-glob] <> a'[Att-glob]
PRECOND a[Att-glob] = a'[Att-comp-1] + a'[Att-comp-2]
```

#### 4.3.2.1.5 Conclusion

Nous avons présenté un ensemble de moyens à la disposition du programmeur pour faire respecter une contrainte d'intégrité liée à un attribut dérivable. Par une série de cas particuliers, nous avons exposé une série de constructions type que l'on pourrait retrouver pour la gestion de tels attributs.

Il existe beaucoup d'autres types d'attributs dérivables. Ceux que nous avons développé ci-dessus nous semblent être les plus fréquents.

Il existe bien entendu beaucoup d'autres techniques pour gérer un attribut dérivable que celles présentées ci-dessus. De plus, même si nous nous limitons aux techniques vues ici, retrouver un attribut dérivable n'est pas une tâche aisée.

Pour simplifier le problème nous allons, à partir des exemples présentés ci-dessus, essayer de dégager une série d'heuristiques générales permettant de retrouver un attribut dérivable entre tables disctinctes.

1. Concernant les triggers de vérification :

   Les triggers portant sur la table contenant l'attribut dérivé ne concerneront que l'ajout ou la modification sur cette table. Ces triggers vérifient que l'un des attributs insérés ou modifiés est à la bonne valeur. L'utilisation de `SUM`, `COUNT`, etc. est un indice supplémentaire.

   Si on a implémenté un (des) trigger(s) de vérification sur la table source, il devra aller verifier si l'attribut dérivé a déjà sa bonne valeur.

2. Concernant les triggers d'action :

   On examinera les triggers qui, pour l'insertion d'un élément dans une table, mettent à jour à chaque fois l'un des attributs (non clé) insérés. Si tel est le cas, cet attribut est peut-être dérivé. On examinera aussi les triggers qui, pour un ajout, une suppression et/ou une modification, mettent à jour l'un des attributs d'une autre table.

3. Concernant les requêtes du programme d'application :

   Si une opération d'ajout est précédée d'un calcul de la future valeur d'un des attributs non-clés insérés, alors cet attribut est peut-être redondant.

   On examinera aussi si la suppression, l'ajout ou la modification d'une table est (presque) toujours faite de pair avec une modification sur un attribut (non clé) d'une autre table.

4. Les noms des attributs dérivables sont souvents évocateurs ('Somme', 'Copie', etc.).

## 4.3.2.2 La redondance entre types d'associations

Nous allons envisager la redondance entre types d'associations one-to-many.

Un T.A. one-to-many peut parfois être redondant avec la composition d'autres T.A. one-to-many. Le schéma de la figure 4.38 illustre cette situation.

Le T.A. RAC est redondant avec la composition de RAB et RBC. Dans le schéma conforme au modèle relationnel, cela se traduira par une clé étrangère redondante dans A. En effet, on y trouvera une clé étrangère référencant C. Ce genre de redondance permet d'établir un lien direct entre deux tables éloignées reliées via d'autres tables par un lien one-to-many.

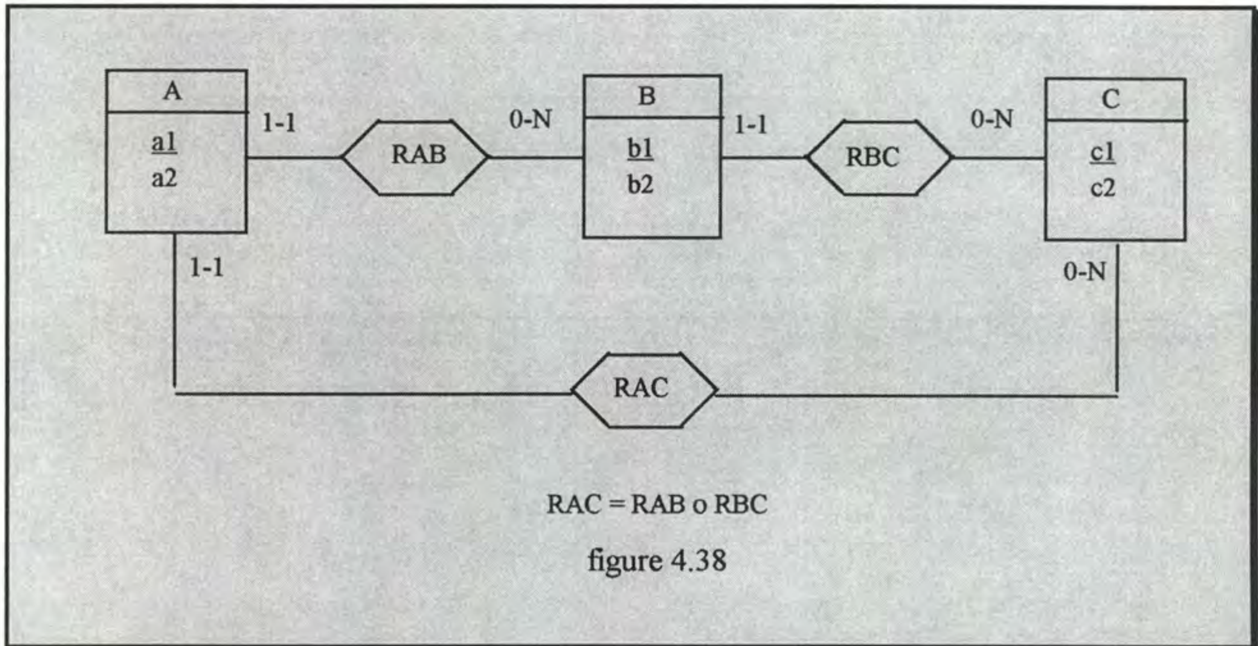



figure 4.38

### 4.3.2.2.1 Recherche basée sur les connectivités

Considérons le schéma conceptuel de la figure 4.39.

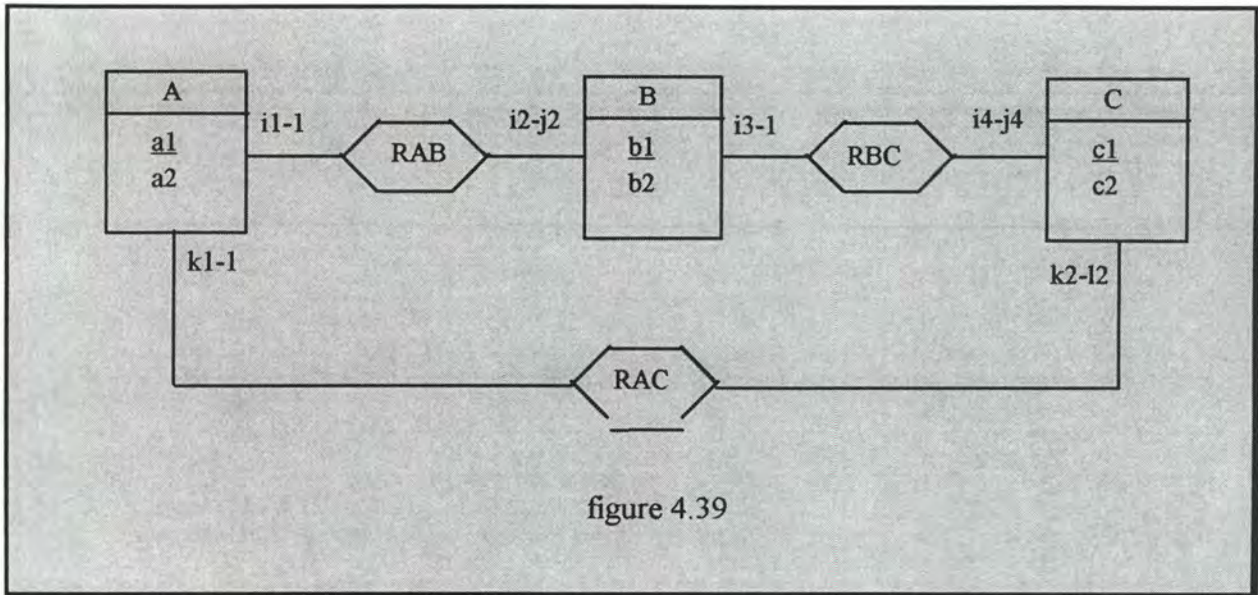



figure 4.39

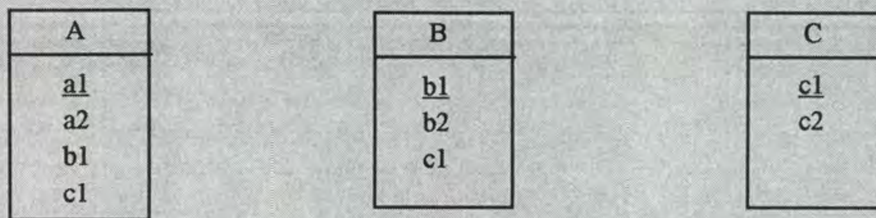
Si

$$k1 = \min(i1, i3)$$
$$k2 = \min(i2, i4)$$
$$l2 = \max(j2, j4)$$

alors, il y a une chance pour que RAC soit redondant avec la composition RAB et RBC.

Cette recherche ne donne pas un niveau de certitude élevé. En effet, si on trouve les connectivités décrites ci-dessus, il n'est pas certain qu'il y a redondance (en effet, on a pu ajouter un T.A. pour des raisons sémantiques), et de plus, il sera parfois impossible de déterminer quelle est le T.A. candidat redondant (sans l'avis de l'utilisateur).

Pour la suite des recherches, nous considérerons la traduction en schéma conforme au modèle relationnel du schéma de la figure 4.38. Cette traduction est donnée à la figure 4.40.




A.b1 ⊆ B.b1

B.c1 ⊆ C.c1

A.c1 ⊆ C.c1

A.c1 redondante avec la composition de A.b1 et B.c1

figure 4.40

#### 4.3.2.2.2 Recherche dans les comportements imposés aux données.

Remarque

Dans le schéma conceptuel de départ (figure 4.38) les connectivités minimales de `A-RAB`, `B-RBC` et `A-RAC` sont 1. Si elle étaient 0, les recherches dans les comportements imposés aux données seraient plus compliquées. En effet, lors chaque vérification ou action, il faudrait vérifier qu'il y a bien un lien entre les tables. Nous avons préféré simplifier et nous concentrer uniquement sur la gestion de la redondance.

**Définition des UC**

*A) Ajout d'une ligne dans A*

```
INSERT a IN A
ou bien PRECOND ∃ b ∈ B | a[b1] = b[b1] AND b[c1] = a[c1]
ou bien IMPLIES REPLACE a | (∃ b ∈ B | a[b1] = b[b1])
                    BY a' IN A| a'[c1] = b[c1]
```

Lorsqu'on ajoute une ligne dans `A`, soit on vérifie que `c1` a la bonne valeur, soit on lui affecte la bonne valeur. Par "bonne valeur" nous entendons la valeur qui représente le même lien que le lien de la ligne insérée avec `C` via `A.b1` et `B.c1`.

*B) Suppression d'une ligne dans A*

```
NO PROTOCOL
```

*C) Modification d'une ligne dans A*

On ne considère que la modification de A.b1.

```
REPLACE a BY a' IN A | a[b1] <> a'[b1]
PRECOND ∃ b ∈ B | a[b1] = b[b1] AND b[c1] = a[c1]
IMPLIES REPLACE a' | (∃ b ∈ B | a'[b1] = b[b1])
                BY a'' IN A| a''[c1] = b[c1]
```

Cette UC est très proche de celle concernant l'insertion d'une ligne dans A.

*C) Ajout d'une ligne dans B*

```
NO PROTOCOL
```

Les opérations d'ajout sur B n'apporteront aucune indication sur la redondance.

*D) Suppression d'une ligne de B*

```
DELETE b FROM B
IMPLIES REPLACE ALL a | a[b1] = b[b1] BY a' IN A | a'[b1] = NULL
                                          AND a'[c1] = NULL
```

Une suppression d'une ligne de B entraîne la suppression de certains liens A-B , B-C. Il faut donc aussi supprimer les liens A-C correspondants.

*E) Modification d'une ligne de B*

Nous ne considérons pas la modification de l'identifiant de B, mais seulement la modification de la clé étrangère B.c1.

```
REPLACE b BY b' IN B | b[c1] <> b'[c1]
IMPLIES REPLACE ALL a | a[b1] = b'[b1] BY a' IN A | a'[c1] = b'[c1]
```

Une modification de B.c1 correspond à une modification de certains liens B-C. Il y a donc des liens A-B , B-C qui sont modifiés. Il faut alors modifier les liens A-C correspondants.

Remarque

Les UC concernant les opérations sur C n'apportent aucune indication sur la gestion de la redondance.

**Recherche dans les triggers**

*A) Ajout d'une ligne dans A*

Le trigger de vérification implémentant la précondition sera de la forme :

```
CREATE TRIGGER ON A
FOR INSERT
AS IF
       (SELECT c1
        FROM inserted)
       <>
       (SELECT B.c1
        FROM B, inserted
        WHERE inserted.b1 = B.b1)

BEGIN
   gestion-erreur
END
```

Ce trigger vérifie que la clé étrangère redondante dans A correspond à un lien entre A et C via B.

Le trigger d'action correspondant à l'IMPLIES sera de la forme :

```
CREATE TRIGGER ON A
FOR INSERT
AS

BEGIN

       UPDATE A SET c1 = (SELECT B.c1
                          FROM inserted , B
                          WHERE inserted.b1 = B.b1)

       FROM inserted
       WHERE A.a1 = inserted.a1

END
```

*B) Modification d'une ligne de A*

On aura le même genre de triggers que pour l'ajout dans A.

*C) Suppression d'une ligne de B*

On aura un trigger du genre :

```
CREATE TRIGGER nom-trigger ON B
FOR DELETE
AS  UPDATE A
    SET A.b1 = NULL , A.c1 = NULL
    WHERE A.b1 = deleted.b1
```

Ce trigger assigne une valeur nulle à l'attribut de A qui référence B, ainsi qu'à l'attribut de A qui référence C. C'est l'assignation d'une valeur nulle à l'attribut de référence vers C qui gère la redondance : en effet, si on supprime un lien entre A et B, le lien entre A, B et C n'existe plus, et il faut donc aussi supprimer le lien entre A et C.

*D) Modification d'une ligne de B*

On aura un trigger de la forme :

```
CREATE TRIGGER ON B
FOR UPDATE
AS IF UPDATE (B.c1)
     BEGIN

          UPDATE A SET A.c1 = (SELECT c1
                              FROM inserted)
          FROM inserted
          WHERE A.b1 = inserted.b1

     END
```

Ce trigger modifie la clé étrangère vers C dans A, suite à une modification de la clé étrangère vers C dans B.

*Conclusion sur la recherche dans les triggers*

Il y a beaucoup de façons de gérer la redondance entre clé étrangères par les triggers. Les triggers développés ci-dessus ne représentent qu'un moyen parmi d'autres. On peut cependant dégager un principe général. Si une opération est réalisée sur une clé étrangère dans un trigger d'action et que cette clé étrangère ne référence pas la table de déclenchement, alors on peut supposer que cette clé étrangère est redondante. Pour les triggers de vérification, il suffit de voir si une clé étrangère de la table de déclenchement est comparée (opérateur <>) au résultat d'une requête portant sur plusieurs tables.

**Recherche dans les requêtes**

*A) Ajout d'une ligne dans A*

Supposons que l'on ajoute dans A la ligne

```
(var-a1 , var-a2 , var-b1 , var-c1)
```

Pour la précondition, on aura d'abord la requête suivante

```
EXEC SQL

        SELECT *
        FROM B
        WHERE b.b1 = :var-b1 AND b.c1 = :var-c1

END-EXEC
```

Si la requête ne donne pas de réponse, il y aura une gestion d'erreur. Sinon, on pourra faire l'insertion dans A.

Pour l'IMPLIES, on connait var-a1, var-a2 et var-b1, mais il est possible que l'on ne connaisse pas var-c1. On fera alors la requête suivante :

```
EXEC SQL

        SELECT B.c1 INTO :var-c1
        FROM B
        WHERE B.b1 = :var-b1

END-EXEC
```

On fera ensuite l'insertion dans A.

*B) Modification d'une ligne de A*

On aura le même genre de requêtes que pour l'ajout dans A.

*C) Suppression d'une ligne de B*

Supposons que l'on supprime la ligne de B identifiée par var-b1. Après le DELETE, on aura une mise à jour dans A :

```
EXEC SQL

         UPDATE A
         SET c1 = NULL , b1 = NULL
         WHERE b1 = :var-b1

END-EXEC
```

*D) Modification d'une ligne de B*

Supposons que l'on modifie la ligne identifiée par `var-b1`, et que l'on remplace `c1` par `var-c1`.

```
EXEC SQL

        UPDATE B
        SET c1 = :var-c1
        WHERE b1 = :var-b1

END-EXEC
```

Après cet `UPDATE`, on aura la requête :

```
EXEC SQL

        UPDATE A
        SET c1 = :var-c1
        WHERE b1 = :var-b1

END-EXEC
```

*Conclusion sur la recherche dans les requêtes*

- Pour les requêtes implémentant les préconditions, avant une mise à jour sur la table contenant la clé étrangère redondante, on aura une requête de sélection comparant la variable correspondant à la clé étrangère redondante avec les clés étrangères source.
- Pour les requêtes implémentant les `IMPLIES`,on peut envisager deux cas. Soit on aura une requête calculant la valeur de la clé étrangère redondante, soit on aura deux opérations (de suppression ou de modification) réalisées consécutivement. L'une portera sur la table contenant la clé étrangère redondante et une clé étrangère source, l'autre sur la table contenant l'autre clé étrangère source.

## 4.3.2.2.3 Généralisation

Dans le schéma sur lequel on a travaillé, le T.A. RAC n'est redondant qu'avec la composition de deux T.A.. Il est possible de généraliser à la composition de n T.A..

La redondance entre T.A. peut aussi concerner des T.A. many-to-many. Conceptuellement, il s'agit d'une généralisation de la redondance entre T.A. one-to-many. Les recherches seront cependant assez différentes et un peu plus complexes. En annexe, nous proposons une recherche de la redondance entre T.A. many-to-many (cf annexe 4).

### 4.3.3 Les transformations de restructuration

Nous avons déjà analysé les transformations de restructuration dans le tome 1. Nous n'avons pas envisagé de développements techniques pour ce type de transformations.

# 4.4 EXPRESSION DU SCHEMA DANS UN MODELE DE PLUS HAUT NIVEAU

## *4.4.1 Les types d'associations complexes*

Nous allons considérer 3 sortes de T.A. complexes : les T.A. de degré supérieur à 2, les T.A. dotés d'attributs, et les T.A. de degré 2 mais qui sont de connectivités maximales N (c-à-d les T.A. many-to-many). La transformation de ces T.A. en structures conformes au modèle relationnel peut être effectuée par une transformation de ceux-ci en types d'entités. C'est la manière classique, mais il y a d'autres façons de les transformer. Nous allons voir dans les 3 cas de quelle façon on peut retrouver le T.A. complexe dans le cas d'une transformation classique et, le cas échéant, dans le cas d'une transformation non-classique.

### 4.4.1.1 Les types d'associations many-to-many

Considérons le schéma conceptuel de la figure 4.41.

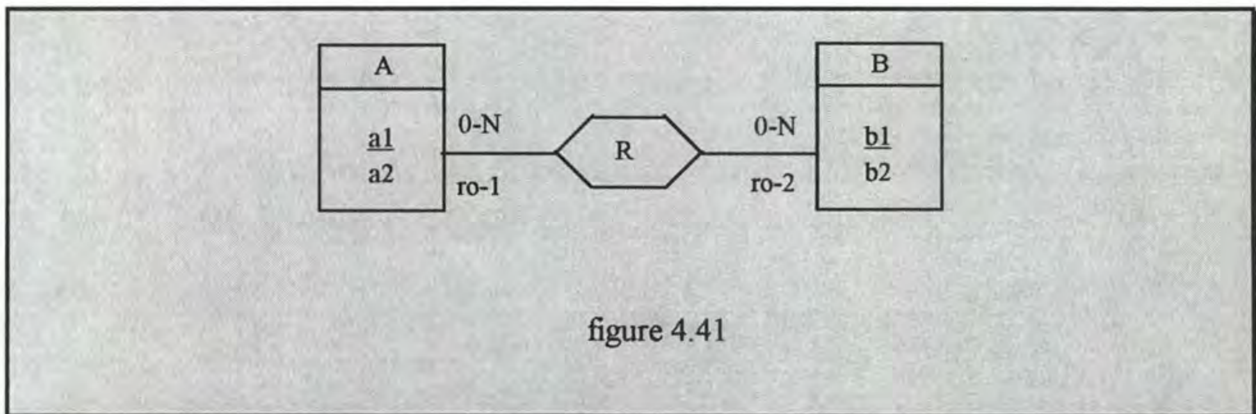


figure 4.41

#### 4.4.1.1.1 Transformation classique

**Principe**

Le type d'associations R sera transformé en type d'entités, et on aura le schéma de la figure 4.42.

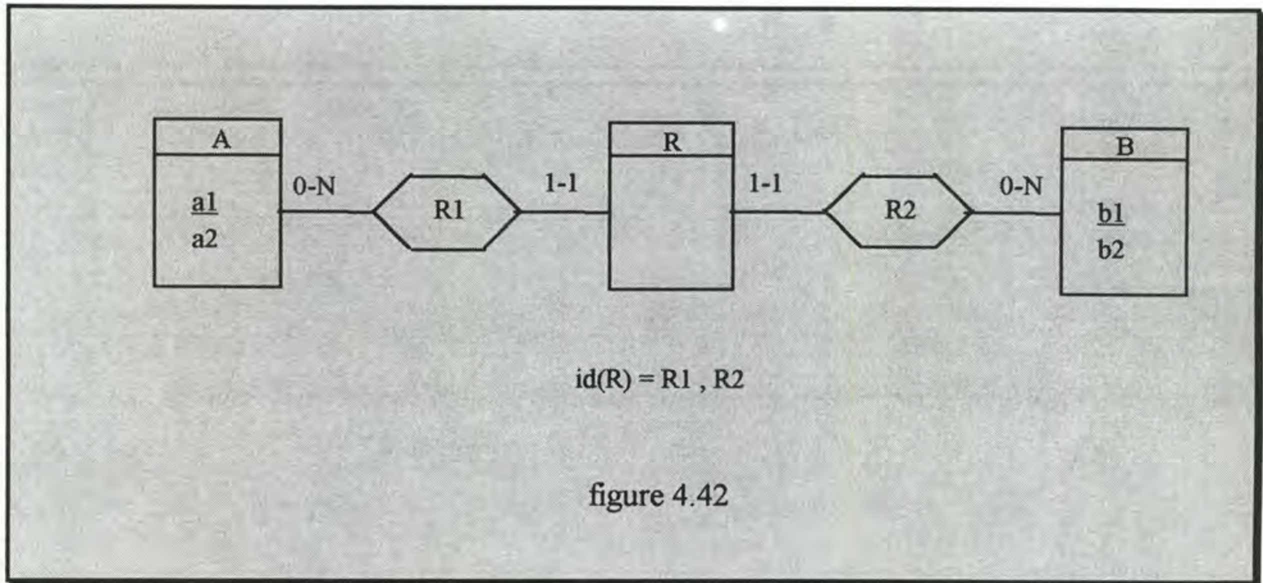



figure 4.42

Si le type d'associations R contenait des attributs, ils feraient maintenant partie du type d'entités R.

**Recherche**

Si on trouve un type d'entités contenant peu (ou pas) d'attributs, lié à deux autres types d'entités, jouant les rôles 1-1, et dont l'identifiant est constitué des rôles des types d'entités qui lui sont liés, alors il y a de fortes chances pour qu'il s'agisse de la traduction d'un type d'associations many-to-many.

Il suffit alors de faire la transformation inverse (remplacer le type d'entités par un type d'associations many-to-many).

Donc, partant du schéma de la figure 4.42, on remplacera le type d'entités R par un type d'associations R, reliant A et B, et les connectivités des rôles joués par A et B dans R (c-à-d les connectivités de ro-1 et ro-2) seront 0-N (puisque les connectivités de A-R1 et de B-R2 sont 0-N).

## 4.4.1.1.2 Transformation non-classique

**Principe**

Si on connaît la connectivité maximale (soit m) du rôle d'un type d'entités dans un type d'associations, on peut remplacer ce type d'associations par une liste de m clés étrangères dans l'autre type d'entités. Par exemple, pour le schéma de la figure 4.41, si la connectivité maximale de ro-2 était 3, on pourrait recopier 3 fois l'attribut identifiant A.a1 dans B. On obtiendra le schéma conforme au relationnel de la figure 4.43.

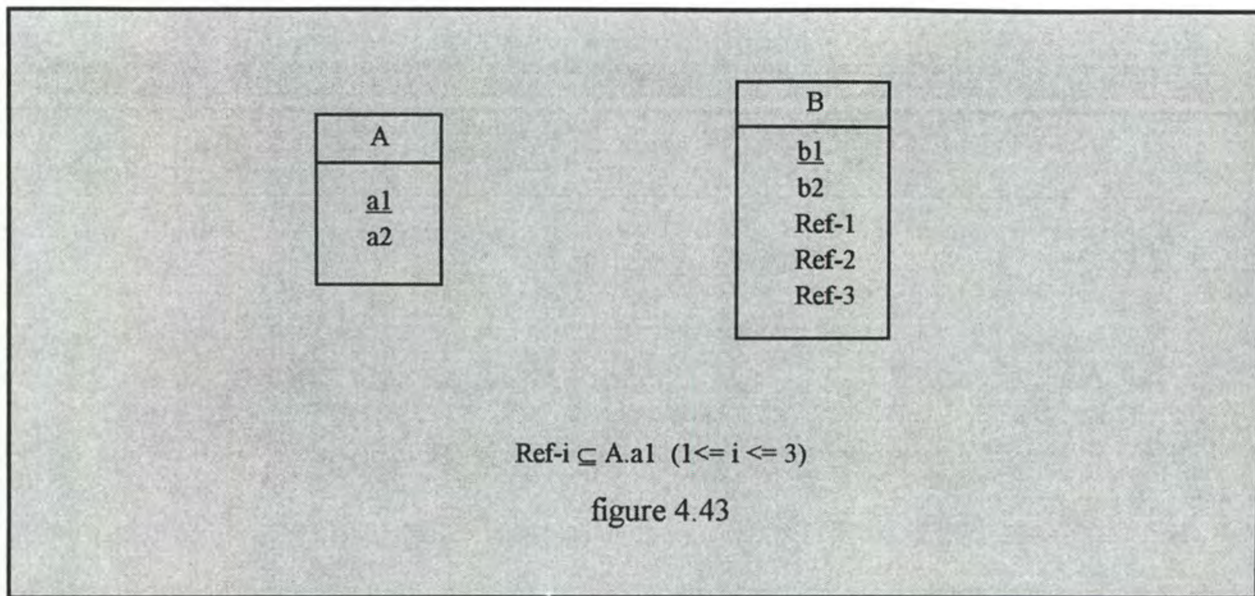


figure 4.43

`Ref-1`, `Ref-2` et `Ref-3` représentent des attributs facultatifs, à cause de la connectivité minimale 0 pour ro-2. Si la connectivité minimale de `ro-2` avait été 1, alors on aurait eu la contrainte suivante : pour un tuple (`Ref-1` , `Ref-2` , `Ref-3`), il y au moins un des éléments qui n'a pas la valeur nulle.

**Recherche**

Si on appliquait au schéma de la figure 4.43 la recherche des types d'associations one-to-many, on obtiendrait le schéma de la figure 4.44.

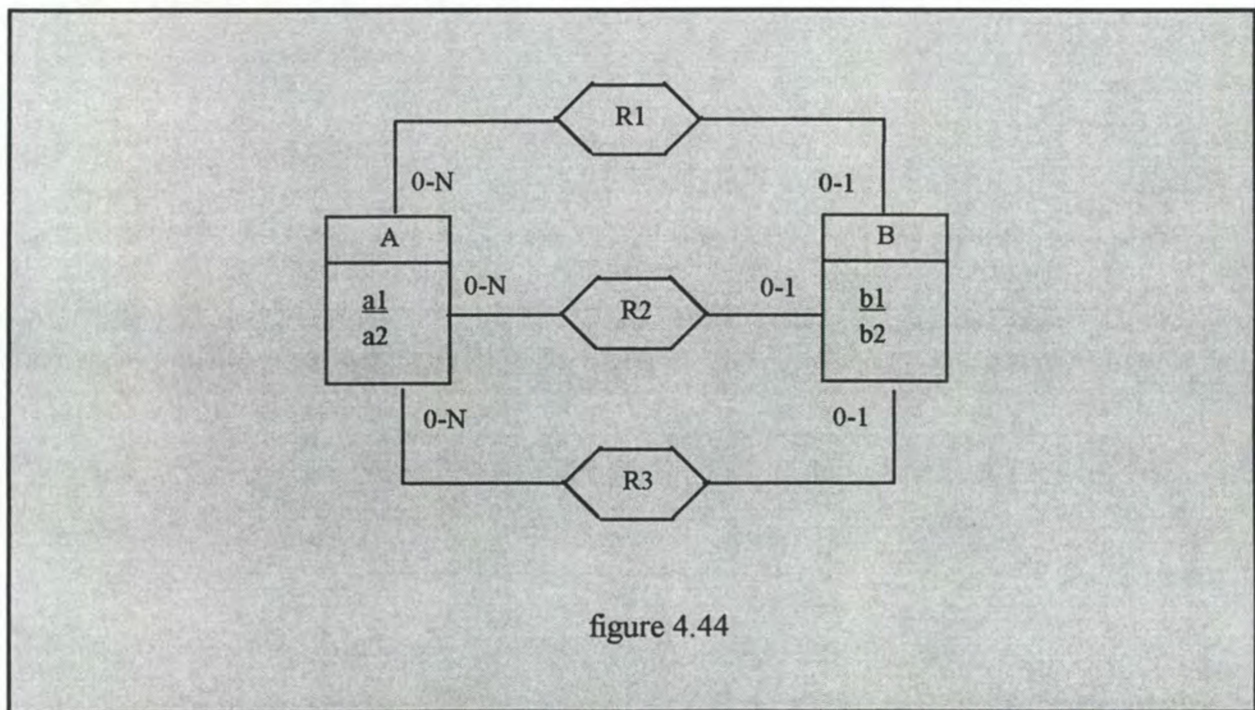


figure 4.44

Ce n'est pas le schéma conceptuel que l'on attendait. Une façon de retrouver le bon schéma est de détecter ce type de configurations, et c'est à l'utilisateur de décider s'il s'agit ou pas d'une many-to-many.

On peut aussi essayer de trouver des méthodes spécifiques pour rechercher des indices de la présence d'associations many-to-many transformées de manière non-classique. Nous allons les examiner.

***Recherche sur les noms***

On s'intéressera aux clé étrangères ayant des noms similaires et aux clé étrangères numérotées (comme Ref-1 , Ref-2 et Ref-3 dans le schéma de la figure 4.43).

***Recherche dans les comportements imposés aux données***

Etant donné que les clés étrangères ne prendront pas nécessairement des valeurs distinctes dans une même ligne, il est très difficile de définir des UC spécifiques au cas qui nous occupe. En effet, voyons l'exemple de l'UC de modification d'une ligne de A. On aurait :

```
REPLACE a BY a' IN A | a[a1] <> a'[a1]
IMPLIES REPLACE b BY b' IN B |
        ((b[ref-1] = a[a1]) => b'[ref-1] = a'[a1])
    AND ((b[ref-2] = a[a1]) => b'[ref-2] = a'[a1])
    AND ((b[ref-3] = a[a1]) => b'[ref-3] = a'[a1])
```

Si on avait 3 clé étrangères représentant 3 types d'associations one-to-many, on aurait exactement la même UC.

Une bonne façon de trouver des indices est de regarder dans les triggers et les requêtes si des clé étrangères sont toujours utilisées ensembles, et jamais utilisées séparément.

***Recherche dans les consultations***

On peut se baser sur les jointures des requêtes de consultation. En effet, un lien entre A et B pourra être réalisé par une jointure à 4 arguments[14]. Ainsi on aura par exemple :

(1)

```
SELECT ...
FROM A , B , ...
WHERE ... AND (A.a1 = B.ref-1 OR A.a1 = B. ref-2 OR A.a1 = B. ref-3)
          AND ...
```

---

[14]Il s'agit en fait d'une jointure particulière pour laquelle la condition est constituée d'une disjonction de trois comparaisons

Si on retrouve une telle jointure, on peut faire l'hypothèse que Ref-1, Ref-2 et Ref-3 représentent le même type d'associations many-to-many.

Notons que cette jointure ne se retrouvera peut-être pas explicitement dans une requête :

(2)

```
EXEC SQL

       SELECT ref-1, ref-2, ref-3 INTO :var-1, :var-2, :var-3
       FROM B
       WHERE b1 = :var-b1

END-EXEC

EXEC SQL

       SELECT ... INTO ...
       FROM A
       WHERE a1 = :var-1 OR a1 = :var-2 OR a1 = :var-3

END-EXEC
```

Il se peut qu'à la place d'avoir un seul SELECT dans A, on en ait trois, un par clé étrangère. On aurait alors :

(3)

```
EXEC SQL

       SELECT ref-1, ref-2, ref-3 INTO :var-1, :var-2, :var-3
       FROM B
       WHERE b1 = :var-b1

END-EXEC

EXEC-SQL
       SELECT ... INTO ...
       FROM A
       WHERE a1 = :var-1
END-EXEC

EXEC-SQL
       SELECT ... INTO ...
       FROM A
       WHERE a1 = :var-2
END-EXEC

EXEC-SQL
       SELECT ... INTO ...
       FROM A
       WHERE a1 = :var-3
END-EXEC
```

## 4.4.1.2 Les types d'associations contenant un attribut

### 4.4.1.2.1 Transformation classique

**Principe**

C'est le même principe que pour la transformation classique d'un type d'associations many-to-many. Ce n'est pas implémentable en relationnel, et il faut aussi remplacer ce type d'associations par un type d'entités.

**Recherche**

C'est la même idée que pour le cas précédent : le type d'entités devient un type d'associations contenant les mêmes attributs.

### 4.4.1.2.2 Transformation non-classique

**Principe**

Considérons le schéma de la figure 4.45.

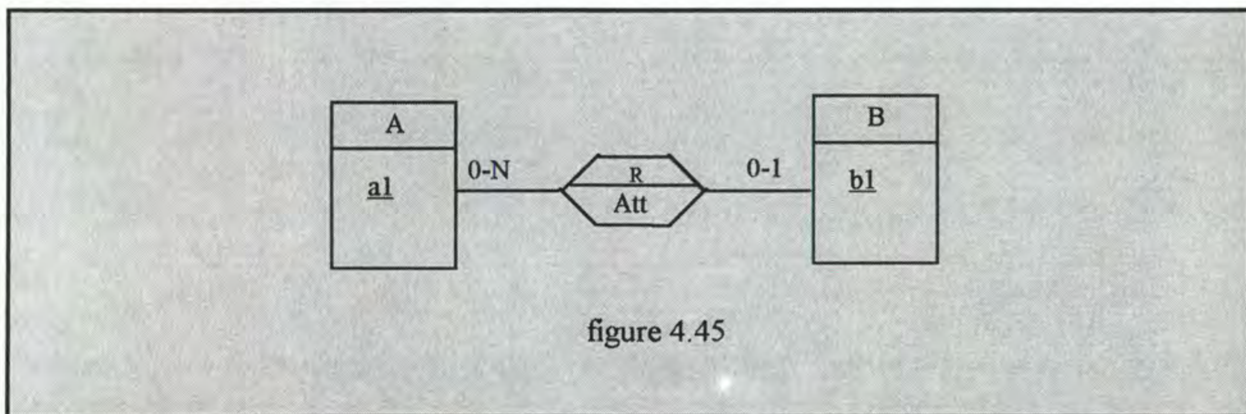


figure 4.45

Une manière de rendre ce schéma conforme au modèle relationnel est de faire passer `Att` dans `B`. Il y aura alors dans `B` une dépendance fonctionnelle de la clé étrangère vers `Att`. On aura donc le schéma conforme au relationnel de la figure 4.46.

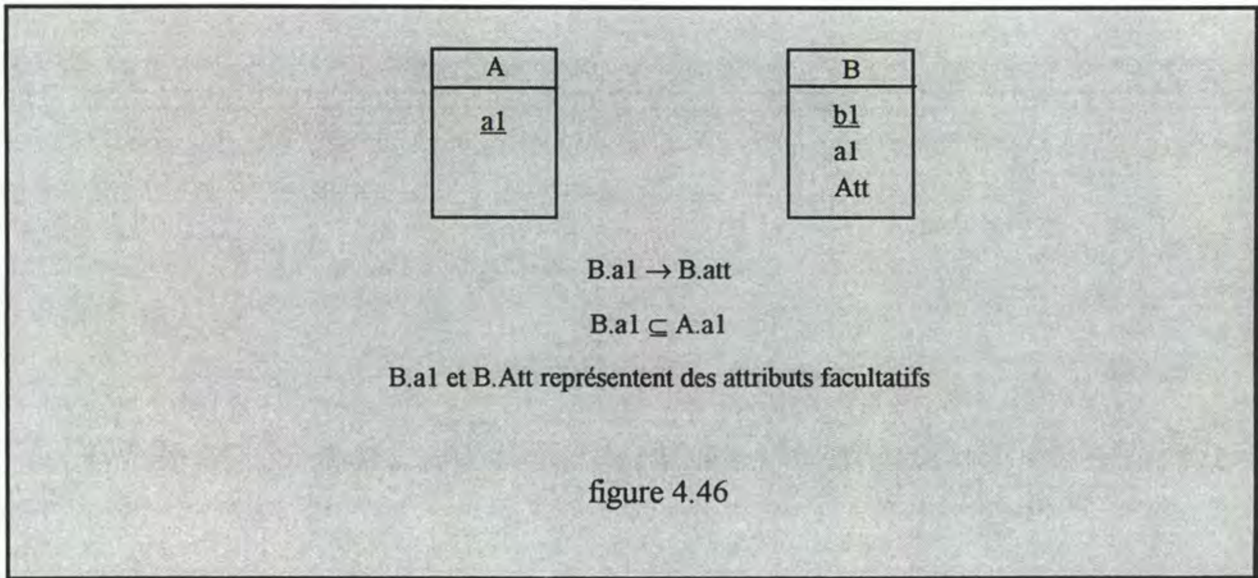



figure 4.46

**Recherche**

Considérons que les tables ne sont pas encore normalisées. Si on trouve une dépendance fonctionelle dont le déterminant est une clé étrangère, alors lorsqu'on transformera la clé étrangère en type d'associations, le déterminé fera partie de ce type d'associations. Ce type de recherche devrait donc être fait en même temps que la recherche des types d'associations one-to-many.

## 4.4.1.3 Les types d'associations de degré supérieur à 2

### 4.4.1.3.1 Principe

Prenons le cas d'un type d'associations ternaire. Considérons le schéma de la figure 4.47. La transformation de ce schéma peut être réalisée en remplacanr R par en un type d'entités. On aura le schéma de la figure 4.48.

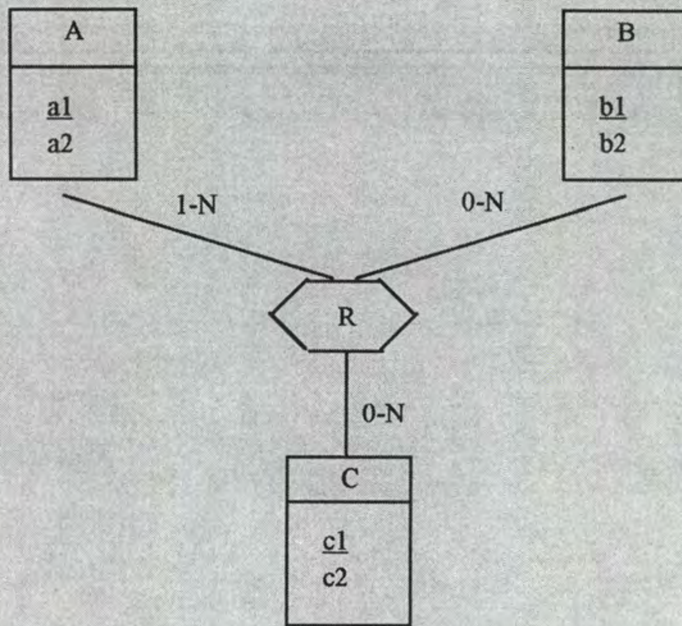


figure 4.47

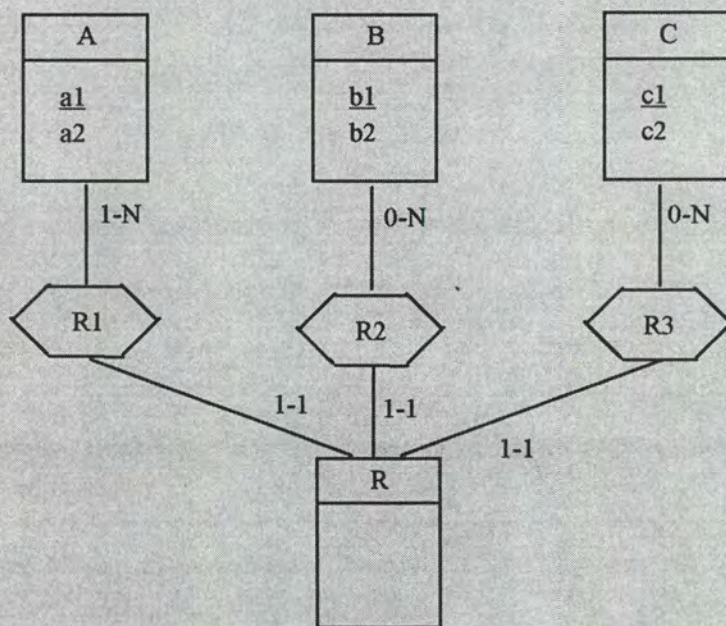


figure 4.48

#### 4.4.1.3.2 Recherche

Si on trouve un type d'entités contenant peu (ou pas) d'attributs, associé à plus de deux autres types d'entités et jouant dans chaque type d'associations un rôle de connectivités 1-1, il y a des chances pour que ce soit la traduction d'un type d'associations entre tous ces types d'entités.

Il suffit ici aussi d'appliquer la transformation inverse (le type d'entités devient un type d'associations).

## *4.4.2 Les attributs particuliers*

Les transformations que nous allons évoquer ici sont tirées de [BEL93].

### 4.4.2.1 Les attributs facultatifs

Nous avons déjà vu dans l'extraction des structures de données une façon de transformer un attribut facultatif. Il s'agissait de le rendre obligatoire. Un attribut facultatif peut être aussi transformé en un type d'entités. Nous allons voir deux formes de représentation de cet attribut.

#### 4.4.2.1.1 Représentation par instance

**Principe**

La transformation est schématisée à la figure 4.49. Une occurence du type d'entités `A''` représente une instance de l'attribut `a2`.

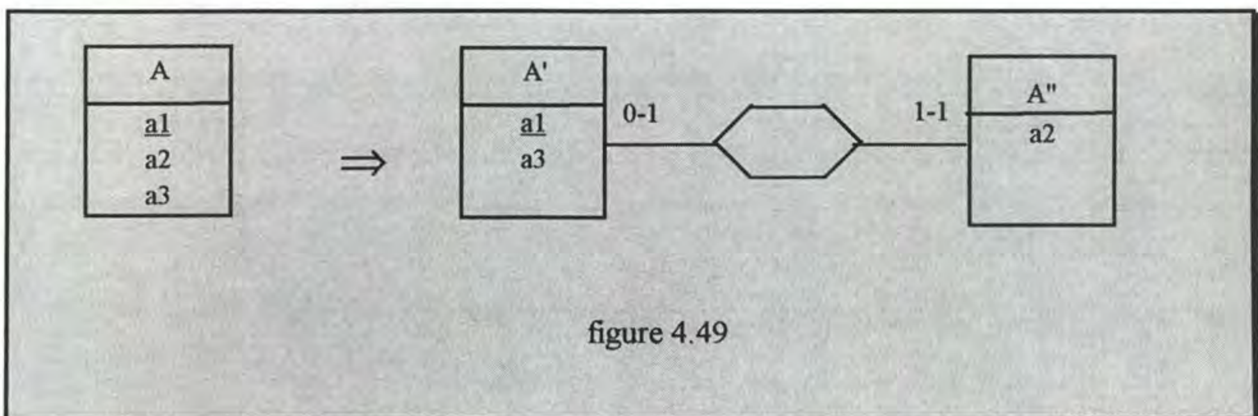


figure 4.49

**Recherche**

On cherchera des T.E. reliés à un et un seul autre T.E. De plus, on se basera sur les connectivités et le nombre d'attributs dans A" (il n'y en a qu'un).

### 4.4.2.1.2 Représentation par valeur

**Principe**

La transformation de la figure 4.50 est une variante de la précédente. Une occurence du type d'entités A'' représente une valeur de a2 pour une ou plusieurs occurences de A. Dès lors, a2 est identifiant de A'' et la connectivité maximale du rôle joué par A'' est N.

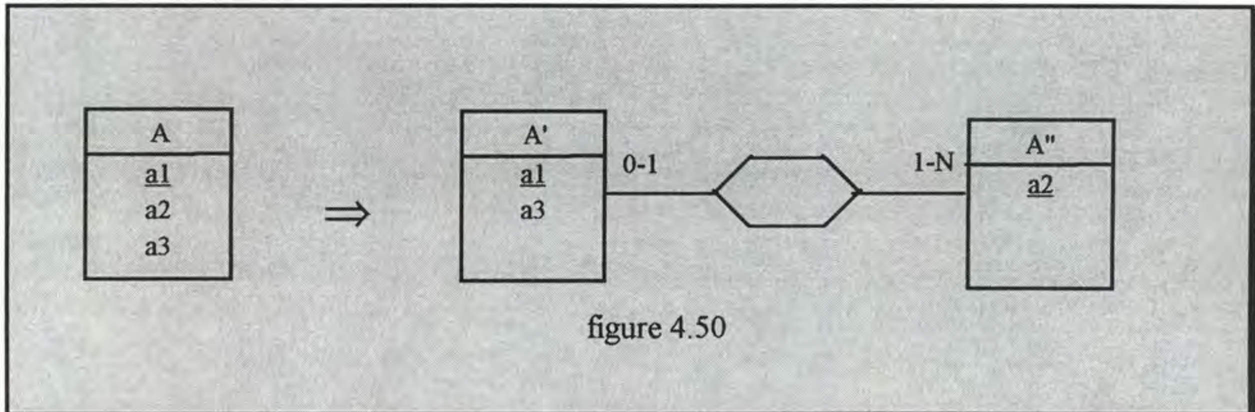



figure 4.50

**Recherche**

La recherche de ce type de transformation est identique à la précédente.

## 4.4.2.2 Les attributs décomposables

### 4.4.2.2.1 Transformations par représentation par des attributs simples

**Transformation par concaténation**

*Principe*

Considérons la transformation de la figure 4.51.

L'attribut a2 de A' contient la concaténation des attributs a2.a21 et a2.a22 de A. Cette transformation entraîne une perte de sémantique. En effet, on n'a plus dans A' l'organisation des données représentée dans A par la structure interne de A.a2.

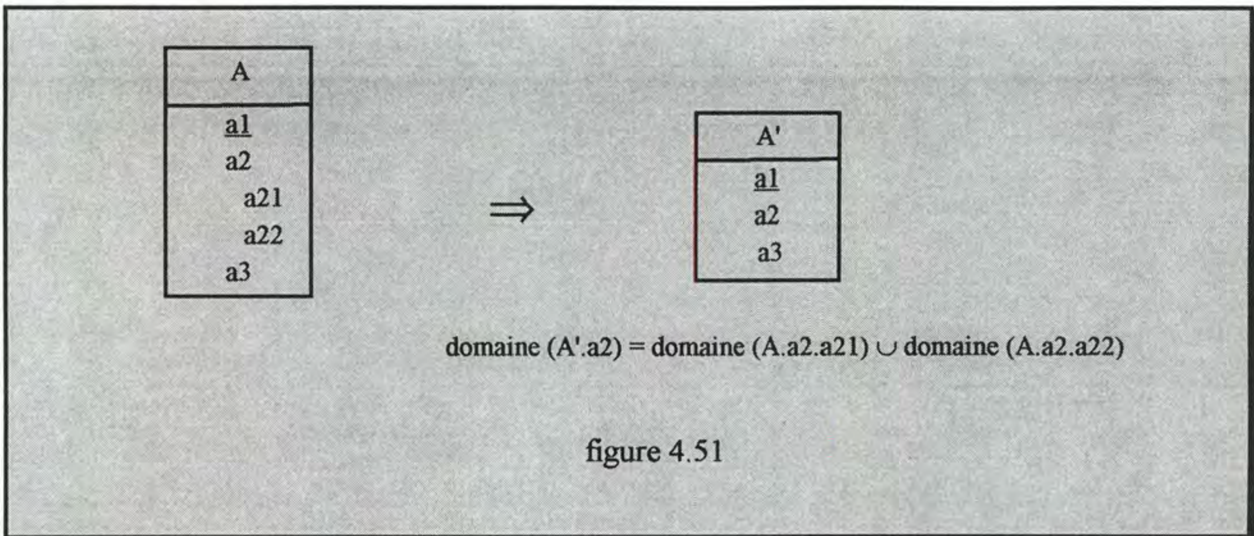



figure 4.51

*Recherche*

La première chose que nous constatons, c'est qu'on ne saurait pas retrouver la structure de a2. On peut cependant détecter la concaténation. On retrouvera une structure de concaténation en s'intéressant principalement à la longueur et au type de l'attribut (il sera de type caractère), ainsi qu'aux vues. Il restera à retrouver la structure de l'attribut décomposable.

**Transformation par décomposition**

*Principe*

Considérons la transformation de la figure 4.52.

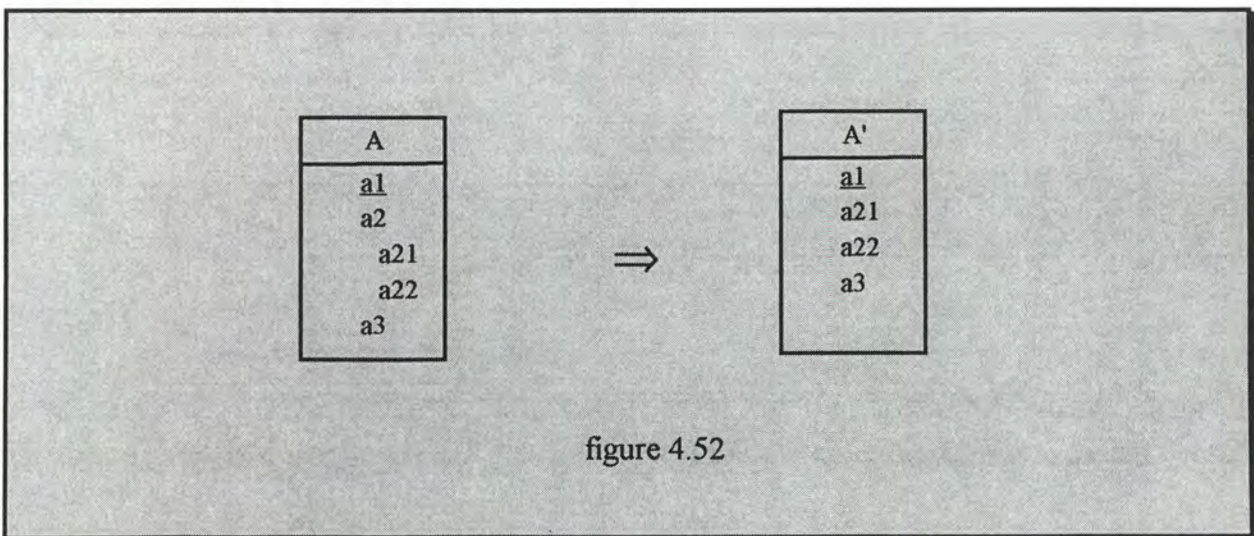



figure 4.52

On remplace dans `A'` l'attribut `a2` de `A` par ses composants. On constate aussi une perte de sémantique due à la déstructuration de `A.a2`.

*Recherche*

On peut détecter le résultat d'une telle transformation par une recherche sur les noms. En effet, les attributs a21 et a22 auront probablement le même préfixe, qui sera une abréviation du nom de a2.

### 4.4.2.2.2 Transformations par représentation par des types d'entités

**Représentation par instance**

*Principe*

Considérons la transformation de la figure 4.53.

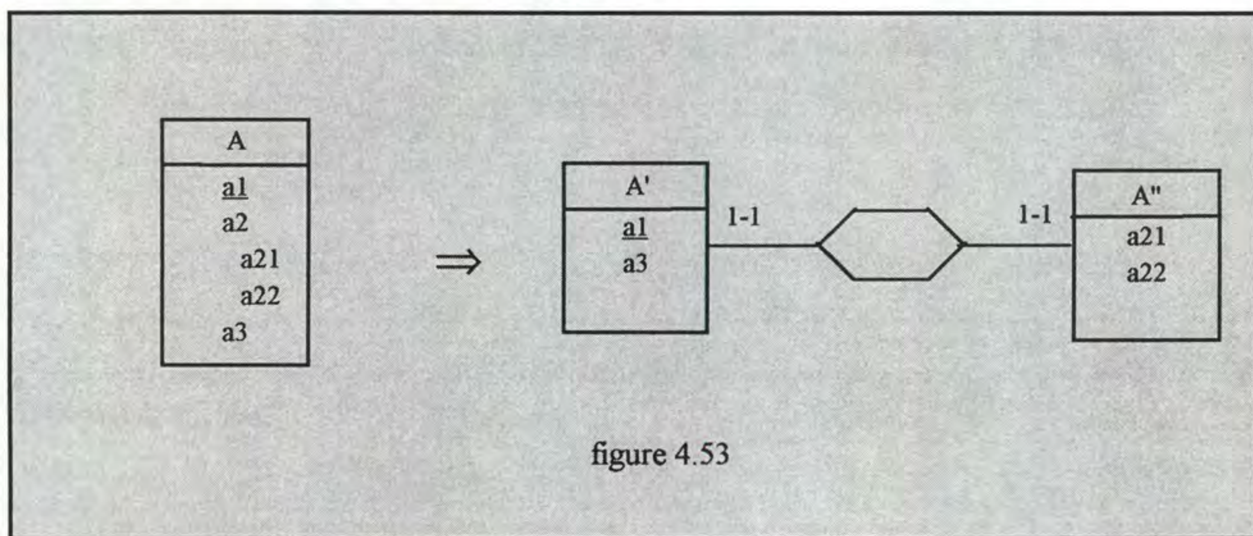


figure 4.53

Une instance de A'' est créée pour chaque instance de A.

*Recherche*

On cherchera les T.E. associés à un et un seul autre T.E. On s'intéressera aussi aux connectivités du type d'associations et au nombre d'attributs présents dans A'' (ils seront peu nombreux). On peut aussi faire une recheche sur les noms.

**Représentation par valeur**

*Principe*

Considérons le schéma de la figure 4.54.

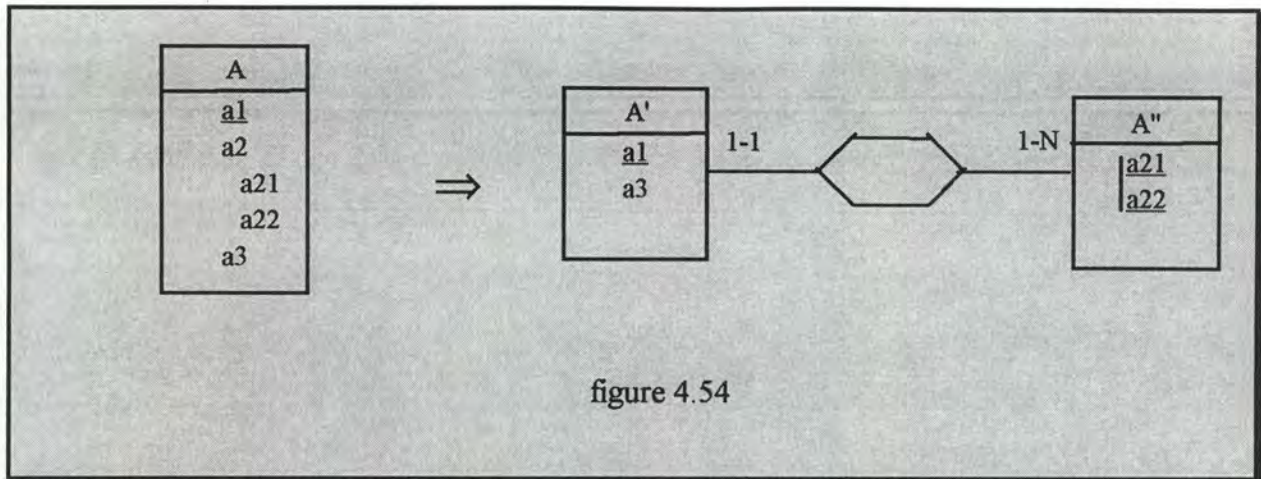


figure 4.54

Pour chaque instance du couple (a21,a22)de A, une instance de A'' est créée. L'identifiant de A'' est donc (a21,a22).

*Recherche*

La recherche sera similaire à la précédente.

## 4.4.2.3 Les attributs multivalués

On trouve dans [BEL93] divers types de transformations. Celles-ci sont différentes selon que l'on considère que les valeurs de l'attribut multivalué possèdent ou pas des doubles et sont ordonnés ou pas. Nous allons considérer ici le cas où les attributs multivalués n'admettent pas de doubles et ne sont pas ordonnés. Il y a 5 transformations de tels attributs.

### 4.4.2.3.1 Représentation des instances

**Principe**

Sur la figure 4.55, on voit que l'on transforme le type d'entités A de telle façon qu'à chaque instance de a2 corresponde une instance du type d'entités A''.

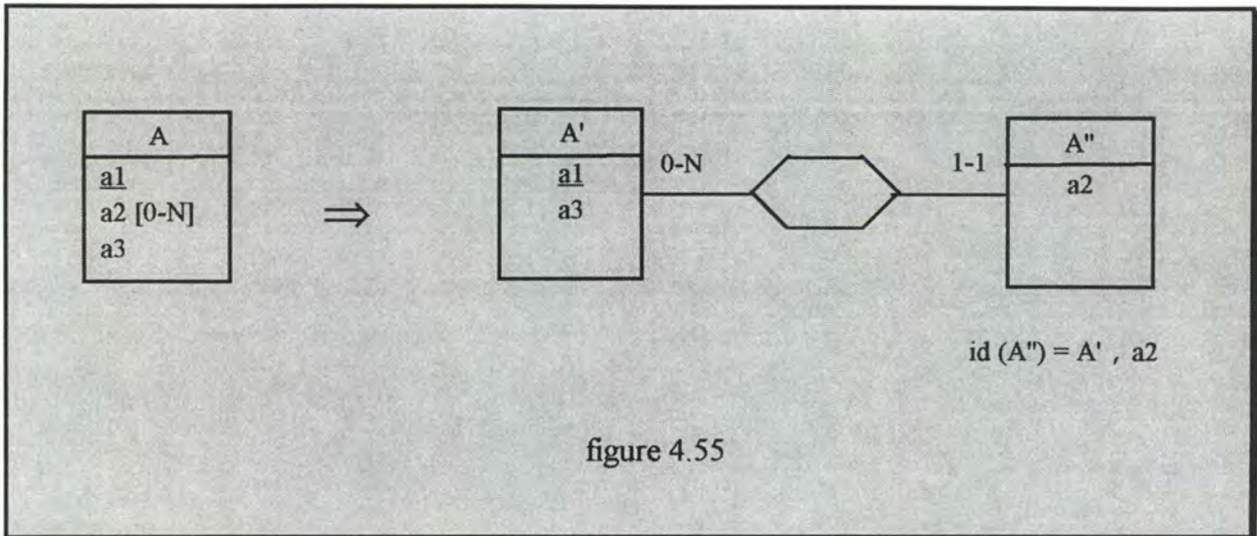


figure 4.55

**Recherche**

On cherchera des T.E. ne contenant qu'un attribut et jouant un rôle de connectivité 1-1 dans un et un seul T.A. De plus, l'identifiant de ce T.E. est hybride : il s'agit du couple formé par son attribut et le rôle joué par l'autre T.E. dans le T.A. On peut augmenter le niveau de certitude de la recherche en analysant les noms des T.E. et des attributs.

### 4.4.2.3.2 Représentation des valeurs

**Principe**

Dans la transformation de la figure 4.56, on voit que le type d'entités `A''` représente une valeur que l'attribut `A.a2` prend pour une ou plusieurs occurences de `A`. On a alors un type d'associations many-to-many, et le type d'entités `A''` est identifié par `a2`.

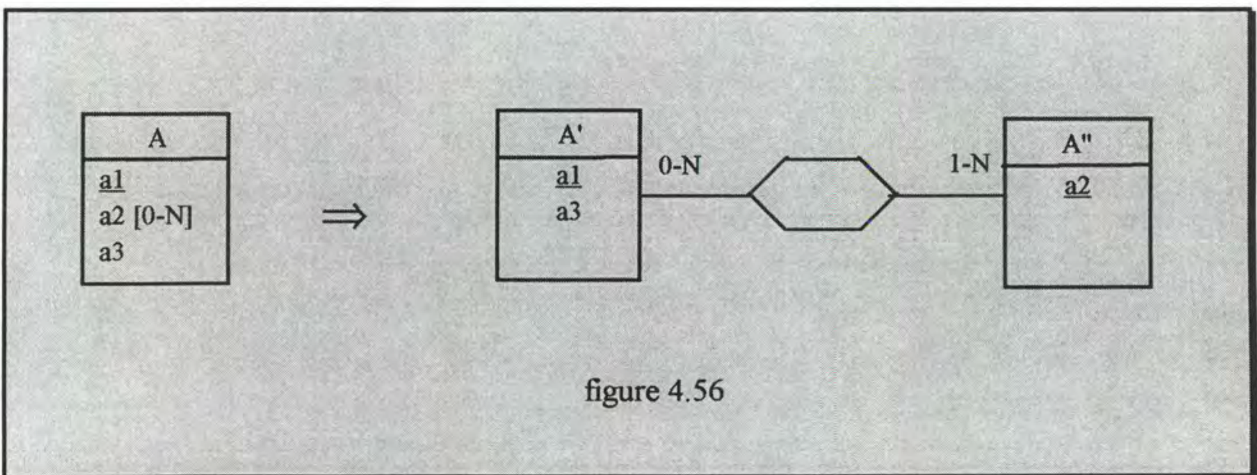


figure 4.56

**Recherche**

On recherchera un type d'associations many-to-many où un des deux types d'entités qu'il associe ne contient qu'un attribut. On peut ici aussi augmenter le niveau de certitude par une recherche sur les noms.

### 4.4.2.3.3 Augmentation de l'identifiant (dénormalisation)

**Principe**

Il s'agit d'une variante de la technique de représentation des instances. Sur la figure 4.57, on voit que pour chaque instance de `A.a2` un type d'entité `A'` est créé. L'identifiant de ce type d'entités est formé de l'identifiant de `A` et de l'attribut `a2`.

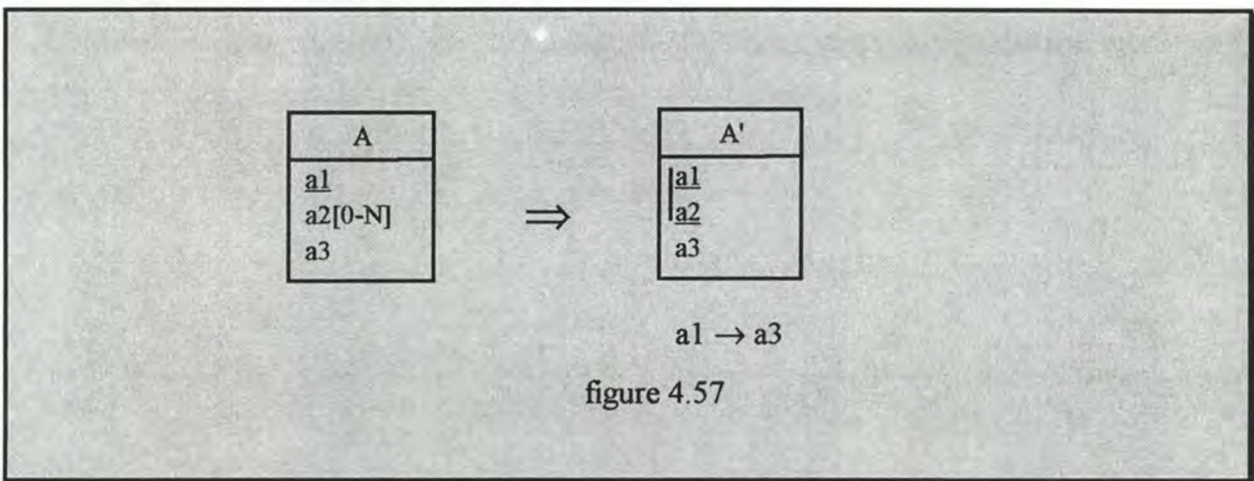



figure 4.57

**Recherche**

On cherchera des types d'entités ayant un identifiant composé, et dont l'un des composants détermine fonctionnellement un ou plusieurs attributs non identifiants.

### 4.4.2.3.4 Instanciation

**Principe**

Considérons la transformation de la figure 4.58. La technique de l'instanciation constiste à représenter chaque instance de l'attribut multivalué par un attribut monovalué. Cette technique n'est applicable que si l'on connaît la valeur de N.

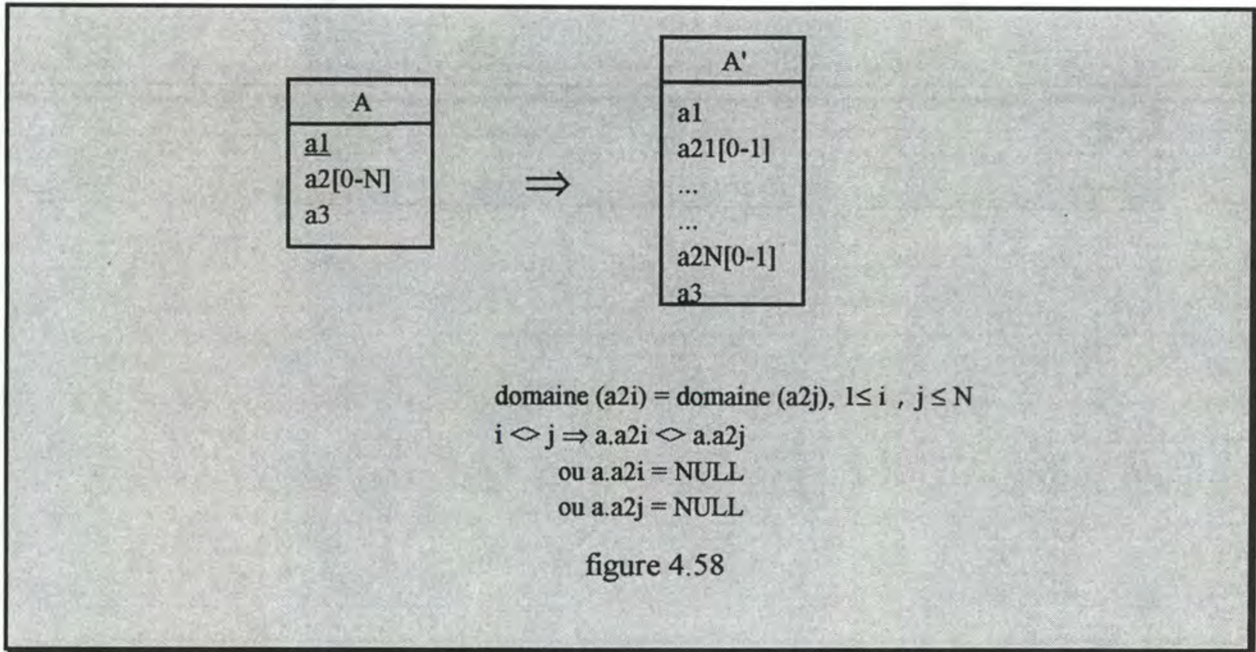



figure 4.58

**Recherche**

On s'intéressera aux types d'entités contenant un grand nombre d'attributs. La recherche se basera sur diverses caractéristiques de ces attributs (nom identique à un suffixe ou préfixe près, même domaine, etc.).

## 4.4.2.3.5 Concaténation

**Principe**

Sur la figure 4.59, on voit qu'on a remplacé l'attribut multivalué par une concaténation des instances de celui-ci. Il faut bien entendu connaître N pour pouvoir appliquer cette transformation.

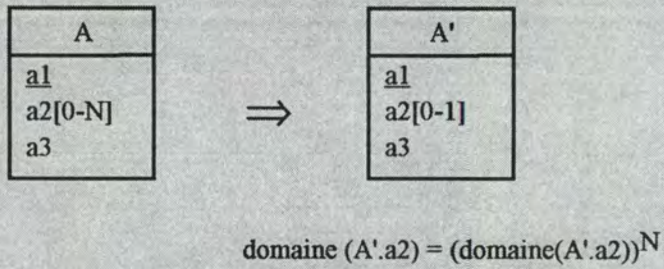



figure 4.59

**Recherche**

La recherche se basera sur le domaine de A'a2. Il présente en effet un certain nombre de caractéristiques particulières (le domaine est toujours du caractère, on trouve des caractères de remplissage, etc.).

## *4.4.3 Les contraintes d'intégrité sur les rôles et les types d'associations*

Dans [BOD89], on trouve une définition des contraintes qui peuvent être posées sur des rôles ou des types d'associations. Il s'agit des contraintes d'inclusion, d'exclusion et d'égalité. Nous allons analyser ces contraintes, en y ajoutant les contraintes d'existence entre rôles.

Après les avoir brièvement décrites, nous verrons comment il est possible de les retrouver. Pour ce faire, nous verrons comment la présence de ces contraintes peut se refléter dans un schéma relationnel résultant de la traduction du schéma E/A et dans les sources d'information que nous utilisons.

Puisqu'en relationnel la notion de rôle n'existe pas, nous parlerons de contrainte entre liens. Dans un schéma relationnel, les liens entre tables sont implémentés par utilisation d'une clé étrangère. C'est donc du côté des clé étrangères qu'il faudra chercher pour trouver des traces de gestion de contraintes sur des rôles.

Remarque

Jusqu'à présent, lorsque nous définissions les UC, nous nous intéressions à l'ajout, la modification ou la suppression d'une ligne d'une table.

Dans cette partie nous allons voir ces mêmes opérations sur des liens entre tables, c-à-d uniquement sur des clé étrangères. Considérons par exemple l'ajout d'un lien. Il peut être réalisé en mettant une valeur non-nulle dans une clé étrangère qui était nulle préalablement. L'UC sera une UC de modification, mais logiquement, il s'agit de l'ajout d'un lien.

Nous ne nous intéresserons donc pas aux opérations sur toute la ligne d'une table. En fait, l'ajout d'un lien entre deux tables peut aussi être réalisé par l'ajout d'une ligne dans cette table avec une valeur non-nulle pour la clé étrangère, et la suppression d'une ligne d'une table entraîne forcément la suppression du lien correspondant à la clé étrangère de cette ligne. Les principes que nous tirerons de la recherche seront donc valables aussi dans le cas d'opérations sur des lignes de tables, avec quelques adaptations.

### 4.4.3.1 Les contraintes sur les rôles

Il s'agit de contraintes entre rôles assumés par un même type d'entités.

#### 4.4.3.1.1 Contraintes d'inclusion

**Définition**

Considérons le schéma de la figure 4.60.

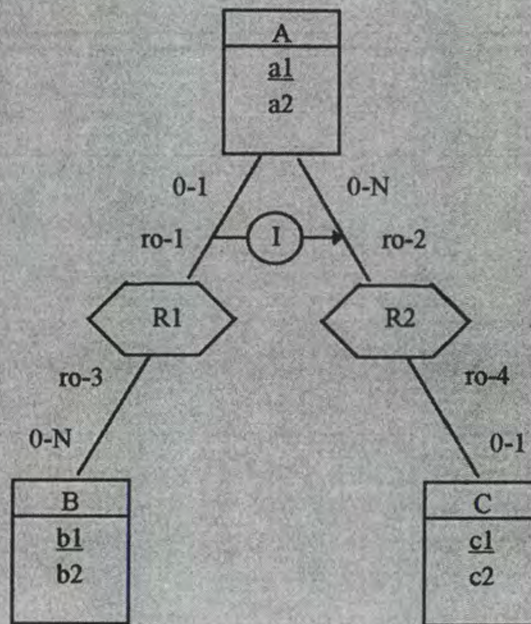



figure 4.60

La contrainte d'inclusion entre ro-1 et ro-2 peut s'exprimer comme suit : si une entité a joue le rôle ro-1 dans une association de type R1, alors elle doit aussi jouer le rôle ro-2 dans une association de type R2.

Si la connectivité minimale de ro-2 était 1, alors il y avait redondance entre la contrainte d'inclusion et la contrainte de connectivité.

Le schéma de la figure 4.61 donne la traduction en relationnel du schéma de la figure 4.60.

```
A(a1, a2, b1)
B(b1, b2)
C(c1, c2, a1)

C.a1 ⊆ A.a1
A.b1 ⊆ B.b1
∀ a ∈ A : (a.b1 <> NULL) ⇒ (∃ c ∈ C | c.a1 = a.a1)
```

figure 4.61

**Recherche dans les checks**

On pourrait avoir un check sur A. Il serait de la forme :

```
CHECK (b1 IS NULL OR
       a1 IN (SELECT a1 FROM C))
```

Ce check vérifie que s'il y a un lien A-B il y a un lien A-C.

**Recherche dans les comportements imposés aux données**

***Définition des UC***

*A) Ajout d'un lien entre A et B*

```
REPLACE a BY a' IN A | a[b1] = NULL AND a'[b1] <> NULL
ou bien PRECOND ∃ c ∈ C | c[a1] = a'[a1]
ou bien IMPLIES REPLACE ANY c | c[a1] = NULL BY c' IN C | c'[a1] =
                                                          a'[a1]
```

Lorsqu'on ajoute un lien de a ∈ A vers un b ∈ B, soit on vérifie qu'il y déjà un lien d'un c ∈ C vers a, soit on crée ce dernier.

*B) Suppression d'un lien entre A et B*

```
NO PROTOCOL
```

*C) Modification d'un lien entre A et B*

```
NO PROTOCOL
```

En effet, une contrainte d'inclusion ne vérifie pas quel est le lien entre deux tables, elle vérifie seulement la présence d'un lien par rapport à la présence d'un autre lien.

*D) Ajout d'un lien entre A et C*

```
NO PROTOCOL
```

En effet, l'ajout d'un lien A-C peut être fait indépendamment de l'ajout d'un lien A-B.

*E) Suppression d'un lien entre A et C*

On va s'intéresser ici à la 'suppression' de la clé étrangère de A.

```
REPLACE c BY c' IN C | c[a1] <> NULL AND c'[a1] = NULL
                         AND ¬ ∃ c''∈ C | c <> c'' AND c''[a1] = c[a1]
ou bien PRECOND ¬∃ a ∈ A | c[a1] = a[a1] AND a[b1] <> NULL
ou bien IMPLIES REPLACE a | c[a1] = a[a1] BY a' IN A | a'[b1] = NULL
```

A cause de la contrainte d'inclusion de A-B vers A-C, quand on supprime le dernier lien d'un c ∈ C vers un a ∈ A, soit on vérifie que a n'avait pas de lien vers un b ∈ B, soit on le supprime.

*F) Modification d'un lien entre A et C*

```
NO PROTOCOL
```

***Recherche dans les triggers***

*A) Ajout d'un lien entre A et B*

Le trigger de vérification correspondant à la précondition sera de la forme :

```
CREATE TRIGGER nom-trigger ON A
FOR UPDATE
AS IF UPDATE (b1)
      AND (SELECT b1 FROM deleted) IS NULL
      AND (SELECT b1 FROM inserted) IS NOT NULL
      AND NOT EXISTS (SELECT *
                      FROM C, inserted
                      WHERE C.a1 = inserted.a1)

BEGIN
   gestion-erreur
END
```

L'IMPLIES a peu de chances d'être implémenté par trigger, puisqu'on ne peut connaître à priori les lignes de C qui vont devoir être reliées au A modifié.

*B) Suppression d'un lien entre A et C*

Pour la précondition de l'UC, on aura le trigger de vérification suivant :

```
CREATE TRIGGER nom-trigger ON C
FOR UPDATE
AS IF UPDATE(a1)
      AND (SELECT a1 FROM deleted) IS NOT NULL
      AND (SELECT a1 FROM inserted) IS NULL
      AND ((SELECT COUNT(*)
             FROM C,deleted
             WHERE C.a1 = deleted.a1) = 0)
      AND (EXISTS(SELECT *
                   FROM A, deleted
                   WHERE A.a1 = deleted.a1 AND A.b1 IS NOT NULL))

BEGIN
    gestion-erreur
END
```

Pour l'`IMPLIES`, on aura le trigger d'action suivant :

```
CREATE TRIGGER nom-trigger ON C
FOR UPDATE
AS IF  UPDATE (a1)
       AND (SELECT a1 FROM deleted) IS NOT NULL
       AND (SELECT a1 FROM inserted) IS NULL
       AND (SELECT COUNT(*) FROM C,deleted WHERE C.a1 = deleted.a1)=0

BEGIN

  UPDATE A SET A.b1 = NULL
  FROM deleted
  WHERE A.a1 = deleted.a1

END
```

***Recherche dans les requêtes***

*A) Ajout d'un lien entre A et B*

Supposons que l'on affecte `var-b1` à la clé étrangère de la ligne de `A` identifiée par `var-a1`.

Pour la précondition, on aura la requête :

```
EXEC SQL SELECT * FROM C WHERE a1 = :var-a1  END-EXEC
```

Si la requête a trouvé quelque-chose (SQLCODE = 0), on peut faire l'insertion d'un lien A-B. Sinon, il y aura une gestion d'erreur.

Pour l'IMPLIES, on aura la requête d'insertion d'un lien A-B et la requête d'insertion d'un lien A-C ensembles.

```
EXEC SQL  UPDATE A SET A.b1 = :var-b1 WHERE A.a1 = :var-a1  END-EXEC

EXEC SQL  UPDATE C.a1 SET C.a1 = :var-a1 WHERE ...  END-EXEC
```

var-b1 et var-a1 ne peuvent bien évidemment pas contenir la valeur nulle.

Ces deux requêtes peuvent être permutées.

*B) Suppression d'un lien entre A et C*

Supposons que l'on affecte une valeur nulle à la clé étrangère de la ligne de C identifiée par var-c1 et qu'on aie déjà vérifié qu'il n'y avait pas d'autres lignes de C qui avaient la même valeur de clé étrangère

Pour la précondition, on aura la requête :

```
EXEC SQL
      SELECT *
      FROM A,C
      WHERE A.a1 = C.a1 AND C.c1 = :var-c1  AND A.b1 IS NULL
END-EXEC
```

Si la requête a trouvé quelque-chose, il y aura une gestion d'erreur. Sinon, on fera la modification.

Pour l'IMPLIES, on aura la requête de suppression d'un lien A-B et la requête de suppression d'un lien A-C ensembles.

```
EXEC SQL  UPDATE C SET C.a1 = NULL WHERE C.c1 = :var-c1  END-EXEC

EXEC SQL
      UPDATE A SET A.b1 = NULL
      FROM C
      WHERE A.a1 = C.a1 AND C.c1 = :var-c1
END-EXEC
```

Ces deux requêtes peuvent être permutées.

*Conclusion*

On remarque qu'il y a une différence entre les triggers de vérification pour l'insertion et la suppression d'un lien. Le premier vérifie s'il existe un lien cible lorsqu'on ajoute le lien source. Le second vérifie s'il n'existe pas un lien source lorsqu'on supprime un lien cible. Grâce à de tels triggers, on peut donc déterminer qu'il y a une contrainte d'inclusion entre deux liens, et on peut aussi déterminer son sens. Le trigger d'action pour la suppression a comme condition de déclenchement la suppression d'un lien cible et il a comme action la suppression d'un lien source. On peut donc ici aussi déterminer la présence et le sens d'une contrainte d'inclusion.

En ce qui concerne les requêtes, on peut dire que si on trouve des insertions ou des suppressions de liens côte à côte (peu importe l'ordre dans lequel elles apparaissent), il y a peut-être une contrainte d'inclusion entre deux liens, mais on n'en connaît pas le sens. Pour les requêtes suivies d'une analyse du `SQLCODE`, la clé étrangère sur laquelle porte la requête est l'implémentation du lien cible dans le cas d'une insertion de lien, et du lien source dans le cas d'une suppression.

Le cas que nous avons traité peut être adapté si les connectivités des types d'associations one-to-many du schéma conceptuel de départ avait été différentes.

En effet, si les deux connectivités maximales 1 étaient du côté de `A`, les deux clé étrangères se trouveraient dans `A`. La gestion serait alors beaucoup plus simple. Il suffit de faire en sorte qu'à chaque fois que la clé étrangère référencant `B` est non-nulle, la clé étrangère référencant `C` soit non-nulle. Notons que dans ce cas, lors de l'insertion d'une ligne `a` dans `A`, la clé étrangère référencant `B` ne sera jamais à non-nulle alors que la clé étrangère référencant `C` est à nulle.

Si les deux connectivités maximales `N` étaient du côté de `A`, les deux clé étrangères seraient dans `B` et `C`, et alors les opérations développées ci-dessus seraient un peu plus complexes, mais le principe reste le même.

Si la connectivité maximale de `ro-1` avait été `N` et la connectivité maximale de `ro-2` avait été 1, les UC seraient quelque peu différentes mais le principe reste le même.

**Recherche dans l'extension**

Considérons le schéma de la figure 4.62, et supposons qu'il y a une contrainte d'inclusion de `ro-1` vers `ro-2`.

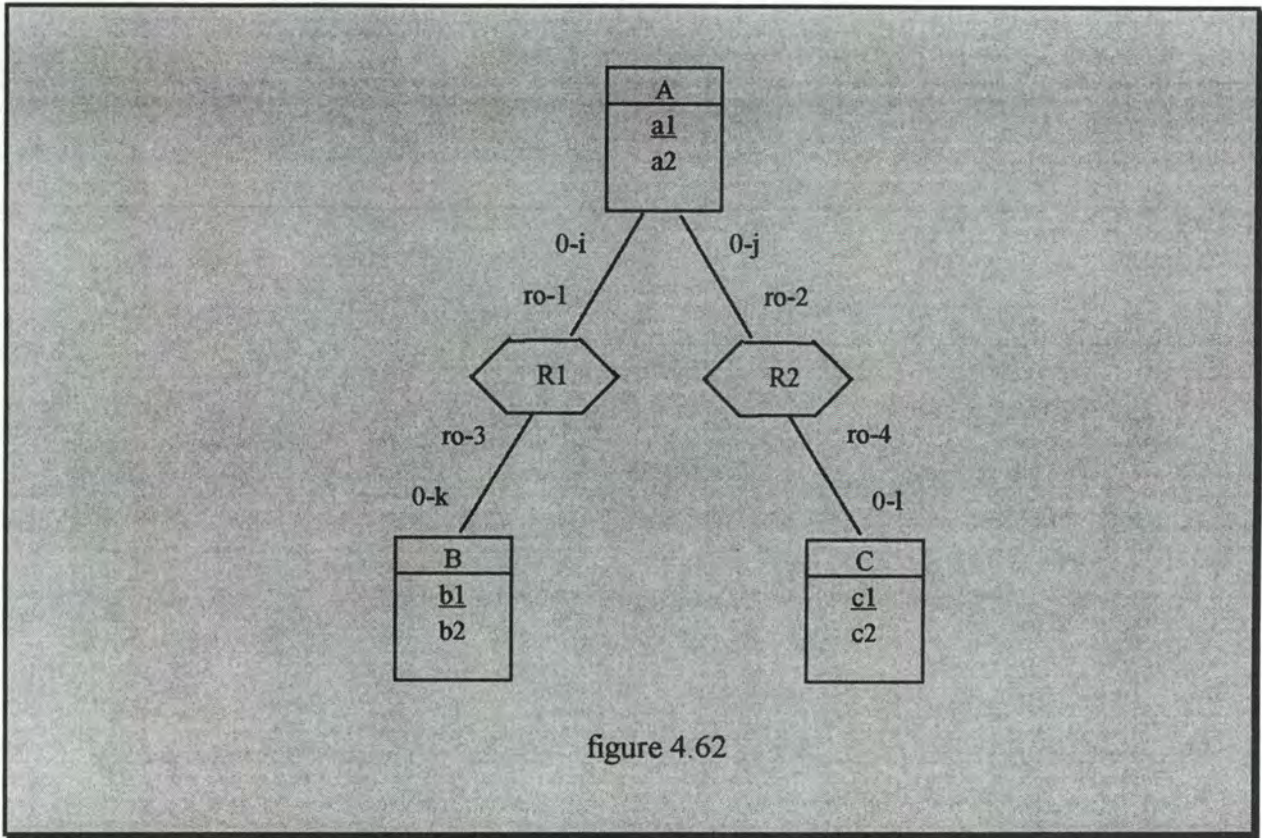



figure 4.62

Nous allons distinguer 4 cas, en fonction des connectivités maximales (en supposant que les types d'associations ont été traduit par des clés étrangères).

1) i = 1, j = N, k = N, l = 1

On peut exécuter la requête

```
SELECT * FROM A
        WHERE A.b1 IS NOT NULL AND
              NOT EXISTS (SELECT C.a1 FROM C WHERE C.a1 = A.a1)
```

Si la requête ne donne pas de réponse, on peut supposer qu'il y a une contrainte d'inclusion de ro-1 vers ro-2.

2) i = 1, j = 1, k = N, l = N

On peut exécuter la requête

```
SELECT * FROM A WHERE b1 IS NOT NULL AND c1 IS  NULL
```

Si la requête ne donne pas de réponse, on peut faire l'hypothèse qu'il y a une contrainte d'inclusion de `ro-1` vers `ro-2`.

3) `i= N, j=N,k=1,l=1`

On peut exécuter la requête

```
SELECT * FROM B WHERE a1 IS NOT NULL AND
                      a1 NOT IN (SELECT C.a1 FROM C)
```

Si la requête ne donne pas de réponse, on peut supposer qu'il y a une contrainte d'inclusion de `ro-1` vers `ro-2`.

4) `i = N , j = 1 , k = N , l = 1`

```
SELECT * FROM A
         WHERE A.c1 IS NOT NULL AND
               NOT EXISTS (SELECT B.a1 FROM B WHERE B.a1 = A.a1)
```

Si la requête ne donne pas de réponse, on peut faire l'hypothèse qu'il y a une contrainte d'inclusion de `ro-1` vers `ro-2`.

Remarque

Cette recherche dans l'extension n'est pas très coûteuse puisqu'on connaît déjà les liens entre tables.

#### 4.4.3.1.2 Contraintes d'égalité

Une contrainte d'égalité correspond à deux contraintes d'inclusion réciproques. On aura donc le même genre de recherches que pour une contrainte d'inclusion.

#### 4.4.3.1.3 Contraintes d'exclusion

**Définition**

Considérons le schéma de la figure 4.63.

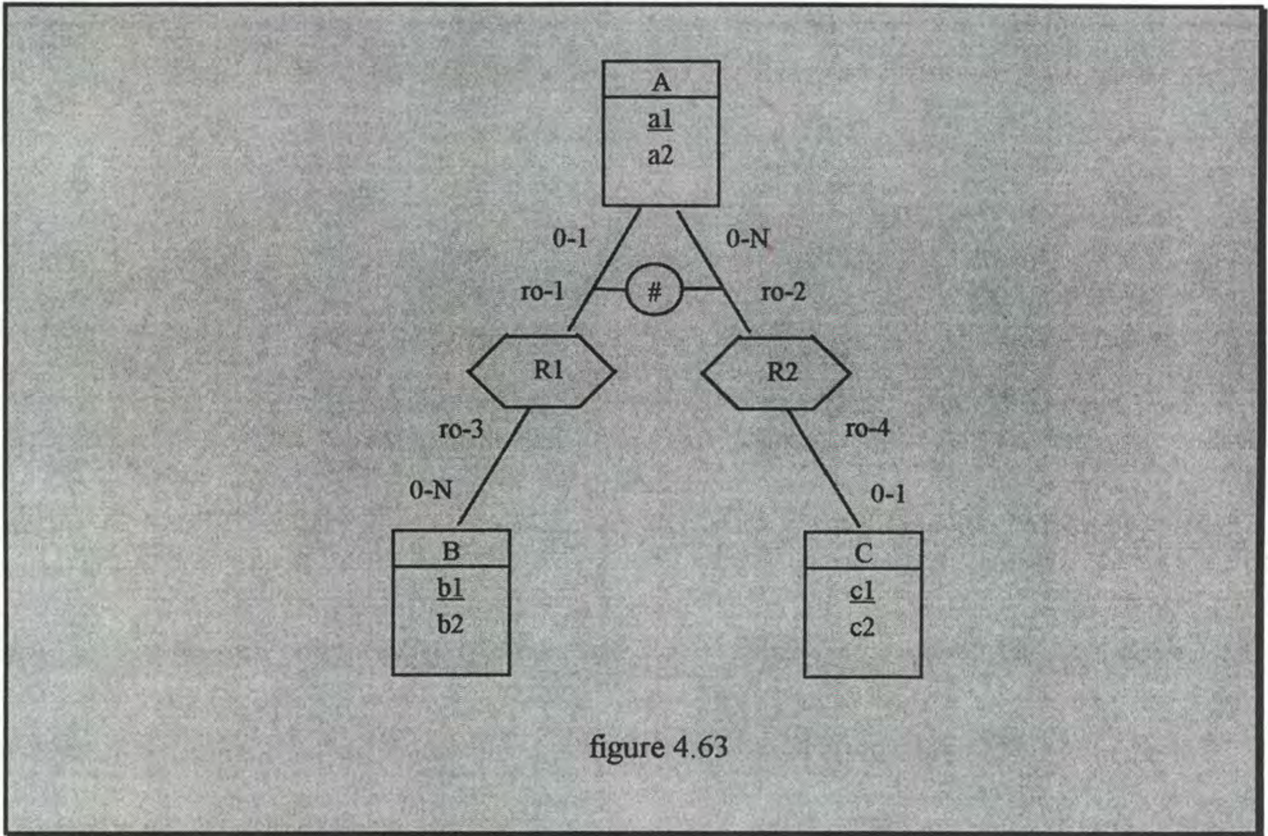



figure 4.63

La contrainte d'exclusion entre ro-1 et ro-2 exprime le fait qu'une occurence de A joue soit le rôle ro-1, soit le rôle ro-2, c-à-d que les rôles ro-1 et ro-2 sont mutuellement exclusifs.

La figure 4.64 donne la traduction en relationnel du schéma de la figure 4.63.

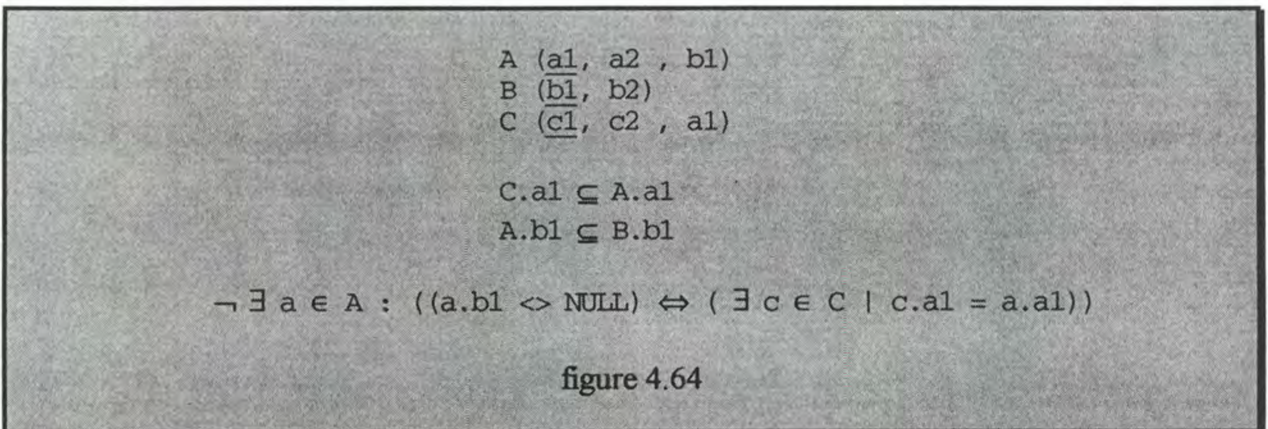



figure 4.64

**Recherche dans les checks**

On pourrait avoir un check attaché à la table A. Il serait de la forme :

```
CHECK ( NOT( (b1 IS NULL OR
             a1 IN (SELECT a1 FROM C))
             AND
             (a1 NOT IN (SELECT a1 FROM C) OR
              b1 IS NOT NULL) ) )
```

**Recherche dans les comportements imposés aux données**

Nous ne nous développerons que les UC.

*A) Ajout d'un lien entre A et B*

```
REPLACE a BY a' IN A | a[b1] = NULL AND a'[b1] <> NULL
ou bien PRECOND ¬∃ c ∈ C | c[a1] = a'[a1]
ou bien IMPLIES REPLACE ALL c | c[a1] = a'[a1] BY c' IN C | c'[a1] =
                                                              NULL
```

Comme il y a une contrainte d'exclusion entre A-B et A-C, l'ajout d'un lien entre A et B ne pourra se faire que s'il n'y a pas le lien correspondant entre A et C. Il se pourrait aussi que lors de l'ajout d'un lien entre A et B, on supprime le (les) lien(s) entre A et C correspondant(s).

*B) Suppression d'un lien entre A et B*

```
NO PROTOCOL
```

En effet, une contrainte d'exclusion exprime le fait qu'une ligne de la table A peut-être liée soit à une ligne de la table B, soit à une (ou plusieurs) ligne(s) de la table C mais pas aux deux en même temps, mais elle n'interdit pas qu'une ligne de A ne soit liée ni à une ligne de B ni à une ligne de C. Il n'y aura donc pas de comportement particulier à imposer aux données dans le cadre d'une suppression de lien.

*C) Modification d'un lien entre A et B*

```
NO PROTOCOL
```

*D) Ajout d'un lien entre A et C*

```
REPLACE c BY c' IN C | c[a1] = NULL AND c'[a1] <> NULL
                        AND ¬ ∃ c'' ∈ C | c'' <> c AND c''[a1] = c'[a1]
ou bien PRECOND ¬∃ a ∈ A | a[a1] = c'[a1] AND a[b1] <> NULL
ou bien IMPLIES REPLACE ALL a | a[a1] = c'[a1] BY a' IN A | a'[b1] =
                                                        NULL
```

*E) Suppression d'un lien entre A et C*

```
NO PROTOCOL
```

*F) Modification d'un lien entre A et C*

```
NO PROTOCOL
```

De la même façon que pour les contraintes d'inclusions entre rôles, il est possible de généraliser à d'autres configurations de connectivités maximales. Notons que si les deux connectivités maximales 1 étaient du côté de `A` (donc si les deux clés étrangères se trouvaient dans `A`), lors de l'insertion d'une ligne `a` dans `A`, les deux clés étrangères ne seraient jamais toutes les deux non nulles.

**Recherche dans l'extension.**

Considérons le schéma de la figure 4.62 et supposons qu'il y a une contrainte d'exclusion entre `ro-1` et `ro-2`. Nous allons distinguer 4 cas en fonction des connectivités (en supposant que les types d'associations ont été traduits par des clés étrangères).

1) `i= 1, j= N, k= N, l= 1`

```
SELECT * FROM A
         WHERE A.b1 IS NOT NULL AND
               EXISTS (SELECT * FROM C WHERE C.a1 = A.a1)
```

2) `i=1, j=1, k=N, l=N`

```
SELECT * FROM A WHERE (b1 IS NOT NULL AND c1 IS NOT NULL)
```

3) `i= N, j=N, k=1, l=1`

```
SELECT * FROM B WHERE a1 IS NOT NULL AND
                     a1 IN (SELECT C.a1 FROM C)
```

4) i = N, j = 1, k = N, l = 1

```
SELECT * FROM A
        WHERE A.c1 IS NOT NULL AND
             EXISTS (SELECT * FROM B WHERE B.a1 = A.a1)
```

Dans les 4 cas, si la requête ne donne pas de réponse, on peut supposer qu'il y a une contrainte d'exclusion entre ro-1 et ro-2.

### 4.4.3.1.4 Contraintes d'existence

**Définition**

Il peut arriver que l'on désire imposer une contrainte exprimant le fait que, parmi tous ses rôles (ou certains de ses rôles), une entité doit en jouer au moins un. Sur le schéma de la figure 4.65, une contrainte d'existence entre ro-1 et ro-2 signifie que A doit jouer soit ro-1, soit ro-2, soit les deux.

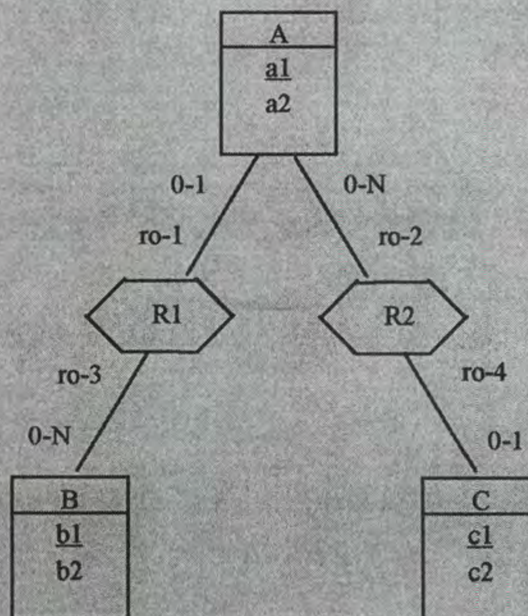



figure 4.65

La figure 4.66 donne la traduction en relationnel du schéma de la figure 4.65.

```
A (a1, a2 , b1)
B (b1, b2)
C (c1, c2 , a1)

C.a1 ⊆ A.a1
A.b1 ⊆ B.b1
∀ a ∈ A : ((a.b1 <> NULL) OR (∃ c ∈ C | c.a1 = a.a1))
```

figure 4.66

**Recherche dans les checks**

On pourrait avoir un check attaché à la table A. Il serait de la forme :

```
CHECK (b1 IS NOT NULL OR
        a1 IN (SELECT a1 FROM C))
```

Ce check vérifie que les lignes de A sont toutes liées à des lignes de C ou à des lignes de B. Il vérifie donc la contrainte d'existence.

**Recherche dans les comportements imposés aux données**

Nous ne développerons que les UC.

*A) Ajout d'un lien entre A et B*

```
NO PROTOCOL
```

*B) Suppression d'un lien entre A et B*

```
REPLACE a BY a' IN A | a[b1] <> NULL AND a'[b1] = NULL
ou bien PRECOND ∃ c ∈ C | c[a1] = a[a1]
ou bien IMPLIES REPLACE ANY c | c[a1] = NULL BY c' IN C | c'[a1] = a[a1]
```

*C) Modification d'un lien entre A et B*

```
NO PROTOCOL
```

*D) Ajout d'un lien entre A et C*

```
NO PROTOCOL
```

*E) Suppression d'un lien entre A et C*

```
REPLACE c BY c' IN C | c[a1] <> NULL AND c'[a1] = NULL
                    AND ¬ ∃ c'' ∈ C | c'' <> c AND c''[a1] = c[a1]
ou bien PRECOND (∀ a ∈ A | a[a1] = c[a1]) ⇒ a[b1] <> NULL
ou bien IMPLIES REPLACE a | a[a1] = c[a1] BY a' IN A | a'[b1] <>
                                                        NULL
```

Comme pour les autres contraintes entre rôles, il est possible de généraliser à d'autres configurations de connectivités maximales. Notons que si les deux connectivités maximales 1 étaient du côté de `A` (donc si les deux clés étrangères se trouvaient dans `A`), lors de l'insertion d'une ligne `a` dans `A`, au moins l'une des deux clés étrangères serait non nulle.

**Recherche dans l'extension**

Considérons le schéma de la figure 4.62 et supposons qu'il y a une contrainte d'existence entre `ro-1` et `ro-2`. Nous allons distinguer 4 cas en fonction des connectivités (en supposant que les types d'associations ont été traduits par des clés étrangères)..

1) `i= 1, j= N, k= N, l= 1`

```
SELECT * FROM A
         WHERE A.b1 IS NULL AND
               NOT EXISTS (SELECT * FROM C WHERE C.a1 = A.a1)
```

2) `i=1, j=1, k=N, l=N`

```
SELECT * FROM A WHERE b1 IS  NULL AND c1 IS  NULL
```

3) `i= N, j=N, k=1, l=1`

```
SELECT * FROM A
         WHERE NOT EXISTS (SELECT * FROM B WHERE B.a1 = A.a1) AND
               NOT EXISTS (SELECT * FROM C WHERE C.a1 = A.a1)
```

4) `i= 1, j= N, k= N, l= 1`

```
SELECT * FROM A
         WHERE A.c1 IS NULL AND
               NOT EXISTS (SELECT * FROM B WHERE B.a1 = A.a1)
```

Dans les 4 cas, si la requête ne donne pas de réponse, on peut supposer qu'il y a une contrainte d'existence entre `ro-1` et `ro-2`.

## 4.4.3.2 Les contraintes sur les types d'associations

On peut définir des contraintes d'inclusion, d'égalité et d'exclusion sur des types d'associations. Elles concernent alors tous les rôles du type d'associations. Nous allons distinguer les contraintes sur les types d'associations one-to-many des contraintes sur les types d'associations many-to-many.

### 4.4.3.2.1 Les contraintes sur les types d'associations one-to-many

**Définition**

Considérons le schéma de la figure 4.67.

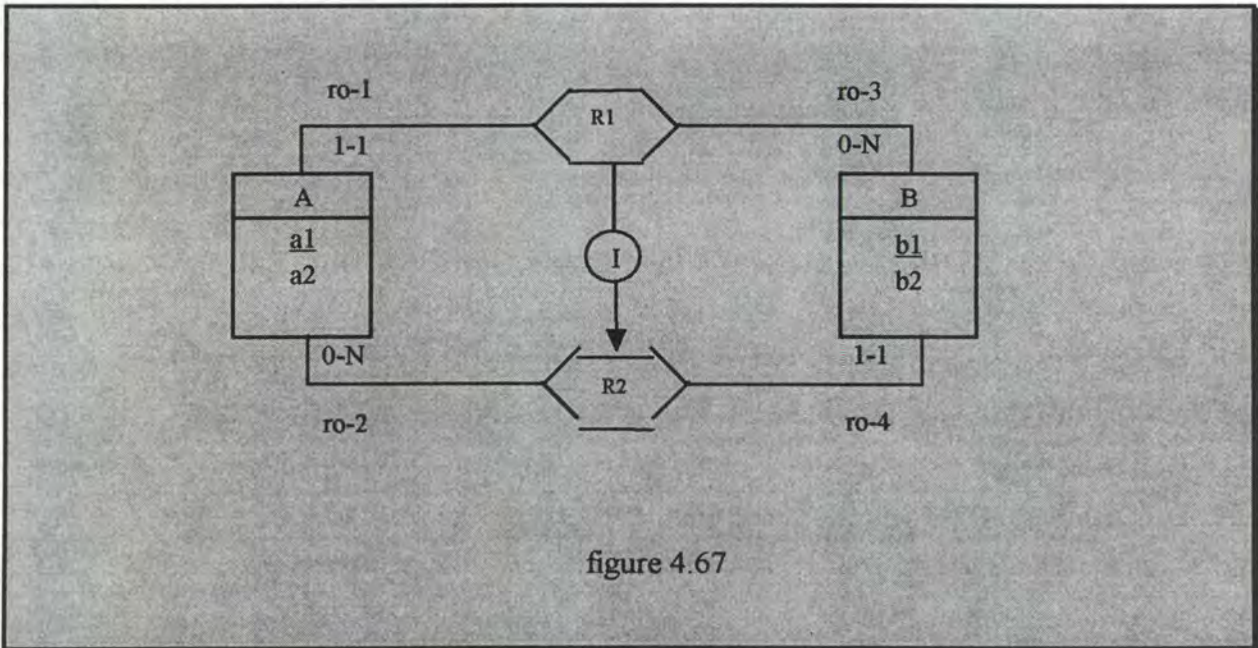



figure 4.67

On voit sur ce schéma qu'il y a une contrainte d'inclusion de `R1` vers `R2`. Nous n'envisageons que le cas d'une contrainte d'inclusion, les recherches concernant les contraintes d'égalité et les contraintes d'exclusion étant assez proches. Nous proposerons une recherche basée sur les UC et une recherche dans l'extension.

La traduction du schéma de la figure 4.67 en relationnel est donnée à la figure 4.68.

```
A(a1, a2, b1)
B(b1, b2, a1)

B.a1 ⊆ A.a1
A.b1 ⊆ B.b1
∀ a ∈ A , b ∈ B : (a[b1] = b[b1] ⇒ b[a1] = a[a1])
```

figure 4.68

**Recherche dans les comportements imposés aux données**

Remarque

Nous appellerons 'lien R1' le lien réalisé à l'aide de la clé étrangère A.b1., et 'lien R2' le lien réalisé à l'aide de la clé étrangère B.a1.

*A) Ajout d'un lien R1 entre A et B*

```
REPLACE a BY a' IN A | a[b1] = NULL AND a'[b1] <> NULL
ou bien PRECOND ∃ b ∈ B | b[b1] = a'[b1] AND b[a1] = a'[a1]
ou bien IMPLIES REPLACE b | b[b1] = a'[b1] AND b[a1] <> a'[a1]
                BY b' IN B | b'[a1] = a'[a1]
```

Lorsqu'on ajoute un lien R1 entre a ∈ A et b ∈ B, il faut qu'il y aie un lien R2 entre a et b. Soit on le vérifie, soit on ajoute ce lien.

*B) Suppression d'un lien R2 entre A et B*

```
REPLACE b BY b' IN B | b[a1] <> NULL AND b'[a1] = NULL
ou bien PRECOND ¬∃ a ∈ A | a[b1] = b[b1] AND a[a1] = b[a1]
ou bien IMPLIES REPLACE a | a[b1] = b[b1] AND a[a1] = b[a1]
                BY a' IN A | a'[b1] = NULL
```

Lorsqu'on supprime un lien R2 entre une ligne a ∈ A et une ligne b ∈ B, il ne faut pas qu'il y aie un lien R1 entre a et b. Soit on le vérifie, soit on supprime ce lien.

**Recherche dans l'extension**

Pour une contrainte d'inclusion de R1 vers R2, on peut appliquer la requête suivante :

```
SELECT *
FROM A , B
WHERE A.a1 = B.a1 AND B.b1 <> a.b1
```

Si la requête ne donne pas une réponse, c'est qu'il y a probablement une contrainte d'inclusion de R1 vers R2.

Remarque

Pour le cas où les deux clé étrangères seront dans la même table (soit A), la recherche dans l'extension sera plus simple. Il suffit d'examiner dans chaque ligne de A si la clé étrangère "cible" est toujours à la valeur de la clé étrangère "source" lorsque celle-ci est non nulle.

#### 4.4.3.2.2 Les contraintes sur les types d'associations many-to-many

Considérons le schéma de la figure 4.69.

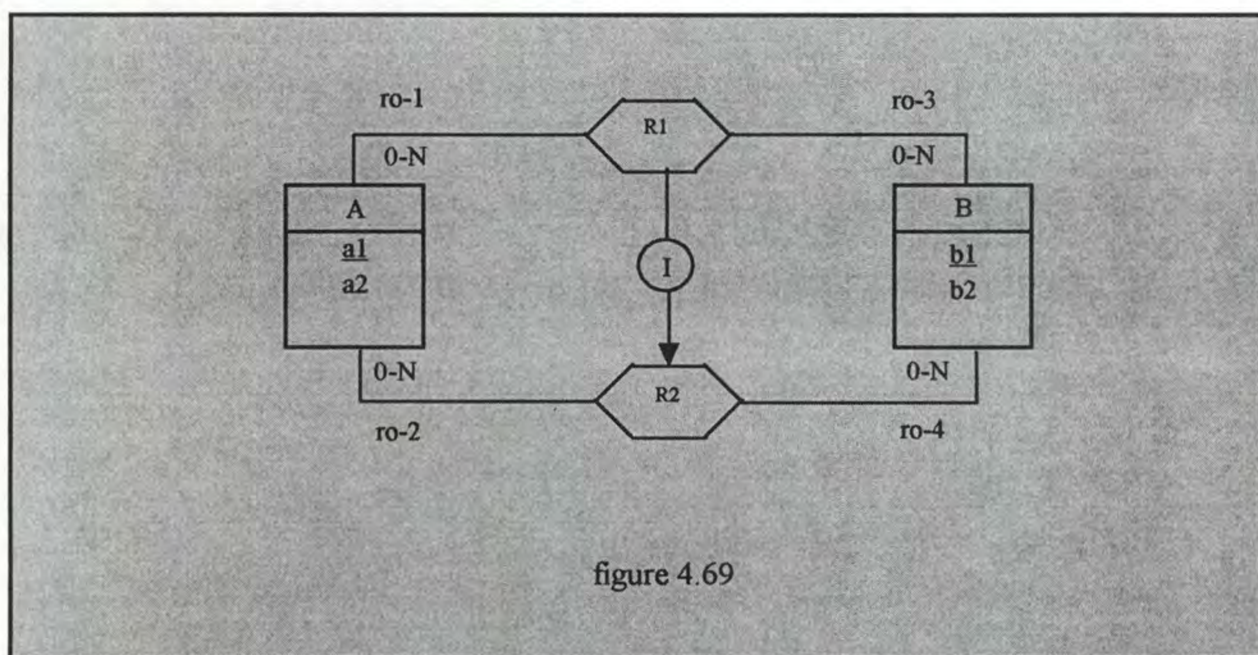



figure 4.69

Supposons que lors de la conception on ait supprimé les types d'association many-to-many de ce schéma de façon classique, en ajoutant deux types d'entités AB1 et AB2.

La traduction en relationnel du schéma ainsi obtenu est donnée à la figure 4.70.

```
A(a1, a2)
B(b1,B2)
AB1(a1, a2)
AB2(a1, a2)

AB1.a1 ⊆ A.a1
AB1.a2 ⊆ B.b1
AB2.a1 ⊆ A.a1
AB2.a2 ⊆ B.b1
AB1 ⊆ AB2
```

figure 4.70

On peut chercher des triggers ou des requêtes qui, lors de l'ajout d'une ligne dans AB1, vérifient qu'il existe la même ligne dans AB2 ou qui ajoutent cette ligne. On peut aussi chercher des triggers et des requêtes qui, lors de la suppression d'une ligne de AB2, vérifient qu'il n'existe pas la même ligne dans AB1, ou qui la suppriment si elle existe.

Pour la recherche dans l'extension, il suffit d'examiner si toute les lignes de AB1 se trouvent dans l'ensemble des lignes de AB2

### 4.4.4 Les relations IS-A

Nous avons déjà abordé les relations IS-A dans le tome 1. Nous ne les développerons pas ici. Elles ne font en effet pas l'objet de développements techniques.

# CHAPITRE 5

# 5. PROPOSITION D'UN OUTIL POUR L'ANALYSE DE TEXTES

Nous allons proposer ici un outil de support pour un utilisateur qui analyse un texte source. Cet outil permet d'effectuer diverses recherches. Il peut s'agir de la recherche d'une structure particulière de texte, mais également d'une recherche plus complète comme l'extraction des tables et des attributs impliqués dans une jointure présente dans un programme d'application. Pour le premier type de recherche un simple outil de **pattern-matching** suffira. Pour le deuxième, l'utilisateur devra décrire la façon dont il veut que l'extraction se fasse dans un **script**.

Un outil de **pattern-matching** est un outil permettant de définir des **patterns** et d'exécuter une recherche de ces patterns dans un texte source.

Un **pattern** est une structure décrivant une partie de texte de façon "générique". Certains composants du pattern peuvent être fixés, d'autres pas. On pourrait par exemple vouloir chercher dans un texte tous les débuts de phrase de la forme "La ville de *nom-ville*", *nom-ville* représentant un nom quelconque de ville. Pour cela, on peut définir un pattern composé de la chaîne fixe de caractères "La ville de", suivie d'un symbole représentant une chaîne de caractères non fixée.

Un **script** est un algorithme décrivant les recherches à effectuer sur le texte source. Un script permet d'enchaîner les recherches de différents patterns.

Nous proposons dans ce chapitre un atelier d'analyse de texte. Cet atelier comprend des outils d'édition et d'exécution de patterns et de scripts. La première partie de ce chapitre décrit l'environnement de cet atelier. La deuxième partie décrit un langage de définition de patterns. On trouvera dans la troisième partie la description de quelques instructions utiles à la définition d'un script.

## 5.1 ATELIER

Nous allons décrire succintement quelle pourrait être la structure d'un atelier d'analyse de textes. L'atelier proposé pourrait, une fois développé, être intégré à l'outil CASE DB-MAIN développé à l'Institut. Les seules fonctions de cet atelier que nous avons implémentées sont les fonctions de pattern matching décrites au point 5.3.

L'environnement de l'atelier est constitué d'une fenêtre mère, de fenêtres filles, et de menus.

- La fenêtre mère est destinée à contenir les fenêtres de travail (fenêtres filles) dans un même espace.
- Les fenêtres filles sont des fenêtres d'édition. Il y en a de 4 types : (1) édition de pattern, (2) édition de scripts, (3) édition d'un texte source (une seule fenêtre), (4) affichage des résultats (une seule fenêtre). Le titre de ces fenêtres sera préfixé par un nom permettant à l'utilisateur de faire la distinction.

- Les menus

**Menu Fichier**

Nouveau → ouverture d'une fenêtre pour l'édition d'un nouveau pattern / script / texte source

Ouvrir → ouverture d'une fenêtre pour l'édition d'un pattern / script/ texte source existant

Enregistrer → enregistrement d'un pattern / script / texte source.

Enregistrer sous → enregistrement d'un pattern / script / texte source sous un nouveau nom

**Menu Script**

Exécuter → exécution d'un script

Arrêter exécution → arrêt de l'exécution

**Menu Pattern**

Chercher → recherche dans le texte source de la première occurrence du pattern et coloriage du texte trouvé

L'atelier permet donc de créer des scripts et d'examiner, éditer et chercher des patterns. L'exécution d'un script peut fournir deux feed-back à l'utilisateur.

- Le coloriage de certaines zones du texte source
- L'écriture de certaines informations dans la fenêtre output

# 5.2 LANGAGE DE DEFINITION DE PATTERNS

Nous proposons ici un langage de définition de patterns. Ce language utilise une notation proche de la notation BNF.

La recherche dans un texte source d'un pattern défini à l'aide de ce langage sera réalisée à l'aide de la fonction UNIX `regex` (adaptée à DOS). La notation du langage proposé est plus abstraite que celle de la fonction `regex`. Ce langage est donc plus facile à utiliser. Nous verrons plus loin que la partie principale de notre travail d'implémentation a été de mettre au point des procédures de traduction qui permettent de passer du langage proposé à la notation `regex`.

Un pattern est défini comme suit :

```
pattern ::= définition
```

où `pattern` est le nom du pattern, et `définition` est sa définition.

La définition d'un pattern se compose de segments. Il y a essentiellement deux types de segments : des **segments terminaux** et des **segments non-terminaux**.

Un **segment terminal** est un segment de texte qui doit se retrouver tel quel dans le texte source[15]. Un segment terminal est entouré de guillemets (ainsi, le segment `"bonjour"` correspondra au mot `bonjour`). S'il y a des guillemets dans un segment terminal, l'utilisateur doit les faire précéder de guillemets, afin d'indiquer que ce n'est pas la fin du segment terminal (autrement dit, deux guillemets à l'intérieur d'un segment terminal signifient un guillemet).

Un **segment non-terminal** est un segment qui est défini ailleurs que dans la définition dans laquelle il est utilisé. Dans l'exemple ci-dessous, on voit que `mot` dans `pattern` est un segment non-terminal, puisqu'il est défini ailleurs.

```
mot ::= "bonjour";
pattern ::= mot;
```

[15]Un segment terminal peut aussi être une expression dans la notation `regex`. Il suffit pour cela de le faire précéder par '/g'. Cette possibilité permet à l'utilisateur connaissant la notation `regex` de définir certaines patterns de façon plus concise.

Le nom d'un segment non-terminal doit comporter au moins un caractère. On peut utiliser tous les caractères sauf les suivants : '(' , ')' , '{' , '}' , '[' , ']' , ':' , '=' , ';' , '"' , '@' , '|', '*' , ' ' , '\n' , '\t' , '/'. Il n'y a pas de limite de taille pour le nom. On ne peut pas utiliser le nom 'RANGE', qui est un mot réservé du language.

Un segment non-terminal doit toujours être défini avant la définition dans laquelle il est utilisé. Il est défini de la même façon qu'un pattern. A la fin d'une définition il faut un point virgule. Les définitions récursives sont interdites.

Voyons maintenant les symboles spéciaux du language :

- un segment entre crochets est un segment facultatif. Ainsi, `"bon"["jour"]` correspondra tant à `bonjour` qu'à `bon` ;
- des accolades permettent de définir un choix entre plusieurs possibilités, séparées par le symbole |. Ainsi `{"bon" | "jour"}` correspondra à `bon` ou à `jour` ;
- pour indiquer une étendue, on utilise l'expression `range(...)`. Ainsi,

  `range(a-z)` correspondra à n'importe quel caractère compris entre a et z.
- un segment entre des parenthèses suivies directement d'un astérisque est un segment répétitif. Cela signifie qu'il peut se retrouver plusieurs fois consécutivement dans le texte correspondant au pattern. Par exemple, `("a")*` correspondra à `a`, mais aussi à `aa`, `aaa`, etc.
- le symbole @ indique que le segment qui suit directement est une variable. Par exemple, si `var_1` est défini de la façon suivante : `var_1 ::= {range(a-z)}`, alors `"bon"@var_1"jour"` matchera `bonajour` ou `bonbjour` ou `boncjour`, et ainsi de suite en utilisant toutes les lettres comprises entre `a` et `z`. Si on trouve deux fois la même variable dans une définition, elle correspondra au même morceau de texte. A la fin de la recherche d'un pattern, on peut connaître la valeur des variables. Une variable ne peut pas se trouver à l'intérieur d'une structure répétitive.
- on trouve encore trois symboles dans le language. Ces symboles ne peuvent être utilisés qu'à l'intérieur d'un segment terminal.
  - le `\t` qui signifie une tabulation
  - le `\n` qui signifie un passage à la ligne

Pour écrire le caractère "\" , il faut faire précéder d'un autre \. Ainsi "`\\t`" ne correspond pas un \ suivi d'une tabulation mais correspond à "`\t`".

Notons que s'il doit y avoir des espaces entre deux segments, l'utilisateur doit le spécifier explicitement.

Remarque

Les définitions des segments non-terminaux peuvent se trouver dans des librairies. Les librairies permettent à l'utilisateur de ne pas devoir redéfinir des morceaux non-terminaux communs à plusieurs textes de définitions de patterns.

Par exemple, l'utilisateur pourrait définir deux symboles d'espacement :

- Le '-' , qui signifie qu'il doit y avoir 1 ou plusieurs espaces entre les segments entre lesquels il se trouve.

```
- ::= ({" "|"\n"|"\t"})*;
```

- Le '~' , qui signifie qu'il peut y avoir des espaces entre les segments entre lesquels il se trouve.

```
~ ::= ([{" "|"\n"|"\t"}])*;
```

Nous utiliserons ces symboles dans les exemples ci-dessous.

## Exemple

Considérons l'expression suivante :

```
select NOM , DATE
from CLIENT , COMMANDE
where CLIENT.NCLI = COMMANDE.NCLI
```

Cette requête nous indique qu'il pourrait y avoir une contrainte référentielle de `commande` vers `client`. On pourrait essayer de chercher cette expression telle quelle, mais cela serait très restrictif. On pourrait aussi rechercher une expression de ce genre là, mais sans fixer les tables et les attributs. On aurait alors les définitions suivantes :

```
mot ::= ({range(a-z) | range(A-Z) | range(0-9)| "-"})*

attribut ::= mot ;

table ::= mot ;

att_1 ::= attribut ;

att_2 ::= attribut ;

contrainte_ref ::= "select"- [table"."]attribut ~ "," ~
                              [table"."]attribut -
                    "from" - table ~ "," ~ table -
                    "where" - table"."@att_1  ~ "=" ~ table"."@att_2
```

`attribut` est défini comme une suite quelconque continue de minuscules, de majuscules, de chiffres ou de '-'. En utilisant `attribut` dans la définition de `contrainte_ref`, on trouvera donc toutes les requêtes semblables à celle définie ci-dessus, mais sans se préoccuper des deux attributs sélectionnés. Les attributs de la condition de jointure sont des variables. Cela permet d'extraire leur valeur.

Notons que cet exemple est encore très restrictif. En effet, la requête peut prendre plusieurs autres formes. On pourrait faire un pattern encore plus générique (on pourrait prendre en compte les alias, la présence de conditions de sélection supplémentaires, etc.).

# 5.3 DEFINITION DE SCRIPTS

Un script permet à l'utilisateur de décrire la façon dont il faut examiner un texte source. Il sera interprété lors de son exécution. Les instructions du script peuvent prendre diverses formes, en fonction du langage de programmation utilisé. Dans DB-MAIN, ce langage sera VOYAGER-2. Nous n'allons pas expliquer la syntaxe de ce langage, mais nous allons expliquer les trois fonctions que nous avons implémentées. On trouvera les codes sources de ces fonctions en annexe.

- **ouvrir (fich_pat , fich_lib , struct_pat)**

  Cette fonction permet de spécifier le fichier de patterns (`fich_pat`) et le fichier librairie (`fich_lib`). `struct_pat` référence la structure dans la laquelle se trouvent les définitions de patterns. Cette fonction rend un code d'erreur.

- **fermer (struct_pat)**

  Cette fonction libère la mémoire référencée par `struct_pat`.

- **matchfwd (struct_pat , source , debut , fin , nom_pattern , tab_inst , nb_var_inst , tab_a_inst , nb_var_a_inst )**

  Cette fonction a pour objectif de rechercher dans le texte `source` entre les positions `debut` et `fin` la première occurrence du pattern de nom `nom_pattern`. `tab-inst` est un

tableau contenant nb_var_inst variables instanciées (c-à-d celles dont on fixe la valeur). tab_a_inst est un tableau contenant nb_var_a_inst variables à instancier (c-à-d celles dont on veut connaître la valeur). La fonction matchfwd renvoie une position dans le texte source. Il s'agit de la position du premier caractère qui suit la partie de texte correspondant au pattern. Des valeurs négatives signifient qu'il y a eu une erreur.

Remarque

La traduction de la définition des patterns dans notre langage en définition conforme à regex est réalisée en partie dans la fonction ouvrir et en partie dans la fonction matchfwd. C'est la fonction matchfwd qui fait l'appel à regex.

**Exemple de script**

Considérons les expressions suivantes :

```
exec sql
select ncli into :x
from commande
where ...
end-exec

... move x to z ...

exec sql
select localite into :y
from client
where num = :z
end-exec
```

Une telle structure correspond à une jointure implicte entre les tables commande et client sur les attributs commande.ncli et client.num

Pour retrouver une telle structure dans un texte source, nous allons définir un script (en pseudo-code). Ce script utilisera les patterns suivants :

```
mot ::= ({range(a-z) | range(A-Z) | range(0-9)| "-"})*
att1 := mot ;
var ::= mot ;
var2 ::= mot;
table1 ::= mot ;
table2 ::= mot ;
alias := mot;
select ::= "select" - @att1 - "into" - ":"@var "- "from" - @table1;
move ::= "move" - @var - "to" - @var2;
select2 ::= "select" any "from" - @table2 - [@alias] - "where" any
                [{@table2|@alias}].@att2 ~ "=" ~ ":"@var2 ;
```

Remarque : ANY est un segment correspondant à une suite quelconque de caractères.

**Script**

```
procedure recherche-premier-select

        remplir tableau des variables à instancier avec
        "att1", "var" et "table1";

        position1 := matchfwd(struct-pat , texte-source, position ,
        fin-texte-source , "select" , tab-var-instanciees ,
        0 , tab-var-a-instancier , 3 );
```

```
procedure recherche-move

        remplir tableau des variables instanciées avec "var" ;

        remplir tableau des variables à instancier avec "var2" ;

        position2 := matchfwd(struct-pat , texte-source, position ,
        fin-texte-source ,  "move" , tab-var-instanciees , 1 ,
        tab-var-a-instancier , 1 );
```

```
procedure recherche-deuxieme-select

        remplir tableau des variables instanciées avec "var2" ;

        remplir tableau des variables à instancier avec "att2" ;

        position3 := matchfwd(struct-pat , texte-source, position ,
        fin-texte-source ,  "select2" , tab-var-instanciees , 1 ,
        tab-var-a-instancier , 1 );
```

```
recherche-structure

ouvrir ("fich-patterns" , "fich-librairie" , struct-pat) ;

position := 0 ;
recherche-premier-select ;

tant que (position1 <> fin-texte-source) et (position1 >= 0)
faire
|    position := position1 ;
|    recherche-move ;
|
|    si (position2 <> fin-texte-source) et (position2 >= 0)
|    alors
|    |
|    |    position := position2 ;
|    |    recherche-deuxieme-select ;
|    |
|    |    si (position3 <> fin-texte-source) et (position3 >= 0)
|    |    alors
|    |    |
|    |    |    ecrire ("jointure implicite entre les tables : " ,
|    |    |    table1 , " " , table2 , "sur les attributs " ,
|    |    |    att1 , " et " , att2)  ;
|    |    |
|    |    |    position := position3 ;
|    |    |
|    |    |    recherche-premier-select ;
|    |    |
|    |    sinon
|    |
|    |    |    position := position2 ;
|    |    |    recherche-premier-select ;
|    |
|    sinon
|
|    |    position := position1 ;
|    |    recherche-premier-select
|
fermer (struct-pat) ;

terminaison programme
```

On cherche successivement le premier select, le move et le deuxième select. Si on a trouvé une sucession de morceaux de texte correspondant à ces trois patterns, on affiche les résultats. Lorsqu'on n'a pas trouvé un des trois patterns, on recommence la recherche du premier. On recommence le processus jusqu'à ce qu'on arrive à la fin du texte source.

# 6. REFERENCES

**[AND94]** : M. Andersson, "Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering", EPFL, 1994

**[BAT92]** : C. Batini, C. Ceri, S.B. Navathe, "Conceptual Database Design", Benjamin/Cummings, 1992

**[BEL93]** : J-F. Bellem et X. Deflorenne, "Rétro-ingénierie de bases de données relationnelles et CODASYL", Mémoire de Licence et Maîtrise en Informatique, FUNDP, 1993

**[BOD89]** : F. Bodart et Y. Pigneur ,"Conception Assistée des Systèmes d'Information", Ed. Masson, 1989

**[CAR83]** : C.R. Carlson et A.K. Arora, "UPM : a formal tool for expressing database update semantics", in Proc. 3rd Entity-Relationship Conference, North Holland, 1983

**[CAS93]** : M.G. Castellanos, "Semiautomatic semantic enrichment for the integrated acces in interoperable databases", Thèse de Doctorat, Université Polytechnique de Catalogne, 1993

**[CHE76]** : P.P. Chen, "The Entity-Relationship Model : Toward a Unified View of Data", ACM TODS, vol. 1, n°1, 1976

**[COD70]** : E.F. Codd, "A Relationnal Model of Data for Large Shared Data Banks", Comm. ACM, vol. 13, N° 6, Juin 1970

**[DAV85]** : K.H. Davis, K.A. Adarsh, "A Methodology for Translating a Conventional File System into an Entity-Relationship", in Proc. of E/R Approach, 1985

**[DAV88]** : H.K. Davis et A.K. Arora, "Converting a Relationnal Database Model into an ER model", in Proc. 6th Int. Conf. on Entity-Relationship Approach, North Holland, 1988

**[FON92]** : M. Fonkam et W.Gray, "An approach to eliciting the semantics of relationnal databases", in Proc. CAISE 1992

**[HAI89a]** : J-L. Hainaut, "Bases de données et bases de connaissances en gestion des organisations", Cinquième école d'automne de bases de données, Port Barcarès, 1989

**[HAI89b]** :J-L. Hainaut,"Introduction à la Théorie Relationnelle des Bases de Données", FUNDP, Novembre 1989

**[HAI91]** : J-L. Hainaut, "Entity-generating Schema Transformation for Entity-Relationship Models", in Proc. 10th Conf. on Entity-Relationship Approach, San Mateo, 1991

**[HAI92]** : J-L. Hainaut, syllabus du cours "Conception et Technologie des Bases de Données", Première Licence et Maîtrise en Informatique (Année Académique 1992-1993), , FUNDP

**[HAI93a]** : J-L. Hainaut, M. Chandelon, C. Tonneau, M. Joris, "Contribution to a Theory of Database Reverse Engineering", IEEE Working Conference on Reverse Engineering, Baltimore, 1993

**[HAI93b]** : J-L. Hainaut, "Schema Transformations for Database Engineering, synthesis and illustration", FUNDP, 1993

**[JOH89]** : Johannesson et Kalman, "A Method for Translating Relationnal Schemas into Conceptual Schemas", in Proc. 8th Int. Entity-Relationship conference, Toronto, 1989

**[NAV88]** : S. Navathe, A. Awong, "Abstracting relationnal and hierarchical data with a semantic data model", in Proc. 6th Int. Conf. on Entity-Relationship Approach, North Holland, 1988, pages 305-333

**[NIL85]** : E.G. Nilsson, "The Translatiion of COBOL Data Structure to an Entity-Relationship Type Conceptual Schema", in proc. of ER Approach, 1985

**[PET94]** : J-M. Petit, J. Kouloumdjian, J-F. Boulicaut, F. Toumani, "Using Queries to Improve Database Reverse Engineering", proposé pour la conférence E/R, Manchester, 1994

**[PHE93]** : PHENIX Project, "Database Reverse Engineering", second version, BIKIT / FUNDP, 1993

**[PRE93]** : W.J. Premerlani, M.R. Blaha, "An approach for reverse enginerring of relationnal databases", IEEE Working Conference on Reverse Engineering, Baltimore, 1993

**[SPA]** : Stefano Spaccapietra, Christine Parent, "Intégration de vues et relativisme sémantique", EPFL.

**[WIN90]** : J. Winans, K.H. Davis, "Software Reverse Engineering From a Currently Existing IMS Database to an Entity-Relationship Model", in Proc. of E/R approach, 1990

# ANNEXES

# ANNEXE 1
# LE MODELE RELATIONNEL

Cette annexe est un inspirée de [HAI89b] et [BEL93].

Le **modèle relationnel** est proposé pour la première fois par Codd en 1970 [COD70]. Il s'agit d'un modèle original de représentation de l'information puisqu'il se base sur le concept mathématique de **relation**. Ainsi, il permet de représenter la réalité de façon simple et formelle.

Ce modèle a suscité un intérêt sans cesse croissant, non seulement dans la communauté scientifique qui lui a dédié bon nombre d'articles, mais également dans le monde des entreprises puisque les SGBD qui s'en sont inspirés (càd les **SGBD relationnels**) sont parmi les plus utilisés aujourd'hui.

On peut diviser ce modèle en 3 parties : une partie structurelle décrivant les **concepts de base** du modèle, une partie composée des **contraintes d'intégrité** imposée aux données, et une partie de manipulations reprenant les différents **opérateurs** applicables aux données et à leurs structures.

## 1.1 LES CONCEPTS DE BASE

### *1.1.1 Les domaines simples*

Un **domaine** est un **ensemble d'informations**, désigné par un nom unique, représentant un même concept du monde réel. On appelle **valeur** un élément de cet ensemble. Par exemple un domaine pourrait être constitué d'informations désignant des employés, des véhicules, etc. provenant du réel perçu.

Dans le modèle relationnel on a des domaines simples où chaque valeur est **atomique** ce qui signifie qu'elle est indécomposable. Une valeur appartenant à un domaine simple est donc la plus petite unité sémantique d'information.

### *1.1.2 Les relations*

Dans le monde réel, les éléments sont en associations mutuelles, c'est pourquoi le modèle relationnel permet de représenter de telles associations par des **n-uplets** (appelés aussi **lignes**) composés de valeurs. Par exemple si x désigne un client et y une entreprise, on peut représenter le fait que x est client de y par le n-uplet (x,y).

L'ensemble des associations de même nature est appelé **relation** (ou table) et possède un nom qui l'identifie. Notons qu'une relation met toujours en association des éléments provenant des mêmes domaines.

Supposons qu'on ait les domaines:

| | |
|---|---|
| `MAGASIN = {GB , Sarma , Colruyt , Match}` | représentant des magasins |
| `PRODUIT = {sucre , farine , conserves}` | représentant des produits |

Un exemple de relation pourrait être le lien `VEND` entre des élément des domaines `MAGASIN` et `PRODUIT`. On peut définir le format général d'une telle relation par la notation suivante :

```
VEND (MAGASIN , PRODUIT)
```

L'ensemble des relations définies sous ce format s'appelle un **schéma relationnel**.

L'**extension d'une relation** est l'ensemble des lignes de cette relation à un instant déterminé. On peut la représenter à l'aide d'un tableau semblable à celui présent sur la figure A1.1.

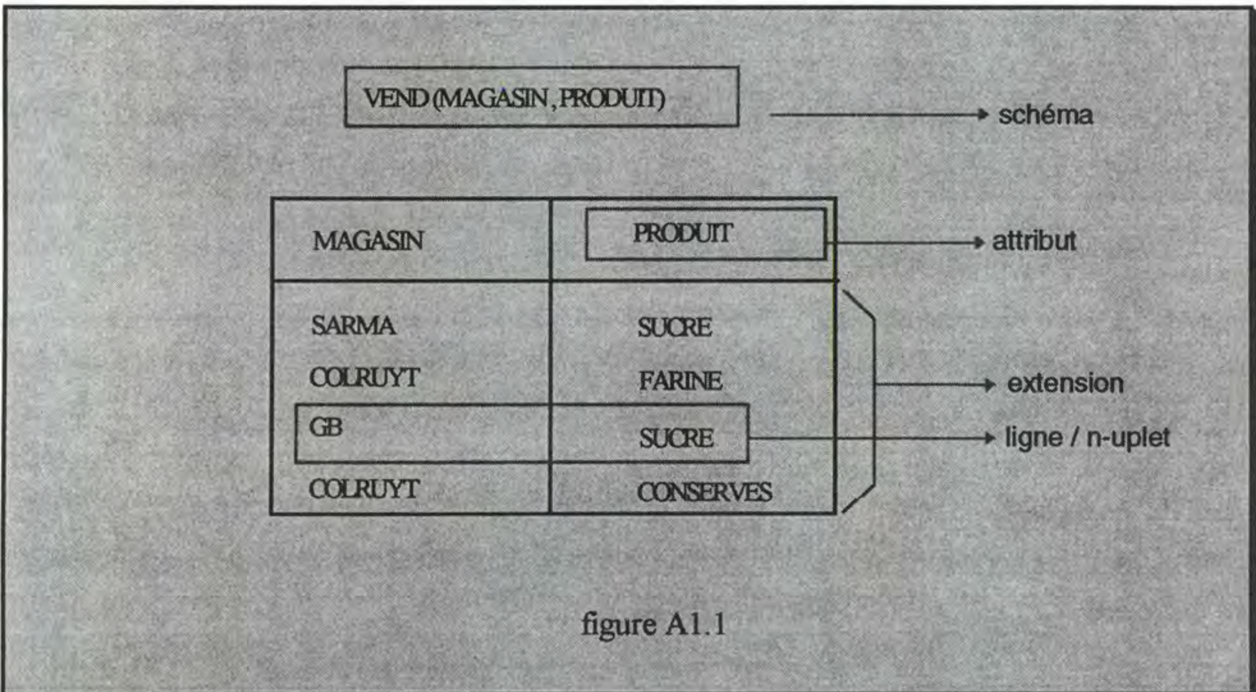



figure A1.1

Le modèle relationnel étant ensembliste, chaque n-uplet d'une relation doit être unique et il n'y a pas de relation d'ordre ni dans les n-uplets, ni dans les domaines participant à la relation.

### 1.1.3 Le concept d'attribut

Un domaine peut être utilisé dans plusieurs relations. On dit que dans chacune, il joue un rôle particulier. En général, désigner le nom d'un domaine revient à désigner son rôle dans la relation. Toutefois, lorsqu'un domaine est présent plusieurs fois dans une relation ce nom devient ambigü. Pour résoudre ce problème on utilise le concept d'**attribut**, qui désigne la participation d'un domaine à une relation comme y jouant un rôle défini.

Par exemple si on veut représenter un lien de filiation entre des éléments du domaine PERSONNE, nous définirons le schéma suivant :

```
FILIATION ( PARENT : PERSONNE, ENFANT : PERSONNE)
```

PARENT et ENFANT sont les attributs de la relation définis sur le même domaine PERSONNE.

## 1.2 LES CONTRAINTES D'INTEGRITE

En plus de la structure des relations, il est utile de préciser certaines propriétés que les données doivent satisfaire afin d'augmenter la sémantique et la représentativité du réel perçu du schéma relationnel. Il y a deux types différents de Contraintes d'Intégrité (CI) : celles qui sont définies par l'utilisateur et qui portent sur un schéma spécifique et celles communes à tout schéma. Nous allons nous intéresser, parmi ces dernières, au contraintes de dépendance et d'inclusion.

### 1.2.1 Les dépendances

Nous allons tout d'abord examiner les identifiants pour aborder ensuite une contrainte plus générale, la dépendance fonctionnelle, et nous terminerons par la dépendance multivaluée dont la dépendance fonctionnelle est un cas particulier.

#### 1.2.1.1 Identifiant d'une relation

L'**identifiant d'une relation** (ou clé de la relation) est composé d'un sous-ensemble d'attributs d'une relation pour lesquels toutes les valeurs sont distinctes les unes des autres. On peut donc déjà dire que dans le modèle relationnel l'ensemble des attributs d'une relation est identifiant de celle-ci. L'identifiant est un moyen permettant de désigner chaque ligne sans ambiguïté. Comme il n'est pas toujours facile d'utiliser l'ensemble des attributs à cet effet, nous nous intéresserons seulement aux **identifiants stricts**, qui sont tel qu'aucun de leurs sous-

ensembles d'attributs n'est aussi identifiant. Les identifiants stricts seront notés par la suite par le soulignement de leurs attributs dans le schéma relationnel.

Une relation peut avoir plus d'un identifiant. Si tel est le cas, un de ceux-ci est désigné comme **identifiant primaire**. Les autres sont les **identifiants candidats**.

### 1.2.1.2 Dépendances fonctionnelles

Une **dépendance fonctionnelle** définie sur une relation R (DF, notée →) est une contrainte qui spécifie qu'une valeur d'un attribut X (ou d'un ensemble d'attributs X) détermine univoquement la valeur d'un autre attribut Y (ou ensemble d'attributs Y).

$X \rightarrow Y$ ssi $\forall$ paire de n-uplets t1, t2 $\in$ relation R :

$$t1.X = t2.X \Rightarrow t1.Y = t2.Y$$

Les seule DF intéressantes sont celle appelées DF 'élémentaires' :

$X \rightarrow Y$ est élémentaire ssi Y est un attribut simple, X est minimal et $X \rightarrow Y$ est non-triviale

($X \rightarrow Y$ est triviale si $Y \subseteq X$)

### 1.2.1.3 Dépendances multivaluées

La **dépendance multivaluée** est une généralisation de la DF. Alors que dans le cas de la DF, le déterminant se voit toujours associer la même valeur du déterminé, dans le cas de la dépendance multivaluée, le déterminant se voit toujours associer le même sous-ensemble de valeur du déterminé.

### *1.2.2 Les contraintes d'inclusion*

Deux relations A(<u>a1</u>, a2) et B(<u>b1</u>, b2) font l'objet d'une contrainte d'inclusion de b2 vers a1 (notée B.b2 $\subseteq$ A.a1 ) si pour chaque valeur de b2, il existe une valeur identique de a1.

Une variante de cette contrainte est la contrainte d'égalité (notée B.b2 = A.a1) qui correspond à deux contraintes d'inclusion réciproques.

Lorsque a1 est identifiant on appelle la contrainte une contrainte référentielle et b2 est appelé clé étrangère. Un sous-ensemble FK d'attributs d'une relation R1 est une **clé étrangère** (ou foreign key en anglais) faisant référence à une relation R2 si les valeurs des attributs de la clé étrangère font référence aux valeurs des attributs de l'identifiant de R2. Par référence nous entendons que chaque valeur de FK est toujours identique à une valeur de l'identifiant de R2.

## 1.3 L'ALGEBRE RELATIONNELLE

Les domaines et relations étant des ensembles au sens mathématique du terme, on peut leurs appliquer la totalité des **opérateurs** ensemblistes et du calcul des prédicats. Rappelons que l'objectif du modèle est la représentation du réel perçu, ce qui nécessite des opérateurs adéquats.

Codd propose une algèbre relationnelle composée d'un jeu d'opérateurs équivalent au calcul des prédicats du premier ordre. Nous allons brièvement définir les principaux.

Soient A, B, C trois domaines et Y ( a : A, b : B) , Z ( a : A, c : C) deux relations. On peut appliquer les opérateurs suivants :

- L'**union** de deux ou plusieurs domaines (ou relations). Notation : A $\cup$ B.
- La **différence** de deux domaines ( ou relations). Notation : A \ B.
- Le **produit cartésien** de deux ou plusieurs domaines : ensemble des arrangements obtenus en prenant une valeur dans le premier domaine, puis une valeur dans le deuxième et ainsi de suite. Notation : A X B.
- Le **produit de relations** : relation qui possède les attributs de chacune des relations initiales. Ses n-uplets sont constitués de toutes les combinaisons possibles obtenues en associant un n-uplet de la première relation, un n-uplet de la deuxième et ainsi de suite. Notation : Y X Z ou prod(Y,Z).
- La **projection** d'une relation : relation obtenue en ne conservant de la relation initiale que les valeurs de certains attributs. L'extension d'une relation étant un ensemble, certaines lignes seront amenées à disparaître. Y [a] ou proj (Y; a) constitue la projection de la relation Y sur l'attribut a .

- La **sélection** : relation obtenue en ne retenant de la relation initiale que les n-uplets dont les valeurs vérifient des conditions du type : *La valeur de tel attribut appartient à tel ensemble.* Notation : `Y(a = 'valeur')` ou `select (Y ; a = 'valeur')`.

- La **jointure** de deux relations : permet de construire une relation en couplant les n-uplets de deux relations qui ont une même valeur pour un ou plusieurs attributs compatibles. Il s'agit en fait d'une composition des opérateur de projection, de sélection et de produit. Notation : `Y(a) * Z(a)` ou `Y * Z (a)` ou `join(Y,Z; a,a)`.

A partir de ces opérateurs il est possible de définir un **langage de définition et de manipulation** de domaines et de relations. Le langage **SQL** (Structured Query Language) est l'un des plus répandus à l'heure actuelle.

# 1.4 REMARQUE

Les concepts définis ci-dessus apparaissent sous un aspect purement **statique**. Cependant, le réel perçu **évolue**, et des objets apparaissent et disparaissent, des associations se nouent et se dénouent, et des propriétés changent d'état.

La notion d'ensemble au sens strict ne permet pas de prendre en compte cette évolution. En effet, on ne peut ni ajouter ni soustraire un élément à un ensemble au sens strict. Il faut alors considérer non plus un domaine (une relation), mais plutôt une suite temporelle de domaines (de relations) du même type. Cela complique le modèle, et dès lors, on admet (abusivement) d'appeler domaine (relation) tout ensemble d'éléments respectant les propriétés qui lui sont associées. On admet donc que l'on ajoute et que l'on retire un élément à un domaine ou une relation.

# ANNEXE 2
# LA REQUETE DE SELECTION SQL

Nous nous somme inspirés pour cette partie de [HAI92].

La commande SELECT permet d'extraire des données de l'ensemble des tables afin de les présenter à l'utilisateur final ou à un programme d'application qui en a demandé l'exécution. Nous allons voir les principales possibilités de cette commande.

## 2.1 PRINCIPE

Le SELECT produit un résultat qui est une table. Il est composé de trois parties :

- la clause SELECT donne les valeurs constituant chaque ligne du résultat ;
- la clause FROM fournit les tables mise en cause par l'extraction ;
- la clause WHERE précise les conditions de sélection que doivent satisfaire les lignes du résultat.

Il est en outre possible d'imbriquer une sous-requête dans une condition de sélection.

Les exemples qui suivent seront basés sur le schéma relationnel de la figure A2.1.

```
CLIENT(Num-cli, Nom, Localite, Num-tel)
COMMANDE(Num-cli, Num-pro, Quantite)
PRODUIT(Num-pro, Libelle, Prix, Quantite-stock)
```

figure A2.1

## 2.2 EXTRACTION ET SELECTION SANS SOUS-REQUETES

Examinons différentes possibilités à l'aide de l'exemple suivant :

```
SELECT  DISTINCT  PRODUIT.Libelle, (PRODUIT.Prix * 0.19)
FROM CLIENT CLI , COMMANDE COM , PRODUIT PRO
WHERE CLI.Localite IN ('Namur', 'Bruxelles', 'Liege')
AND CLI.Num-tel IS NOT NULL
AND PRO.Prix BETWEEN 100 AND 4000
AND COM.Quantite < = PRODUIT.Quantite-stock
AND CLI.Num-cli = COM.Num-cli
AND CLI.Num-pro = PRODUIT.Num-pro
```

Cette requête fourni le libellé et le prix des produits (TVA comprise) commandés par des clients de Namur, Bruxelles ou Liège qui ont un numéro de téléphone, dont le prix est compris entre 100 et 4000 et dont la quantité commandée et inférieure à la quantité de stock. On peut faire diverses remarques sur cette requête :

- la clause `DISTINCT` assure que toutes les lignes du résultat seront différentes ;
- il est possible de préciser des colonnes résultats qui sont dérivées des colonnes d'origine (on trouve par exemple dans la requête ci-dessus `Prix-prod * 0.19`) ;
- les tables `CLIENT`, `COMMANDE` et `PRODUIT` de la clause `FROM` sont suivies d'un libellé. Il s'agit d'un synonyme (alias) que l'on attribue à ces tables pour la requête ;
- si deux tables ont deux colonnes de même nom, on préfixe le nom de la colonne par le nom de sa table (ou de son alias) ;
- la clause `FROM` contient plusieurs tables : cela constitue un produit (au sens relationnel du terme) de ces tables. Pour effectuer une jointure, nous avons simplement mis en correspondance (par l'égalité), les clés étrangères avec les identifiants primaires correspondant (ici `Num-cli` et `Num-pro`) dans les conditions ;
- l'opérateur `IN` porte sur l'appartenance à une liste ;
- l'opérateur `IS NOT NULL` (`IS NULL`) permet de vérifier qu'une valeur est non nulle (est nulle).

# 2.3 EXTRACTION DE DONNEES AGREGEES

Examinons la requête suivante :

```
SELECT DISTINCT COUNT(*) FROM CLIENT WHERE Localite = 'Namur'
```

La fonction d'agrégation, `COUNT(*)` compte le nombre de clients de Namur.

Voici les principales autres fonctions d'agrégation :

- `AVG(`*`nom-colonne`*`)` donne la moyenne des valeurs de nom-colonne ;
- `SUM(`*`nom-colonne`*`)` donne la somme des valeurs de nom-colonne ;

- `MIN(nom-colonne)` donne la valeur minimale de nom-colonne ;
- `MAX(nom-colonne)` donne la valeur maximale de nom-colonne.

# 2.4 LES SOUS REQUETES

Il est possible d'imbriquer une requête dans une condition. On pourrait par exemple avoir :

```
SELECT Num-com
FROM COMMANDE
WHERE Num-cli IN (SELECT Num-cli
                  FROM CLIENT
                  WHERE Localite = 'Namur')
```

Cette requête donne les commandes des clients habitant à Namur.

Le résultat d'une sous requête peut être utilisé comme un ensemble. C'est pourquoi une série de quantificateurs ensemblistes sont présents dans SQL.

Le quantificateur `EXISTS(sous-requête)` est vérifié si la sous-requête fournit au moins une colonne.

Les quantificateurs `ANY` et `ALL` permettent de comparer une valeur à celles d'un ensemble. `ANY` signifie qu'au moins un élément satisfait à la condition, `ALL` que tout les éléments y satisfont.

# 2.5 EXTRACTION DE DONNEES GROUPEES

Il est possible de grouper certaines lignes du résultat à l'aide de la clause `GROUP BY`. La clause `HAVING` permet de ne sélectionner que certains groupes ainsi formés.

```
SELECT DISTINCT Nom
FROM CLIENT
GROUP BY Localite
HAVING COUNT(*) >=3
```

Cette requête fournit les noms des clients groupés par localités. La clause `HAVING` assure que le résultat ne contiendra que les localités qui ont plus de 3 clients.

# 2.6 ORDRE DU RESULTAT

On peut demander que le résultat soit trié sur une ou plusieurs colonnes. Pour ce faire on utilisera la clause `ORDER BY`. On utilisera les clauses supplémentaires `DESC/ASC` pour demander un ordre descendant ou ascendant.

```
SELECT *
FROM CLIENT
ORDER BY Localite (ASC)
```

Le résultat de cette requête sera trié de façon ascendante sur `Localite`.

# ANNEXE 3
# LE MODELE ENTITE/ASSOCIATION

Cette annexe est inspirée de [PHE93] et [BOD89].

Le **modèle** Entité/Association (E/A) a été proposé par Chen en 1976 [CHE76]. Ce modèle permet de représenter les structures conceptuelles sous-jacentes aux données, indépendamment de leur représentation physique et de leur utilisation. Il permet de représenter la réalité sous forme d'objets qui ont des propriétés et qui sont associés les uns aux autres. On trouve dans le modèle E/A utilisé dans ce travail les concepts d'**entités**, d'**attributs**, d'**associations** et de **spécialisations/généralisation**. On y trouve aussi des contraintes d'intégrité, qui sont des règles auxquelles les données doivent satisfaire.

Un **schéma** E/A est la spécification de données qui décrit une partie des données d'une application. Il se constitue de la spécification des **types d'entités**, des **types d'associations** et des **contraintes d'intégrité**.

## 3.1 ENTITES ET TYPES D'ENTITES

Une **entité** représente un objet du monde réel qui est distinct de tous les autres objets. Des entités similaires sont regroupées en un **type d'entités (T.E.)**, et elles forment la population de ce T.E. Par exemple, on peut avoir le type d'entités `CLIENT`, et une entité DUPONT, qui est une instance de `CLIENT`.

La représentation graphique d'un T.E. est donnée à la figure A3.1.

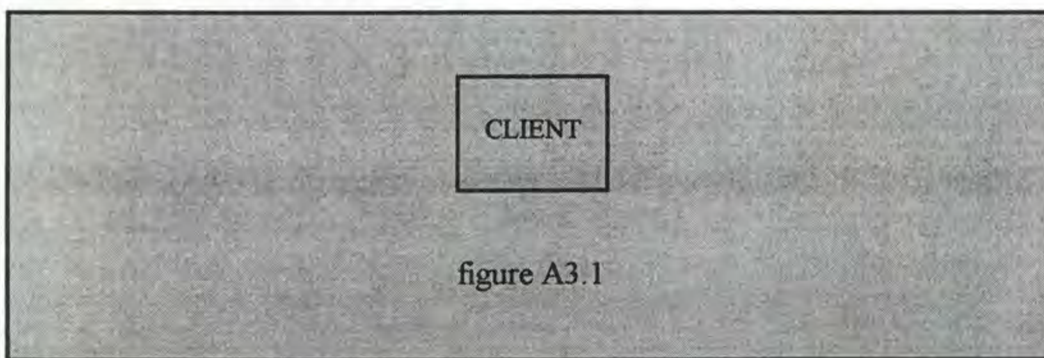


figure A3.1

## 3.2 SOUS-TYPES ET SUR-TYPES

Il peut exister des relations de **sous-typage/sur-typage** entre certains T.E. Voyons le sur un exemple. Soient les 2 T.E. `PARTICULIER` et `ENTREPRISE`, qui représentent respectivement les clients qui sont des particuliers et les clients qui sont des entreprises. A tout moment,

une instance de PARTICULIER ou de ENTREPRISE est aussi une instance de CLIENT. On dira qu'il existe une relation **IS-A** entre PARTICULIER et CLIENT, de même qu'entre ENTREPRISE et CLIENT.

La figure A3.2 donne la représentation graphique de cette relation IS-A.

Les T.E. PARTICULIER et ENTREPRISE sont appelés les sous-types du T.E. CLIENT, qui est leur sur-type. Une relation de sous-typage/sur-typage est aussi appelée relation de **spécialisation/généralisation**.

Deux contraintes intéressantes peuvent être posées sur un ensemble de T.E. liées par une relation IS-A. Il s'agit des contraintes de couvrance et d'exclusion.

Une contrainte d'exclusion exprime le fait qu'à une instance du sur-type ne peut correspondre qu'une instance d'un seul des sous-types. Sur le schéma de la figure A3.2, un client peut être une entreprise ou un particulier, mais pas les deux à la fois.

Une contrainte de couvrance exprime le fait qu'à une instance du sur-type doit correspondre au moins une instance de l'un des sous-types. Sur le schéma de la figure A3.2, un client doit être une entreprise ou un particulier.

Lorsqu'on a à la fois la couvrance et l'exclusion, on dit qu'on a une structure de partition. Le schéma de la figure A3.2 correspond donc à une telle structure.

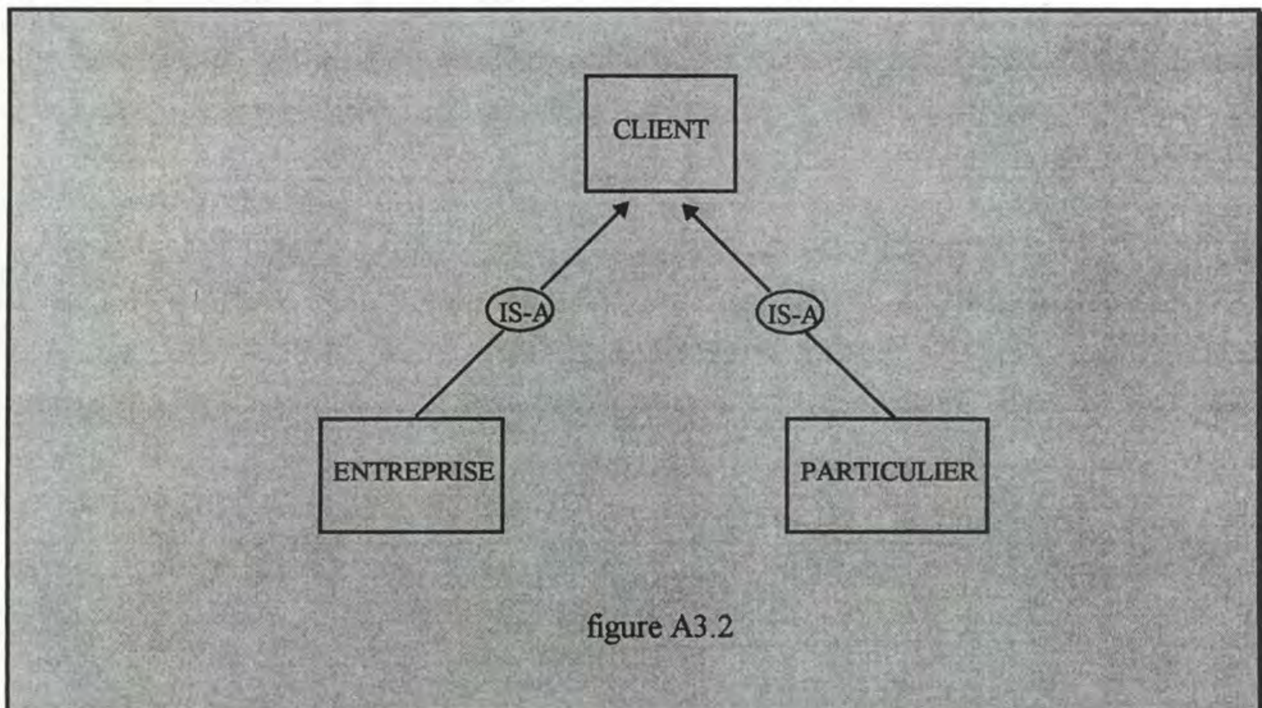


figure A3.2

Voyons maintenant le concept d'héritage. Il s'agit d'un ensemble de règles d'inférence appliquées à la structure d'un schéma dans lequel des relations IS-A sont définies. Il y a deux types d'héritage : l'héritage descendant et l'héritage ascendant.

- Héritage descendant : les attributs et les rôles d'un sous-type sont ses propres attributs et rôles et ceux de son (ses) sur-type(s). Dans le cas de plusieurs sur-types, on parle d'héritage multiple.
- Héritage ascendant : les attributs et rôles d'un sur-type sont ses propres attributs et rôles, ainsi que ceux de ses sous-types. Les attributs et rôles hérités sont facultatifs.

# 3.3 ASSOCIATIONS ET TYPES D'ASSOCIATIONS

Certaines entités sont liées entre elles par des **associations**. Dans une association, chaque entité joue un **rôle** spécifique. Par exemple, si c est une entité de type `CLIENT` et p est une entité de type `PRODUIT`, l'association (c,p) représente le fait que le client c a acheté le produit p. Dans cette association, le rôle de c est `ACHETE`, tandis que le rôle de p est `VENDU`. Des associations similaires sont classées dans un **type d'associations (T.A.)**. Dans notre exemple, on aura donc un T.A. `ACHAT` entre `CLIENT` et `PRODUIT`. La figure A3.3 donne la représentation graphique de cet exemple.

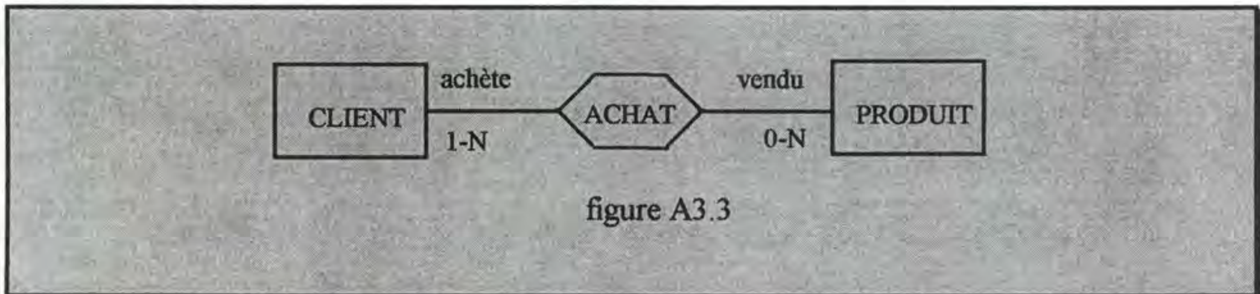



figure A3.3

Le **degré** d'un T.A. est le nombre de T.E. qui y participent. Un T.A. de degré 2 est appelé T.A. binaire.

Quand un T.A. relie un T.E. à lui-même, on parle de T.A. récursif, ou cyclique.

Les **connectivités** d'un rôle permettent de définir le nombre d'associations dans lesquelles chaque entité peut jouer le rôle. Il y a une connectivité minimale et une connectivité maximale sur chaque rôle. La connectivité minimale permet de définir le nombre minimum d'associations dans lesquelles chaque entité doit jouer un rôle. Les valeurs les plus fréquemment utilisées sont 0 et 1. La connectivité maximale permet de définir le nombre maximum d'association dans lesquelles chaque entités peut jouer un rôle. Les valeurs les plus fréquentes sont 1 et N (N représentant un nombre quelconque de fois). Sur le schéma de la figure A3.3, les connectivités des deux rôles sont indiquées en-dessous de chacun de ceux-ci.

Pour les T.A. binaires, on utilise un vocabulaire simplifié : (i représente soit 0 soit 1)

- T.A. one-to-many : l'un des rôles est de connectivité i-1 ;
- T.A. one-to-one : les deux rôles sont de connectivité i-1 ;
- T.A. many-to-many : aucun rôle n'est de connectivité i-1 ;

# 3.4 VALEURS D'ATTRIBUTS, ATTRIBUTS ET DOMAINES

Les caractéristiques propres d'un objet du monde réel (représenté par une entité) sont représentées par des valeurs associées à cette entité. Par exemple, un client dont le nom est Dupont sera représenté par l'entité e du type CLIENT à laquelle est associé la valeur 'DUPONT'. On dira que le type d'entité CLIENT a un **attribut** appelé NOM. 'DUPONT' est la valeur de l'attribut NOM de l'entité e. Chaque attribut tire sa valeur d'un ensemble de valeurs spécifique appelé **domaine**. Le domaine de l'attribut NOM de CLIENT est par exemple l'ensemble des chaînes de caractères de longueur maximale 30. En plus des domaines de base (chaînes de caractères, entiers, réels, ...), on peut définir des domaines en énumérant toutes les valeurs. Par exemple, le domaine de l'attribut STATUT-MARITAL est (marié , célibataire , divorcé , veuf).

Les attributs d'un type d'entités peuvent être obligatoires ou facultatifs, mono-valués ou multi-valués, élémentaires ou décomposables. Nous allons utiliser l'exemple de la figure A3.4 pour expliquer ces concepts.

- un attribut est **obligatoire** s'il a une valeur pour chaque occurrence de son type d'entités. Autrement, il est **facultatif**. Par exemple, l'attribut NOM de PERSONNE est obligatoire, tandis que l'attribut NOM-DE-J-F (représentant le nom de jeune fille) est facultatif.
- un attribut est **monovalué** si au plus une valeur de cet attribut peut être associée a une entité. Si plusieurs valeurs d'un attribut peuvent être associées à une entité, il est **multivalué**. Par exemple, NOM de PERSONNE est monovalué, tandis que PRENOMS est multivalué.

  Remarque

  Il est possible de spécifier le caractère obligatoire/facultatif ou monovalué/multivalué d'un attribut de la même façon que pour les connectivités. Par exemple, un attribut auquel on associe la cardinalité 0-1 est facultatif et monovalué.

- un attribut est **décomposable** si ses valeurs peuvent être décomposées en éléments significatifs. Sinon, il est **élémentaire**. Par exemple, l'attribut ADRESSE est décomposé en RUE, NUMERO, VILLE.

On représente les attributs en inscrivant leur nom dans la boîte représentant le type d'entités auquel ils appartiennent.

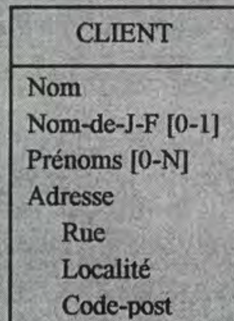



figure A3.4

On peut de la même façon définir les attributs d'un type d'associations.

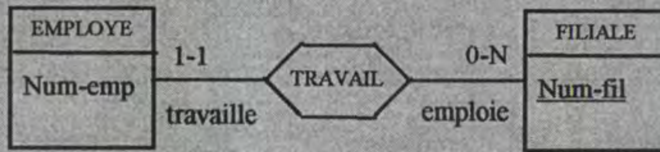
# 3.5 IDENTIFIANTS DES TYPES D'ENTITES

Dans les cas les plus simples, l'**identifiant** d'un type d'entité `TE` est un attribut `A` tel qu'à tout moment, il n'existe pas deux entités du type `TE` pour lesquelles `A` a la même valeur. Par exemple, l'identifiant du type d'entités `CLIENT` pourrait être `NUM-CLI`, représentant le numéro du client. Un type d'entités peut avoir plusieurs identifiants, et ne doit pas forcément avoir d'identifiant.

L'identifiant d'un type d'entités peut être composé de plusieurs attributs. Dans ce cas, à tout moment, il n'y a pas deux entités ayant les mêmes valeurs pour ces attributs.

L'identifiant d'un type d'entité peut être composé d'attributs et d'entités liées. On parle alors d'**identifiant hybride**. Voyons-le sur un exemple. Considérons le schéma de la figure A3.5.

Les employés d'une filiale ont des numéros distincts. Cependant, si l'on considère une firme dans son ensemble, les numéros des employés ne sont plus distincts. L'attribut `Num-emp` ne suffit donc pas pour identifier `EMPLOYE`. L'identifiant de employé est alors composé de `Num-emp` et de `FILIALE`. Pour éviter toute ambiguïté, le composant type d'entités de l'identifiant ne sera pas `FILIALE`, mais plutôt son rôle dans `TRAVAILLE`. L'identifiant de `EMPLOYE` sera donc `(Num-emp , emploie)`.




Id (EMPLOYE) = (Num-emp , emploie)

figure A3.5

Un identifiant peut aussi être uniquement constitué de rôles .

Lorsqu'un identifiant est composé d'un ou de plusieurs attributs, on représente le fait que cet (ces) attributs est (sont) identifiant(s) en le (les) soulignant. Sur le schéma de la figure A3.5, on constate que `Num-fil` est identifiant de `FILIALE`. Lorsqu'un attribut est composé en tout ou en partie de rôles, on l'indique explicitement. On a donc la mention `Id (EMPLOYE) = (Num-emp , emploie)` sur le schéma de la figure A3.5.

# 3.6 IDENTIFIANTS DES TYPES D'ASSOCIATIONS

On peut aussi définir un identifiant sur un type d'associations. Un tel identifiant est composé de rôles et/ou d'attributs. Dans la plupart des cas, l'identifiant d'un type d'associations est l'ensemble de ses rôles.

Il y a deux situations dans lesquelles un identifiant ne sera pas déclaré explicitement :

- si un rôle a la connectivité 0-1 ou 1-1, il est identifiant implicite du type d'associations ;
- si l'identifiant est composé de tous les rôles, il est considéré comme un identifiant par défaut.

# 3.7 CONTRAINTES D'INTEGRITE SUPPLEMENTAIRES

Une contrainte d'intégrité est une propriété formelle que les instances des objets d'un schéma E/A doivent respecter à tout moment. Ces propriétés permettent de décrire le plus fidèlement possible le réel perçu. Nous venons de définir les principales contraintes d'intégrité que l'on peut poser sur un schéma E/A, à savoir les contraintes d'identifiant et de connectivités. On peut potentiellement définir une infinité de contraintes d'intégrité supplémentaires sur un schéma E/A. Nous allons mentionner celles qui sont les plus fréquemment utilisées.

## 3.7.1 Contrainte d'inclusion entre rôles

Considérons le schéma de la figure A3.6. La contrainte d'inclusion entre `ro-1` et `ro-2` peut s'exprimer comme suit : si une entité `a` de type `A` joue le rôle `ro-1`, alors elle doit aussi jouer le rôle `ro-2`.

Si la connectivité minimale de ro-2 était 1, alors il y aurait eu redondance entre la contrainte d'inclusion et la contrainte de connectivité.

## 3.7.2 Contrainte d'égalité entre rôles

Une contrainte d'égalité entre deux rôles correspond à deux contraintes d'inclusion réciproques.

La représentation graphique d'une contrainte d'égalité entre rôles est réalisée sans flèche, et le symbole utilisé est "=".

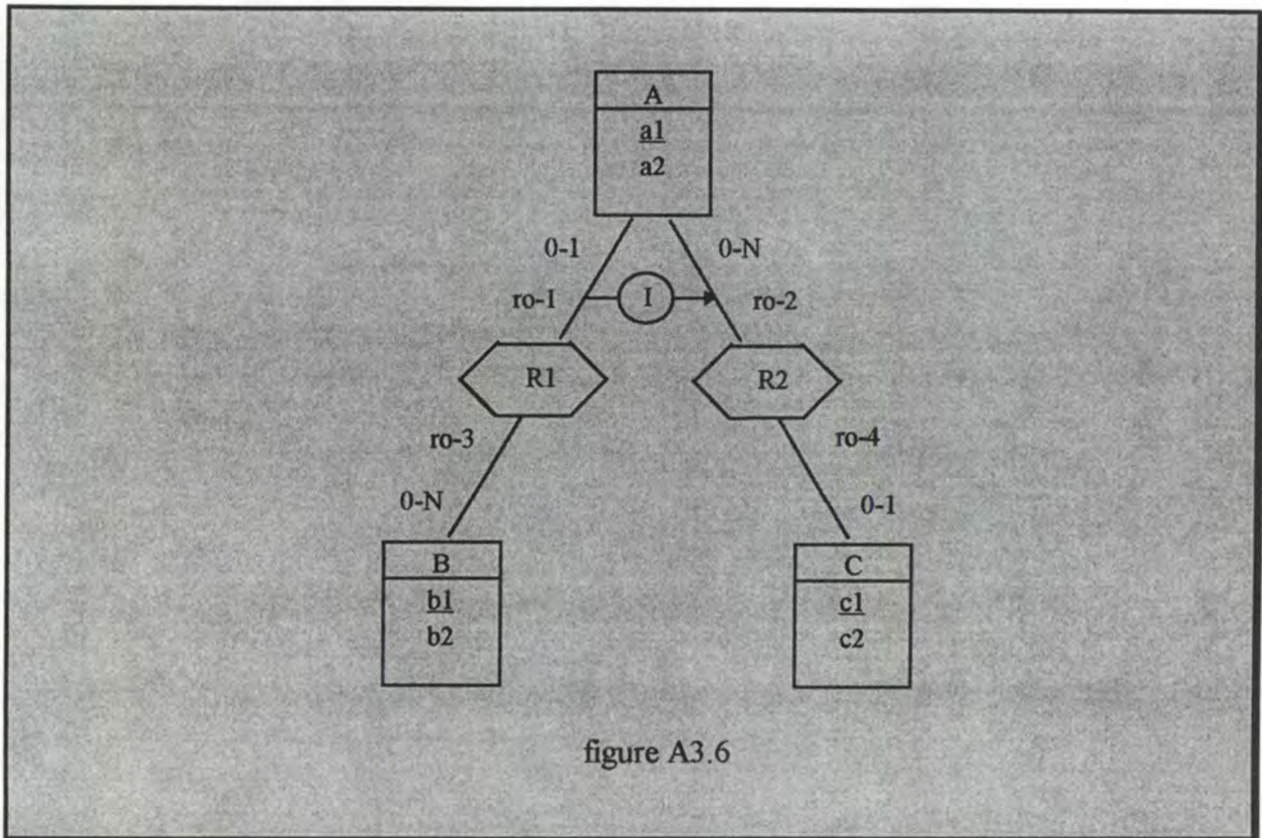


figure A3.6

## 3.7.3 Contrainte d'exclusion entre rôles.

Cette contrainte spécifie que deux rôles ne peuvent pas prendre de valeur simultanément pour une même entité.

En ce qui concerne les rôles, la représentation graphique diffère de celle de la contrainte d'inclusion par le fait qu'il n'y a plus de flèches, et l'exclusion est représentée par le symbole "#".

## 3.7.4 Contrainte d'existence entre rôles

Il peut arriver que l'on désire imposer une contrainte exprimant le fait que, parmi tous ses rôles (ou certains de ses rôles), une entité doit en jouer au moins un.

Prenons comme exemple le schéma de la figure A3.7. Une firme qui loue des voitures et des caravanes. Un client de cette firme peut louer une voiture ou une caravane ou les deux, mais il est client de la firme s'il loue quelque-chose. Il faut donc que le type d'entités `CLIENT` joue au moins un des deux rôles.

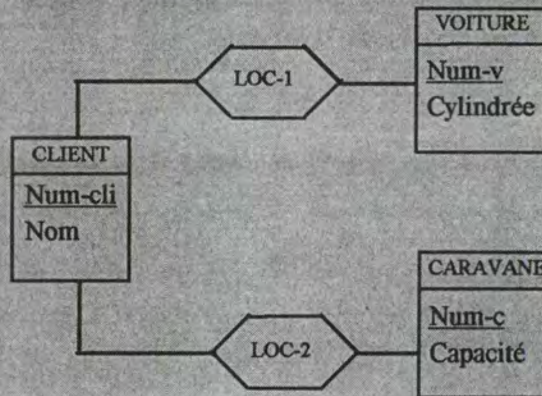



figure A3.7

## 3.7.5 Les contraintes sur les types d'associations

On peut aussi définir des contraintes d'inclusion, d'égalité et d'exclusion sur des T.A. Elles concernent alors tous les rôles du T.A.

La figure A3.8 donne un exemple (tiré de [BOD89]) de contrainte d'exclusion entre T.A.

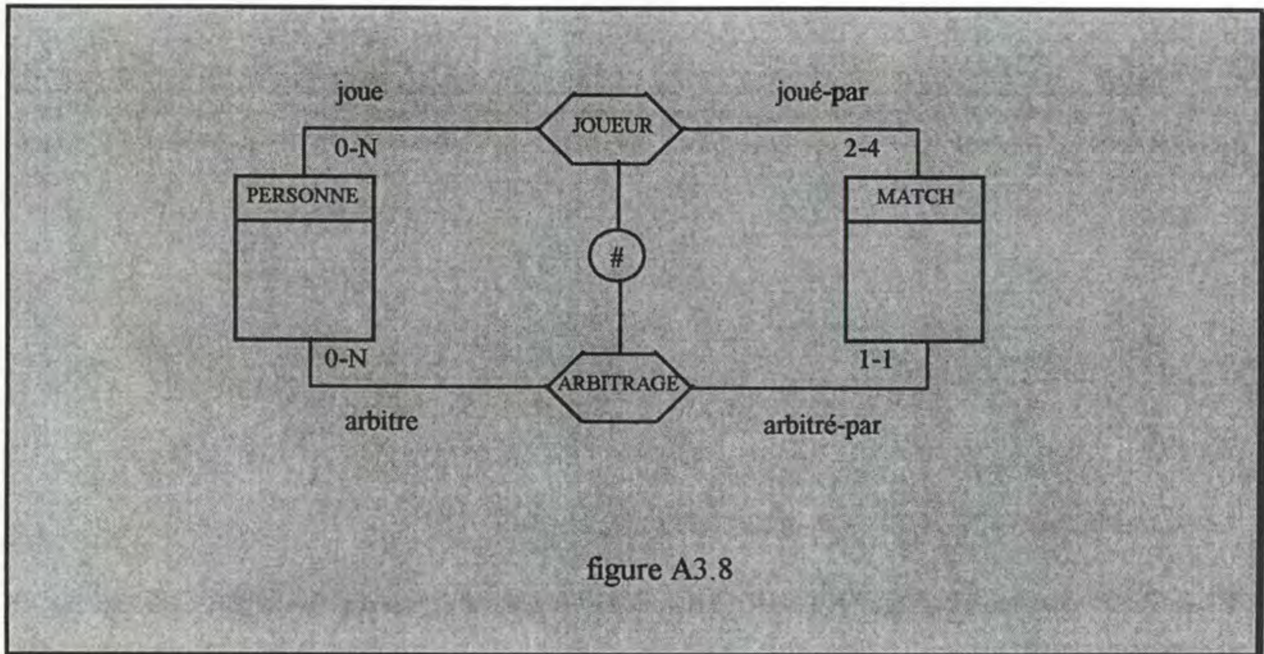



figure A3.8

Cette contrainte d'exclusion exprime qu'une personne ne peut-être à la fois joueur et arbitre d'un match de tennis.

## 3.7.6 Dépendances fonctionnelles

Nous renvoyons le lecteur à l'annexe consacrée au modèle relationnel (cf. annexe 1) pour une défintion des dépendances fonctionnelles. Il s'agit en effet de la même notion dans les deux modèles (mentionnons seulement le fait que dans le modèle E/A, le déterminé et le déterminant peuvent être composés d'attributs et/ou de rôles).

# ANNEXE 4

# REDONDANCE ENTRE TYPES D'ASSOCIATIONS MANY-TO-MANY

## 4.1 DEFINITION

Considérons le schéma conceptuel de la figure A4.1.

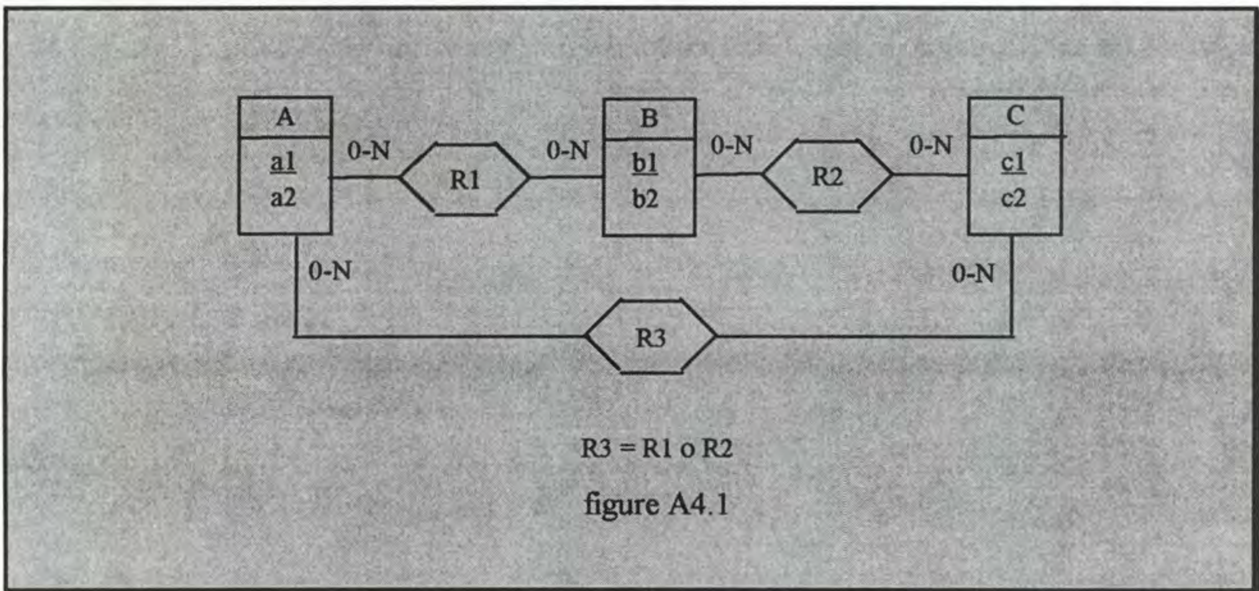
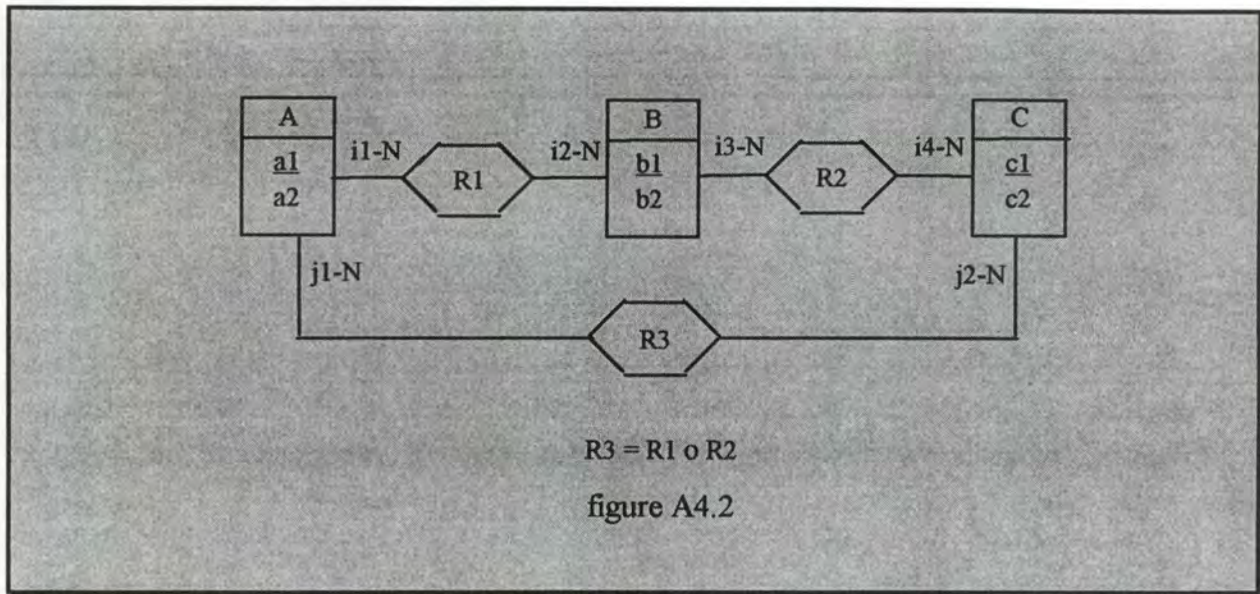



figure A4.1

Le type d'associations `R3` est redondant avec la composition des associations `R1` et `R2`.

## 4.2 RECHERCHE BASEE SUR LES CONNECTIVITES

On peut trouver un indice de redondance entre T.A. many-to-many en se basant sur les connectivités. Considérons le schéma de la figure A4.2.




R3 = R1 o R2

figure A4.2

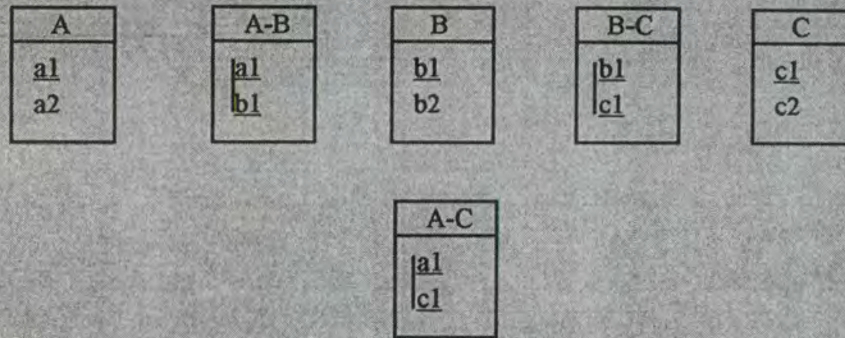
On aura peut-être une redondance entre `R3` et `R1 o R2` si

j1 = min (i1 , i3)

j2 = min (i2 , i4)

Cela peut facilement être généralisé à la redondance entre un T.A. many-to-many et la composition de n T.A many-to-many.

La traduction du schéma de la figure A4.1 peut se faire en transformant les T.A. many-to-many en types d'entités. On n'a plus alors que les T.A. one-to-many. La traduction en structures conformes au modèle relationnel est alors aisée. Le schéma de la figure A4.3 donne une telle traduction. On constate sur ce schéma que la redondance entre T.A. many-to-many a été transformée en redondance entre tables. En effet, la table `AC` est redondante avec la composition des tables `AB` et `BC`. Ce genre de table redondante est utilisée pour établir un lien direct entre deux tables éloignées reliées via d'autres tables par un lien many-to-many.




$$AB.a1 \subseteq A.a1$$
$$AB.b1 \subseteq B.b1$$
$$BC.b1 \subseteq B.b1$$
$$BC.c1 \subseteq C.c1$$
$$AC.a1 \subseteq A.a1$$
$$AC.c1 \subseteq C.c1$$
$$(\forall\ ab \in AB,\ bc \in BC \mid ab[b1] = bc[b1]) \ :\ \exists\ ac \in AC \mid ac[a1] = ab[a1] \text{ et } ac[c1] = c[c1]$$

figure A4.3

# 4.3 RECHERCHE DANS LES COMPORTEMENTS IMPOSES AUX DONNEES

Nous allons nous contenter ici de définir les UC.

Nous allons voir quelles vont être les conditions et conséquences de l'ajout, de la suppression et de la modification d'éléments des tables concernées par la redondance. Dans le schéma ci-dessus, les tables `AB` et `BC` sont appelées tables source et la table `AC` est appelée table dérivée. Pour les UC sur une table source, nous ne verrons que ceux sur `AB`, les UC sur `BC` étant très proches.

*A) Ajout d'une ligne dans une des tables source*

```
INSERT ab IN AB |(∃ bc ∈ BC | bc[b1] = ab[b1])
IMPLIES INSERT ac IN AC | ac[a1] = ab[a1] AND ac[c1] = bc[c1])
```

On insère dans AC le (les) lien(s) créé(s) entre A et C par l'ajout dans AB si cet ajout a créé un ou plusieurs liens entre A et C. On insère dans AC autant de lignes qu'il y a de couples ac[a1] , ac[c1] qui vérifient les conditions.

*B) Suppression d'une ligne dans une des tables source.*

```
DELETE ab FROM AB | (∃ bc ∈ BC | bc[b1] = ab[b1])
IMPLIES DELETE ac FROM AC | ac[a1] = ab[a1] AND ac[c1] = bc[c1])
```

On supprime dans AC les éléments correspondants aux liens entre A et C détruits suite à la suppression de ab.

*C) Modification d'une ligne dans une des tables source*

```
REPLACE ab BY ab' IN AB | (∃ bc' ∈ BC | bc'[b1] = ab[b1]) OR
                        . (∃ bc'' ∈ BC | bc''[b1] = ab'[b1])

IMPLIES

DELETE ab FROM AB
IMPLIES DELETE ac FROM AC | ac[a1] = ab[a1] AND ac[c1] = bc'[c1])

INSERT ab IN AB
IMPLIES INSERT ac IN AC | ac[a1] = ab'[a1] AND ac[c1] = bc''[c1])
```

On supprime dans AC les éléments correspondants aux anciens liens entre A et C modifiés suite au changement dans AB, et on y ajoute les nouveaux liens créés suite à ce même changement.

*D)Ajout d'une ligne dans la table redondante*

```
INSERT ac IN AC
PRECOND ∃ ab ∈ AB, bc ∈ BC | ac[a1] = ab[a1] AND ab[b1] = bc[b1] AND
        bc[c1] = ac[c1]
```

Lorsqu'on ajoute une ligne dans AC, on vérifie qu'elle correspond à un lien entre A et C via AB et BC.

*E) Suppression d'une ligne dans la table redondante*

```
DELETE ac FROM AC
PRECOND ¬ ∃ ab ∈ AB, bc ∈ BC | ac[a1] = ab[a1] AND ab[b1] = bc[b1] AND
        bc[c1] = ac[c1]
```

Lorsqu'on supprime une ligne de AC, on vérifie qu'elle ne correspondait plus à un lien entre A et C via AB et BC.

*F) Modification d'une ligne dans la table redondante*

```
REPLACE ac BY ac' IN AC

PRECOND

¬ ∃ ab ∈ AB, bc ∈ BC | ac[a1] = ab[a1] AND ab[b1] = bc[b1] AND bc[c1] =
                                                              ac[c1]

AND

∃ ab ∈ AB, bc ∈ BC | ac'[a1] = ab[a1] AND ab[b1] = bc[b1] AND bc[c1] =
                                                              ac'[c1]
```

Cette UC est la composition des deux précédentes. En effet, une modification de liens correspond à une suppression de liens suivie d'une insertion de liens.

*Généralisation*

Si le T.A. AC est redondant avec la composition de n T.A., il faudra considérer des UC pour toutes les tables liant A et C et contenant des clés étrangères.