

BASES DE DONNÉES ET MODÈLES DE CALCUL

**Outils et méthodes
pour l'utilisateur**

Cours et exercices corrigés

Jean-Luc Hainaut

Professeur à l'Institut d'Informatique
des Facultés Universitaires Notre-Dame de la Paix, Namur

4^e édition

DUNOD

Illustration de couverture : *Contexture, digitalvision®*

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.

Le Code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements

d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour

les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée. Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du

Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).



© Dunod, Paris, 2000, 2002, 2005

© InterEditions, Paris, 1994

ISBN 2 10 049146 6

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2^e et 3^e a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

à

Yves, Véronique et Jean-Pierre,
Alain, Benoît et Carine,
Bertrand, Catherine, Muriel et Michel,
Olivier, Mario et Bernard,
Didier, Jean, Vincent et Jean-Marc,
Alain, Pierre, Thierry et Anne-France,
Arnaud, Stéphane, Olivier, Philippe et Majid,
Denis, Virginie et Thomas,
Christine
Aurore et Fabrice,
Jean-Roch,
Ravi et Julien,
Eric,
Anthony,
Yannis et Frédéric

Table des matières

AVANT-PROPOS	15
CHAPITRE 1 • MOTIVATION ET INTRODUCTION	17
1.1 L'utilisateur-développeur, heurs...	17
1.2 ... et malheurs	18
1.3 Objectif de l'ouvrage	21
 PARTIE 1	
LES BASES DE DONNÉES	
 CHAPITRE 2 • INTRODUCTION	25
2.1 L'utilisateur et les données	25
2.2 Bases de données et SGBD relationnels	26
2.3 Construction d'une base de données	28
2.4 Description de la première partie	29
2.5 Pour en savoir plus	30
 CHAPITRE 3 • CONCEPTS DES BASES DE DONNÉES	31
3.1 Table, ligne et colonne	31
3.2 Rôles d'une colonne	33
3.2.1 Les identifiants	33
3.2.2 Les clés étrangères	34
3.2.3 Les informations complémentaires	34

3.2.4	Les identifiants et clés étrangères multicomposants	34
3.2.5	Les identifiants primaires	34
3.2.6	Les contraintes référentielles	35
3.2.7	Les colonnes facultatives	35
3.3	Structure et contenu d'une base de données	36
3.4	Représentation graphique d'un schéma	37
3.5	Un exemple de base de données	38
3.6	Autres notations graphiques	40
3.7	Note sur les contraintes référentielles	41
3.8	Modification et contraintes d'intégrité	43
3.8.1	Les contraintes d'unicité (identifiants)	43
3.8.2	Les contraintes référentielles (clés étrangères)	43
3.8.3	Les colonnes obligatoires	45
3.9	La normalisation	45
3.9.1	Le phénomène de redondance interne	46
3.9.2	Normalisation par décomposition	46
3.9.3	Analyse du phénomène	47
3.9.4	Remarques	49
3.10	Les structures physiques	50
3.11	Les systèmes de gestion de données	51
3.12	SQL et les bases de données	53
3.13	Exercices	54
CHAPITRE 4 • LE LANGAGE SQL DDL		55
4.1	Introduction	55
4.2	Le langage SQL DDL	56
4.3	Création d'un schéma	56
4.4	Création d'une table	57
4.5	Suppression d'une table	61
4.6	Ajout, retrait et modification d'une colonne	61
4.7	Ajout et retrait d'une contrainte	62
4.8	Les structures physiques	63
CHAPITRE 5 • LE LANGAGE SQL DML		65
5.1	Introduction	65
5.2	Consultation et extraction de données dans une table	66
5.2.1	Principes	66
5.2.2	Extraction simple	66

5.2.3	Extraction de lignes sélectionnées	67
5.2.4	Lignes dupliquées dans le résultat	68
5.2.5	Des conditions de sélection plus complexes	70
5.2.6	Un peu de logique	72
5.2.7	Données extraites et données dérivées	76
5.2.8	Les fonctions SQL	77
5.2.9	Les fonctions agrégatives (ou statistiques)	80
5.3	Sélection utilisant plusieurs tables : les sous-requêtes	82
5.3.1	Les sous-requêtes	82
5.3.2	Sous-requête et clé étrangère multi-composant	83
5.3.3	Attention aux conditions d'association négatives	84
5.3.4	Références multiples à une même table	86
5.3.5	Les quantificateurs ensemblistes	88
5.4	Extraction de données de plusieurs tables (jointure)	91
5.4.1	La jointure de plusieurs tables	91
5.4.2	Conditions de jointure et conditions de sélection	92
5.4.3	Jointures sans conditions : produit relationnel	93
5.4.4	La jointure et les lignes célibataires - Les opérateurs ensemblistes	93
5.4.5	Les requêtes sur des structures de données cycliques	96
5.4.6	Sous-requête ou jointure ?	100
5.4.7	Valeurs dérivées dans une jointure	103
5.4.8	Les jointures généralisées	103
5.4.9	Interprétation du résultat d'une jointure	104
5.5	Extraction de données groupées	107
5.5.1	Notion de groupe de lignes	107
5.5.2	Sélection de groupes et sélection de lignes	108
5.5.3	Groupes et jointures	109
5.5.4	Composition du critère de groupement	110
5.5.5	Attention aux groupements multi-niveaux	111
5.5.6	Peut-on éviter l'utilisation de données groupées ?	112
5.6	Ordre des lignes d'un résultat	113
5.7	Interprétation d'une requête	114
5.8	Modification des données	115
5.8.1	Ajout de lignes	115
5.8.2	Suppression de lignes	116
5.8.3	Modification de lignes	117
5.8.4	Mise à jour et contraintes référentielles	117
5.8.5	Modification des structures de données	120
5.9	Exercices	121
5.9.1	Énoncés de type 1	121
5.9.2	Énoncés de type 2	121
5.9.3	Énoncés de type 3	122
5.9.4	Énoncés de type 4	123

5.9.5	Énoncés de type 5	126
5.9.6	Énoncés de type 6	128
5.9.7	Énoncé de type 7	129
CHAPITRE 6 • SQL AVANCÉ		131
6.1	Le contrôle d'accès	131
6.2	Les vues SQL	133
6.2.1	Principe et objectif des vues	134
6.2.2	Définition et utilisation d'une vue	134
6.2.3	Les vues comme interface pour des besoins particuliers	135
6.2.4	Les vues comme mécanisme de contrôle d'accès	135
6.2.5	Les vues comme mécanisme d'évolution de la base de données	136
6.2.6	Les vues comme aide à l'expression de requêtes complexes	136
6.2.7	Mise à jour des données via une vue	136
6.3	Extension de la structure des requêtes SFW	137
6.3.1	Extension de la clause select	137
6.3.2	Extension de la clause from	138
6.3.3	Les requêtes récursives	143
6.4	Les prédicats (check)	144
6.5	Les procédures SQL (stored procedures)	145
6.6	Les déclencheurs (triggers)	146
6.7	Le catalogue	147
6.8	Les extensions proposées par SQL3	151
6.9	Les interfaces entre BD et programmes d'application	152
6.10	SQL et l'information incomplète	156
6.10.1	Introduction	156
6.10.2	La valeur null de SQL	156
6.10.3	La logique ternaire de SQL	156
6.10.4	La propagation de null en SQL	157
6.10.5	La propagation de unknown en SQL	158
6.10.6	Les problèmes de l'information incomplète en SQL	159
6.10.7	Deux recommandations	164
6.11	Exercices	165
6.11.1	Contrôle d'accès	165
6.11.2	Le catalogue	165
CHAPITRE 7 • APPLICATIONS AVANCÉES EN SQL		167
7.1	Les structures d'ordre	167
7.2	Les bases de données actives	170
7.2.1	Les contraintes d'intégrité statiques	170
7.2.2	Les contraintes d'intégrité dynamiques	170

7.2.3	Le contrôle de la redondance	171
7.2.4	Les alerteurs	172
7.2.5	Personnalisation des comportements standard	173
7.2.6	Intégration d'une règle de gestion dans la base de données	173
7.3	Les données temporelles	174
7.3.1	Représentation des données temporelles	174
7.3.2	Interrogation de données temporelles	175
7.3.3	La projection temporelle	176
7.3.4	La jointure temporelle	178
7.3.5	Gestion des données historiques	180
7.4	La génération de code	183
7.4.1	Migration de données	183
7.4.2	Génération de migrateurs de données	184
7.4.3	Génération de définitions de bases de données	186
7.4.4	Génération de pages HTML	187
7.4.5	Génération de documents XML	189
7.4.6	Génération de générateurs de pages HTML ou de documents XML	189
7.5	Exercices	190
7.5.1	Les structures d'ordre	190
7.5.2	Les bases de données actives	190
7.5.3	Les données temporelles	192
7.5.4	La génération de code	195
CHAPITRE 8 • CONSTRUCTION D'UNE BASE DE DONNÉES		199
CHAPITRE 9 • LE MODÈLE ENTITÉ-ASSOCIATION		203
9.1	Types d'entités	203
9.2	Attributs	204
9.3	Types d'associations	205
9.3.1	Propriétés d'un type d'associations	206
9.4	Les identifiants	211
9.4.1	Les identifiants hybrides	212
9.4.2	Composition des identifiants	214
9.4.3	Identifiants minimaux et identifiants implicites	214
9.4.4	Importance du concept d'identifiant	215
9.5	Autres contraintes d'intégrité	216
9.5.1	Les contraintes d'intégrité statiques	217
9.5.2	Les contraintes d'intégrité dynamiques	217
9.6	Contenu informationnel d'un schéma	217
9.7	Exemples	218
9.7.1	Une structure administrative	219

9.7.2	Gestion d'une bibliothèque	220
9.7.3	Voyages en train	221
9.8	Quelques règles de présentation	222
9.9	Extensions du modèle entité-association	222
9.10	... et UML ?	226
9.10.1	Le modèle de classes d'UML	226
9.10.2	Un exemple de schéma de classes en UML	229
9.10.3	Le modèle de classes d'UML revisité	230
9.11	Exercices	232
CHAPITRE 10 • ÉLABORATION D'UN SCHÉMA CONCEPTUEL		233
10.1	Introduction	233
10.2	Décomposition de l'énoncé	235
10.3	Pertinence d'une proposition	241
10.4	Représentation d'une proposition	241
10.5	Non-redondance des propositions	250
10.6	Non-contradiction des propositions	253
10.7	Les contraintes d'intégrité	254
10.8	Documentation du schéma	256
10.9	Complétude du schéma	257
10.10	Normalisation du schéma	258
10.11	Validation du schéma	260
10.12	Exercices	261
CHAPITRE 11 • PRODUCTION DU SCHÉMA DE LA BASE DE DONNÉES		269
11.1	Introduction	269
11.2	Représentation des types d'entités	270
11.3	Représentation des attributs	270
11.4	Représentation des types d'associations	270
11.4.1	Types d'associations un-à-plusieurs	270
11.4.2	Types d'associations un-à-un	273
11.4.3	Types d'associations plusieurs-à-plusieurs	275
11.4.4	Types d'associations cycliques	276
11.5	Représentation des identifiants	276
11.6	Traduction des noms	277
11.7	Synthèse des règles de traduction	277
11.8	Les structures physiques	278

11.9 Traduction des structures en SQL	280
11.10 Compléments	281
11.10.1 Les contraintes d'intégrité additionnelles	281
11.10.2 Au sujet des rôles de cardinalité 1-N	283
11.11 Rétro-ingénierie d'une base de données	284
11.12 Extensions de la méthode	288
11.13 Exercices	289

CHAPITRE 12 • BASES DE DONNÉES : ÉTUDES DE CAS 293

12.1 Introduction	293
12.2 Les animaux du zoo	294
12.2.1 Énoncé	294
12.2.2 Construction du schéma conceptuel	294
12.2.3 Production du schéma de tables	296
12.2.4 Production du code SQL	297
12.3 Voyages aériens	299
12.3.1 Énoncé	299
12.3.2 Construction du schéma conceptuel	300
12.3.3 Production du schéma de tables	303
12.4 Exercice	303

PARTIE 2 LES MODÈLES DE CALCUL

CHAPITRE 13 • INTRODUCTION 307

13.1 Le tableur	307
13.2 Le concept de modèle	308
13.3 Construction d'un modèle de calcul	308
13.4 Description de la deuxième partie	309
13.5 Pour en savoir plus	309

CHAPITRE 14 • CONCEPTS DES MODÈLES DE CALCUL 311

14.1 Modèles et processeurs de modèles	311
14.2 Modèles et tableaux	312
14.3 Représentation d'un modèle dans une feuille de calcul	313
14.4 Le marché des tableurs	315

CHAPITRE 15 • UN TABLEUR TYPE : EXCEL	317
15.1 Présentation d'Excel	317
15.2 La feuille de calcul	318
15.3 Organisation des feuilles de calcul et des modèles	318
15.4 Les composants d'un modèle	319
15.4.1 Désignation de cellules	319
15.4.2 Le contenu des cellules	319
15.4.3 Les formules	320
15.5 Modifications élémentaires d'un modèle	322
15.6 Déplacement et copie de fragments de modèles	323
15.6.1 Adresses relatives et adresses absolues	323
15.7 Les références circulaires	325
15.8 Fonctions de bases de données	326
15.9 Les tables de données	326
15.10 Les scénarios	327
15.11 Macros et fonctions personnalisées	327
15.12 Les solveurs avancés	328
15.12.1 La valeur cible	328
15.12.2 Le solveur	328
CHAPITRE 16 • CONSTRUCTION D'UN MODÈLE DE CALCUL	331
CHAPITRE 17 • EXPRESSION ABSTRAITE D'UN MODÈLE	335
17.1 Introduction	335
17.2 Grandeurs et règles	336
17.3 Notion de modèle	338
17.4 Descriptions externe et interne d'un modèle	340
17.5 Grandeurs à définition multiple	341
17.6 Grandeurs et règles logiques	342
17.7 Graphe de dépendance	342
17.8 Les valeurs d'exception	345
17.9 Grandeurs et modèles dimensionnés	347
17.10 Les fonctions agrégatives	349
17.11 Règles de récurrence et récursivité	351
17.12 Sous-modèles et modularisation	356

CHAPITRE 18 • CONCEPTION D'UN MODÈLE	361
18.1 Démarche de conception d'un modèle	361
18.2 Les principes	362
18.3 La démarche	363
18.3.1 Analyse	363
18.3.2 Normalisation du modèle	368
18.3.3 Validation du modèle	370
18.3.4 Généralisation par dimensionnement	371
18.4 Sous-modèles non directionnels	372
18.5 Cohérence d'un modèle	374
18.5.1 Cohérence structurelle	374
18.5.2 Cohérence des règles de définition multiple	375
18.5.3 Cohérence des règles de récurrence	376
18.5.4 Cohérence des unités	377
18.5.5 Cohérence des dimensions	379
18.5.6 Cohérence des domaines de valeurs du modèle	381
18.6 Exercices	388
18.6.1 Modèles élémentaires	388
18.6.2 Modèles avancés	389
18.6.3 Validation de modèles	394
18.6.4 ... et en guise de dessert	396
CHAPITRE 19 • IMPLANTATION D'UN MODÈLE DANS UNE FEUILLE DE CALCUL	397
19.1 Élaboration d'une maquette	397
19.1.1 Représentation des grandeurs dimensionnées	398
19.1.2 Les grandeurs internes	400
19.1.3 Les sous-modèles	401
19.1.4 Exemple de maquette	402
19.1.5 Ergonomie des modèles	403
19.2 Traduction des règles	404
19.2.1 Principes généraux	404
19.2.2 Grandeurs à définition multiple	405
19.2.3 Règles de récurrence et règles récursives	405
19.2.4 Les contraintes	406
19.3 Séquentialisation d'un modèle	407
19.4 Réalisation d'un programme séquentiel	410
19.5 Exercices	412
CHAPITRE 20 • MODÈLES : ÉTUDES DE CAS	413
20.1 Introduction	413
20.2 Les animaux du zoo	413

20.2.1 Énoncé	414
20.2.2 Construction du modèle abstrait	414
20.2.3 Implantation du modèle dans une feuille de calcul	418
20.3 Voyages aériens	421
20.3.1 Construction du modèle abstrait	421
20.3.2 Implantation du modèle dans une feuille de calcul	424
BIBLIOGRAPHIE	427
INDEX	431

Avant-propos

Un ouvrage qui combine bases de données et feuilles de calcul, ce que certains lecteurs traduiront un peu trop rapidement par *Access + Excel*, pourrait étonner à une époque où les ouvrages techniques sont généralement très ciblés.

Avant d'être technique, cet ouvrage est essentiellement *méthodologique*. Il s'attaque à la question du *savoir-(bien-)faire*, plutôt qu'à celle de la maîtrise technique des arcanes d'un outil, thème qui est largement majoritaire dans la production littéraire informatique. Bien sûr, construire une base de données ou une feuille de calcul correctes suppose des connaissances raisonnables sur les bases de données et les tableurs, mais ces connaissances sont sans objet si nous ne sommes pas capable de poser correctement le problème, ce qu'on appelle *modéliser*. Une fois le problème posé, exprimer sa solution à l'aide des outils informatiques puissants dont nous disposons aujourd'hui devient un problème étonnamment simple.

C'est l'ambition de cet ouvrage que d'amener le lecteur motivé, qu'il soit débutant ou informaticien curieux, à bien comprendre les bases des deux outils essentiels que sont les systèmes de bases de données et les tableurs, et à les utiliser pour résoudre de manière correcte des problèmes non triviaux.

Tout ceci ne répond pas à la question : pourquoi coupler bases de données et feuilles de calcul ? Pour deux raisons. D'une part, il s'agit de deux modes de résolution de problèmes très puissants, mais néanmoins à la portée des utilisateurs, pour peu qu'ils soient raisonnablement motivés. Rappelons que toutes les suites bureautiques, d'Office de Microsoft jusqu'à Star Office, comportent un gestionnaire de données et un tableur. D'autre part, il apparaît que construire de manière disciplinée une solution informatique, qu'il s'agisse d'une base de données ou d'une feuille de calcul, constitue un seul et même exercice intellectuel.

Cet ouvrage est le fruit d'une démarche d'enseignement et de recherche tant en milieu universitaire qu'en entreprise. Sa structure a en grande partie été dictée par les difficultés que des générations d'étudiants et de professionnels ont rencontrées

lors des enseignements et des séminaires que nous avons organisés depuis près de vingt ans.

Dans ce contexte, il est clair qu'une telle matière n'a pu prendre forme sans de nombreuses collaborations, parfois ténues en apparence, mais toujours significatives à terme. Je remercie donc tous ceux qui m'ont aidé, et en particulier :

- Véronique Goemans, Lysiane Gailly-Goffaux et Jean-Pierre Thiry avec qui j'ai animé les sessions consacrées à l'aide à la décision, et les membres du personnel d'UTA. Claire Faure, qui organisait ces formations;
- Anne Borsu-Bilande, Vincent Sapin, Damien Lanotte, Carine Papin, Alain Gofflot, Muriel Chandelon, Olivier Marchand, Jean Henrard, Jean-Marc Hick, Vincent Englebert, Didier Roland, Alain Gofflot, Pierre Delvaux, Anne-France Brogneaux, Philippe Thiran, Virginie Detienne, Aurore François et Jean-Roch Meurisse qui m'ont aidé, et m'aident encore, dans mes cours de l'Institut d'Informatique et à la faculté des Sciences Économiques de Namur;
- mes collègues, anciens et actuels, Karin Becker, François Bodart, Jean-Paul Leclercq, Jean-Marie Lambert, Patrick Heymans pour nos discussions méthodologiques; une pensée toute particulière à Vincent Englebert, pour sa sagacité à déminer SQL;
- les étudiants des facultés de Sciences Économiques, Sociales et Politiques, de Droit et d'Informatique de l'université de Namur, qui, par leurs réactions et leurs exigences, ont peu à peu imposé à cet ouvrage sa forme actuelle,
- mes collègues d'autres écoles et universités, ainsi que leurs étudiants, qui n'ont pas hésité à me faire part de leurs commentaires et suggestions concernant les premières éditions de cet ouvrage; je remercie en particulier le professeur Yves Pigneur de l'Université de Lausanne et ses étudiants,
- tous ceux que ma mauvaise mémoire et ma distraction m'auraient fait oublier.

Cet ouvrage est la quatrième édition de celui qui avait été publié chez InterEditions en 1994 [Hainaut, 1994]. Il s'en distingue sur les points suivants : la première partie consacrée aux bases de données a été substantiellement révisée et augmentée afin que soient prises en compte les suggestions qui m'ont été formulées, l'évolution des outils (SQL2), celle des méthodes (rétro-ingénierie, AGL, UML) et les nouvelles applications (BD actives, BD temporelles, HTML, XML). La deuxième partie (modèles de calcul) a été rafraîchie et actualisée. La troisième partie abordant le couplage entre bases de données et modèles de calcul a été supprimée.

Un site Web accompagne cet ouvrage. Il contient des matériaux complémentaires concernant les exercices, dont certains accompagnés d'une solution, des corrections éventuelles, des outils (dont l'AGL DB-MAIN), ainsi que la troisième partie de [Hainaut, 1994] absente de cette édition.

Adresse du site : www.info.fundp.ac.be/libd (volet Documents / Ouvrages)

Contacts : jlh@info.fundp.ac.be

Chapitre 1

Motivation et introduction

Ce premier chapitre présente les problèmes que posent les outils populaires de développement destinés à l'utilisateur final, en particulier les gestionnaires de bases de données et les tableurs. Il évoque les lacunes sur le plan méthodologique qui conduisent à des résultats peu fiables et difficiles à modifier. Les objectifs et la structure de cet ouvrage se déduisent de cette analyse.

1.1 L'UTILISATEUR-DÉVELOPPEUR, HEURS...

Dès leur apparition sur ordinateurs personnels, à la fin des années 70, les systèmes de gestion de bases de données et les tableurs ont été perçus comme des outils de développement accessibles à l'utilisateur final, celui-là même qui devait jusqu'alors s'en remettre au service informatique pour obtenir le moindre traitement d'information. L'utilisateur pouvait désormais se libérer et devenir son propre développeur. Il devait se préparer à ce nouveau métier. Il s'y prépara. On enseigna donc dBase II puis dBase III, VisiCalc puis Lotus 1-2-3.

L'avenir de l'informatique s'annonçait exaltant. D'un côté une informatique traditionnelle, lourde, aux mains des professionnels, basée sur les *mainframes*¹, en marge de l'évolution naturelle, manifestement destinée à disparaître à terme, entraînant avec elle ses informaticiens. De l'autre, le foisonnement croissant et

1. On désignait (et on désigne encore) sous ce terme un gros ordinateur central traditionnellement destiné à l'informatique lourde, scientifique ou plus généralement de gestion et auquel sont connectés des terminaux.

multicolore d'ordinateurs personnels conviviaux, bon marché, puissants, transformant le moindre utilisateur, le moindre amateur en développeur passionné et efficace. Celui-ci ne vient-il pas de construire, partant de zéro, sa première base de données en vingt minutes, son premier modèle de calcul (une vraie comptabilité en miniature) en une heure, son premier programme d'interrogation dBase en deux heures ? Et ce pour un coût dérisoire. L'informaticien de métier n'avait qu'à bien se tenir, lui qui réclamait plusieurs semaines pour livrer le même résultat, d'ailleurs au prix fort.

1.2 ... ET MALHEURS

L'informaticien est toujours là. Son métier a changé, sa position dans l'entreprise a évolué, de même que ses rapports avec les utilisateurs. Les bouleversements technologiques que nous avons évoqués y sont pour beaucoup. Bien sûr, il développe aujourd'hui autant en Access, Visual Basic et en Excel qu'en COBOL (ou C ou Java) et SQL, mais son rôle de développeur ne lui a toujours pas été retiré. L'utilisateur n'en a plus voulu, les raisins étaient sans doute encore un peu verts.

Cette histoire en rappelle d'autres :

- COBOL devait permettre aux comptables eux-mêmes de décrire les informations et les traitements relatifs à leur métier, pour pallier le manque de programmeurs (1958);
- BASIC, par sa simplicité, était destiné à l'utilisateur final (*Beginners' All-purpose Symbolic Instruction Code*), enfin libéré du joug du programmeur (1970);
- les systèmes experts devaient être construits par les experts eux-mêmes, la chose était évidente; qu'y a-t-il de plus simple en effet qu'exprimer ses connaissances et ses raisonnements sous la forme de règles de production (années 80) ?
- le modèle relationnel devait révolutionner le domaine des bases de données : la structure tabulaire des données et un langage déclaratif semi-naturel devaient mettre cette technologie à la portée des utilisateurs (années 80); la construction de la base de données elle-même devait se simplifier fortement : l'utilisateur décrit ses tables et l'ordinateur se charge de l'implantation efficace et de l'optimisation des requêtes.

Loin de nous l'idée que ces outils originaux que sont les tableurs et les systèmes de bases de données ne sont pas *aussi* à la portée des utilisateurs. Nous voulons simplement rappeler que la construction d'une application informatique, quelle qu'en soit la complexité et quel qu'en soit l'outil de réalisation, est une affaire sérieuse, qu'elle exige rigueur, soin et méthode, afin que cette application constitue pour ses utilisateurs un outil fiable et efficace. L'utilisateur peut devenir, sans douleur mais pas sans efforts, un développeur compétent, pourvu qu'il se soumette à certaines exigences du métier, qui vont le plus souvent bien au-delà de la simple maîtrise de l'outil informatique, d'ailleurs lui-même de moins en moins facile à maîtriser.

Les professionnels du développement informatique sont confortablement outillés pour répondre à ces exigences. Ils ont reçu un enseignement technique et théorique solide, le plus souvent accompagné d'une expérience pratique de départ, expérience qu'ils enrichissent rapidement sur le terrain. Ils disposent de méthodes de modélisation et de construction d'applications qui les guident dans leurs activités de production. Ils utilisent des environnements de développement puissants, tels les dictionnaires de données, les générateurs d'application, les environnements de développement rapide (RAD), les bibliothèques de composants ou les ateliers de génie logiciel.

Si l'état de l'art en matière d'outils de développement pour l'utilisateur, en l'occurrence les tableurs et les systèmes de bases de données, évolue très rapidement, on ne peut en dire autant des aspects méthodologiques. L'utilisateur-développeur, qui n'est pas un professionnel de l'informatique, et qui le plus souvent ne peut justifier d'une formation préalable solide, est confronté à des logiciels de plus en plus puissants, de plus en plus difficiles à maîtriser, et surtout aborde des problèmes de plus en plus complexes. Au contraire du développeur professionnel, il ne dispose ni de modèles, ni de méthodes, ni *a fortiori* d'outils d'aide au développement. La situation est donc proche de celle du programmeur du début des années 60, peu à peu conscient que la maîtrise de FORTRAN ou de COBOL ne suffit plus à la construction systématique de programmes fiables et maintenables.

Les ouvrages largement disponibles chez les libraires, voire dans les grandes surfaces, contribuent à masquer le problème. Destinés à l'utilisateur, ils présentent une vision idéalisée du développement d'applications qui ne correspond pas à la réalité. Combien de titres ne comportent-ils pas les mots *simple, facile, pour les nuls* ou *en un week-end*? Non, Excel n'est ni simple ni facile dès qu'on attaque des problèmes complexes. Comment pourrait-il en être autrement d'un logiciel dont la documentation de base, notoirement insuffisante¹, occupe plus de 1 800 pages? Non, SQL n'est plus un langage naturel et sûr, dès qu'on s'éloigne des requêtes élémentaires. Il ne l'est pas plus que la logique mathématique, dont il n'est qu'une expression lisible certes, mais maladroite, incomplète, irrégulière voire contradictoire (voir la section 6.10 par exemple).

Il existe peu de résultats publiés concernant la qualité des applications réalisées par des utilisateurs, tant en bases de données que sur tableurs. Nous citerons deux références qui abordent le problème. [Ronen, 1989] attire l'attention sur le problème de la qualité des modèles de calcul, propose une classification des erreurs et suggère une démarche de construction systématique et raisonnée. [Brown, 1987] va plus loin et fait état d'une analyse quantitative des types d'erreurs. Les résultats sont rien moins qu'inquiétants. Nous les résumons².

Neuf utilisateurs de tableurs chevronnés (1 à 5 années d'utilisation, 2,7 en moyenne) ont été soumis à des tests consistant à réaliser trois modèles simples en

1. D'où la prolifération d'ouvrages de complément, souvent indispensables, tels que [Kyd, 1992] ou [Blattner, 1999].

2. Voir aussi [Teo, 1999].

1-2-3 de Lotus, à partir d'énoncés d'une demi-page chacun. Tous les participants sauf un avaient une expérience en programmation classique. L'exigence portait sur la qualité et la précision de la solution, et non sur la rapidité de réalisation. Chaque problème a été résolu en moyenne en 41 minutes, les participants ont déclaré avoir confiance en leurs solutions (degré de confiance moyen de 4 sur une échelle de 1 à 5). Selon leur avis, les problèmes posés étaient plus simples que ceux qu'ils avaient généralement à résoudre. Les résultats sont les suivants :

- 44 % des modèles produisent des résultats faux (12 modèles sur 27),
- 63 % des modèles comportent des omissions ou produisent des résultats faux (17 modèles sur 27),
- sans compter les erreurs indirectes (induites par d'autres erreurs), 17 erreurs ont été détectées; 12 sont des erreurs de formules, 1 est directement visible à l'écran,
- il n'y a pas de corrélation entre la qualité des modèles et le degré de confiance annoncé par les participants,
- le nombre d'erreurs est fonction directe de la complexité du problème : les 9 solutions du problème le plus complexe comportaient des erreurs, alors que ce nombre est de 3 pour le plus simple.

Les auteurs ajoutent que d'autres études¹ font état d'un taux de 20 à 40 % de modèles opérationnels contenant des erreurs.

Les cours et les formations que nous avons organisés nous ont également donné quelques indices sur les performances des apprenants. La population est celle des étudiants de deuxième année en Sciences Économiques et Politiques et en Informatique, auxquels s'ajoutent les étudiants de troisième en Informatique. Les observations sont les suivantes :

Formulation de requêtes SQL de complexité moyenne (niveau : types 3, 4 et 5 selon le chapitre 4 de cet ouvrage) :

- 55 % des requêtes sont correctes,
- 15 % des requêtes comportent des erreurs syntaxiques (erreurs détectables),
- 25 % des requêtes comportent des erreurs de logique qui produiront des réponses fausses mais plausibles (erreurs difficilement détectables),
- 10 % des requêtes comportent des erreurs de logique qui produiront des réponses aberrantes (erreurs généralement détectables).

Construction d'un petit schéma de base de données SQL à partir d'un énoncé en français :

- 50 % des schémas sont corrects,
- 20 % des schémas contiennent des erreurs conceptuelles graves,
- 30 % des schémas contiennent des erreurs conceptuelles légères,

1. En particulier : Creet, R., *Micro-computer spreadsheets : their uses and abuses*, J. Account, Juin 1985.

- 15 % comportent des erreurs de traduction du schéma conceptuel en structures SQL.

Modèles de calcul, résolution des études de cas (chapitre 18, formulation des modèles abstraits uniquement, erreurs syntaxiques ignorées) :

- 40 % des modèles sont corrects,
- 20 % des modèles contiennent des erreurs de structure (erreurs d'indices par exemple),
- 30 % des modèles comportent des erreurs de logique qui produiront des réponses incorrectes mais plausibles, et donc difficiles à détecter,
- 20 % des modèles comportent des erreurs de logique qui produiront des réponses aberrantes.

Les études et les expériences confirment ce qui est une évidence pour l'informaticien : la difficulté principale ne réside pas dans la maîtrise de l'outil, mais dans l'élaboration des principes de l'application à développer. Les problèmes techniques étant désormais relativement bien maîtrisés, reste alors la tâche la plus ardue, l'analyse du problème que l'utilisateur veut résoudre et la construction d'une solution logique correcte et facile à modifier.

C'est dans ce domaine que ce livre voudrait apporter une contribution.

1.3 OBJECTIF DE L'OUVRAGE

Cet ouvrage, s'il pourra intéresser l'informaticien de métier, est cependant aussi destiné à l'utilisateur final. Il tente de combler ce qui nous semble constituer un vide sur le plan méthodologique : l'absence de concepts et de méthodes qui soient à la portée de l'utilisateur et qui permettent l'analyse rigoureuse des problèmes et la construction de solutions qui seront immédiatement prises en charge par les gestionnaires de bases de données et par les tableurs.

En clair, l'ouvrage se propose d'aider l'utilisateur à mieux maîtriser le développement d'applications à l'aide de ces outils privilégiés de résolution de problèmes. Il est organisé en deux parties. La première est consacrée aux **bases de données** et à leurs gestionnaires et la deuxième aux **modèles de calcul** et aux tableurs. Une troisième partie est développée dans [Hainaut, 1994] et est désormais disponible sur le site Web de l'ouvrage. Elle aborde le couplage des modèles de calcul et des bases de données, ainsi que les technologies qui permettent ce couplage.

Ces parties ont une structure identique, illustrée par le schéma 1.1 :

- description des **concepts fondamentaux** (des bases de données et des modèles de calcul),
- description d'**outils représentatifs** (bases de données relationnelles et SQL, EXCEL),
- description d'une **méthode de conception** d'applications adaptée à l'utilisateur final (conception de bases de données, conception de modèles de calcul),

- résolution de deux **études de cas**. Ces deux cas sont identiques pour les deux parties.

Des exercices d'application clôturent les chapitres les plus importants de l'ouvrage.

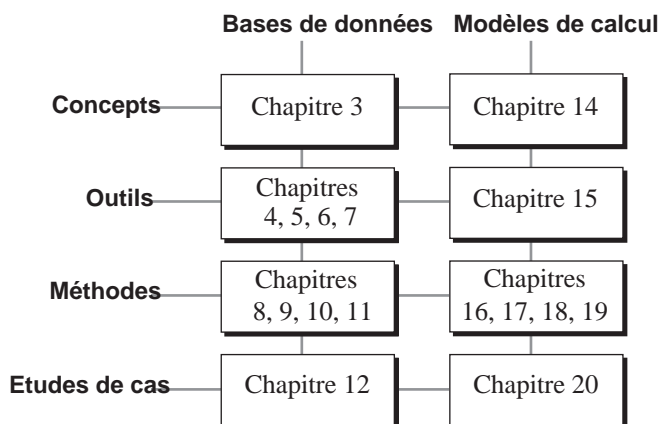


Figure 1.1 - Organisation de l'ouvrage

Le contenu de ce livre étant intrinsèquement de nature conceptuelle, il ne prétend se substituer ni aux manuels qui accompagnent les logiciels, ni aux nombreux ouvrages techniques de complément disponibles en librairie. En particulier, la description des outils inclut ce qui nous semble nécessaire et suffisant pour aborder le problème de la conception d'une base de données et d'une feuille de calcul.

PARTIE 1

LES BASES DE DONNÉES

Chapitre 2

Introduction

Ce chapitre présente, notamment dans une perspective historique, le concept de base de données à la fois comme un service de gestion de données destiné aux utilisateurs et comme une ressource, dont l'élaboration requiert des méthodes adéquates. Il décrit la structure de la première partie de l'ouvrage.

2.1 L'UTILISATEUR ET LES DONNÉES

Les données permanentes constituent certainement le matériau de base à partir duquel vont s'élaborer la plupart des applications informatiques. À tel point d'ailleurs que certaines de celles-ci se réduisent à gérer et consulter des données : répertoire d'adresses, références bibliographiques ou catalogue de pièces de rechange. Ces données sont rangées dans ce qu'on appelle un fichier et sont structurées en enregistrements. Chaque correspondant du répertoire d'adresses y est décrit par un enregistrement qui reprend son nom et son prénom, ainsi que ses coordonnées postales et téléphoniques. De tels fichiers sont gérés par des logiciels simples et intuitifs et ne posent guère de problèmes à leurs utilisateurs.

Les applications plus complexes réclament des données dont la structure est elle aussi plus complexe. Les données sont classées dans plusieurs fichiers en fonction des objets qu'elles décrivent : fichier des clients, fichier des produits, fichier des commandes, fichier des factures, etc. Il existe entre les fichiers des liens qui sont à l'image des relations entre les objets décrits. C'est via ces liens entre les données qu'on indique les commandes qui ont été émises par tel client, ou les produits qui sont référencés par telle commande. Il apparaît aussi que les données nécessaires à

une application pourraient être utiles à d'autres applications, voire même à d'autres utilisateurs. Ces données constituent alors ce qu'on appelle une **base de données**. Gérer de telles données n'est plus à la portée de logiciels élémentaires. Garantir la qualité des données enregistrées (trouve-t-on ce qu'on a enregistré?), leur cohérence (le client de chaque commande est-il répertorié?), les protéger en cas d'incident, permettre à plusieurs utilisateurs d'y accéder simultanément, tout en contrôlant strictement l'accès aux données confidentielles, offrir de bonnes performances d'accès à toutes les applications, en particulier celles qui sont interactives, sont des fonctions qui réclament des logiciels puissants et complexes, les **systèmes de gestion de bases de données**, ou **SGBD**.

Ce type de logiciels constitue l'un des outils fondamentaux de développement des grosses applications informatiques, mais aussi d'applications plus légères ou de serveurs Web. Ils existent sur toutes les plates-formes, depuis les gros ordinateurs (mainframes ou serveurs de grande puissance) jusqu'au PC portable ou au PDA, voire au téléphone mobile.

2.2 BASES DE DONNÉES ET SGBD RELATIONNELS

a) Principes

Une **base de données relationnelle** apparaît comme une collection de tables de données, ou fichiers *plats*. Il s'agit d'une structure extrêmement simple et intuitive qui, pour l'utilisateur du moins, ne s'encombre d'aucun détail technique concernant les mécanismes de stockage sur disque et d'accès aux données. La figure 3.6 montre une petite base de données relationnelle constituée des tables CLIENT, COMMANDE, DETAIL et PRODUIT. Ce schéma, qui est suffisant pour décrire les données et leur structure, ne dit rien des techniques d'implantation de ces données sur disque par exemple.

Toutes les manipulations s'effectuent au moyen d'un unique langage, SQL (*Structured Query Language*). Ce langage permet à l'utilisateur de demander au SGBD de créer des tables, de leur ajouter des colonnes, d'y ranger des données et de les modifier, de consulter les données, de définir les autorisations d'accès. Les instructions de consultation des données sont essentiellement de nature déclarative et non procédurale¹. On y décrit les propriétés des données qu'on recherche, notamment en spécifiant une condition de sélection, mais on n'indique pas le moyen de les obtenir, décision qui est laissée à l'initiative du SGBD.

Le langage SQL semble donc destiné tant au développeur informaticien qu'à l'utilisateur final. Il est vrai que des requêtes simples s'expriment simplement.

1. On qualifie de *procédurale* une suite d'instructions indiquant les opérations à exécuter pour obtenir un certain résultat. Si les instructions se contentent de décrire les caractéristiques du résultat recherché, alors elles sont de type *déclaratif*. L'instruction `select` d'SQL est de nature *déclarative* dans la mesure où l'utilisateur y décrit les données recherchées, mais ne peut spécifier la procédure permettant de les obtenir. C'est d'ailleurs le rôle du SGBD que de construire cette procédure et de l'exécuter.

Rechercher le nom et l'adresse de tous les clients de Toulouse s'exprime par la requête suivante, qui, appliquée à la table CLIENT comportant les colonnes NOM, ADRESSE et LOCALITE, ne réclame guère d'effort d'interprétation :

```
select NOM,ADRESSE
from   CLIENT
where  LOCALITE = 'Toulouse'
```

Bien sûr, l'extraction de données issues de plusieurs tables, et satisfaisant des conditions complexes, s'exprimera par des requêtes également plus complexes, qui nécessitent de la part de l'utilisateur qui les formule un apprentissage adéquat. On imagine aisément en effet que si on désire obtenir la liste des localités, accompagnées des quantités totales commandées par produit, et ordonnées par nombre décroissant de clients dont le compte est négatif, il nous faudra rédiger une requête beaucoup plus élaborée que celle qui vient d'être proposée.

Le marché offre aujourd'hui nombre d'outils qui permettent d'accéder à une base de données relationnelle sans qu'il soit nécessaire de maîtriser le langage SQL; citons notamment MS Access et FileMaker Pro. Ces modes d'accès sont cependant limités à des requêtes simples.

b) De l'algèbre à l'industrie : un peu d'histoire

Les bases de données relationnelles trouvent leur origine dans des travaux théoriques sur les structures de données des années 60. La première présentation officielle de l'approche relationnelle est incontestablement l'article de E.F. Codd, chercheur d'IBM, paru en 1970 [Codd, 1970]. L'auteur y proposait une structure de données tabulaire minimale, indépendante des technologies de mise en œuvre, accompagnée d'opérateurs d'extraction de données. Ce modèle de données est par essence mathématique : les ensembles de valeurs sont des domaines, les tables correspondent à des relations sur les domaines et les opérateurs sont des extensions de ceux de l'algèbre des relations.

Il restait à rendre ces propositions opérationnelles. C'est ainsi que sont nés les premiers SGBD prototypes : System R d'IBM et INGRES à l'Université de Berkeley entre autres. Sont également apparus les premiers langages de requête dérivés des langages mathématiques d'origine, mais accessibles aux praticiens : SEQUEL, qui deviendra SQL, pour le System R [Chamberlin, 1974] et QUEL pour INGRES² [Date, 1990]. Les premiers SGBD relationnels commerciaux sont apparus rapidement. Dès 1979, ORACLE (alors Relational Software), puis SQL/DS chez IBM (1981) ont été introduits sur le marché, ainsi qu'une version industrielle d'INGRES. D'autres ont suivi, tels INFORMIX, DB2, UNIFY, RDB, SYBASE et SQL-Server. A l'heure actuelle, les SGBD relationnels, et leurs extensions relationnel-objet, ont investi tous les types de machines, au point qu'il faut désormais les

2. Bien que QUEL ait été généralement jugé supérieur à SQL, ce dernier l'a rapidement supplanté sur le plan commercial.

considérer comme formant la famille principale de SGBD, renvoyant les autres vers des niches confidentielles. Cette situation découle sans doute des qualités intrinsèques du modèle relationnel et du langage SQL, mais aussi, plus prosaïquement, d'un phénomène irrésistible de standardisation. Les arguments sont imparables : portabilité des applications, indépendance par rapport au fournisseur du SGBD, stabilité de la formation, mobilité et interchangeabilité des développeurs, disponibilité d'outils annexes, pour n'en citer que quelques-uns. SQL fait d'ailleurs l'objet d'efforts constants de normalisation au niveau de l'ANSI, mais aussi de l'ISO [ANSI, 1989], dont il est le membre le plus actif. Nous verrons cependant que le concept de norme est interprété avec beaucoup de sens poétique par les éditeurs de SGBD. Les extensions vont actuellement (via la norme SQL:1999) vers une intégration avec le Web, un enrichissement des structures de données, le couplage avec le langage Java et XML, l'intégration de fonctions de gestion de données multimédia et de données spatiales, ainsi que des fonctions de fouille de données (*data mining*).

2.3 CONSTRUCTION D'UNE BASE DE DONNÉES

L'interrogation, voire la gestion, d'une base de données sont donc désormais à la portée de l'utilisateur averti et motivé. Qu'en est-il de la construction elle-même de la base de données? Cette activité est traditionnellement le domaine réservé des informaticiens. Ils disposent en effet non seulement de modèles et de méthodes spécifiques qui leur permettent de définir progressivement les structures de la base de données, mais ils utilisent aussi des outils qui les aident dans cette activité : les ateliers logiciels. Le lecteur en contact avec des informaticiens aura sans doute entendu parler du modèle Entité-association, de la méthode MERISE, des notations UML ou des ateliers d'ingénierie logicielle AMC-Designer, Power-Designer, Rose, Designer-2000 ou MEGA, pour n'en citer que quelques-uns.

Toutes les méthodes de conception sont basées sur la même idée : découpler l'analyse du problème de l'implantation de la solution dans une machine. L'analyse du problème conduit au schéma conceptuel de la base de données, qui est une solution abstraite, c'est-à-dire indépendante de la technologie, et qui s'exprime le plus souvent sous une forme graphique du modèle Entité-association³. La seconde phase, l'implantation, consiste à traduire le schéma conceptuel en une structure de tables et en instructions SQL de création de ces tables. Le lecteur pressé comparera la figure 12.1, qui décrit de manière abstraite la gestion d'un parc zoologique, avec le texte SQL qui définit les tables devant contenir les données relatives à cette gestion (section 12.2.4).

S'il est hors de question, dans un tel ouvrage, de tenter de former l'utilisateur à ces méthodes et à ces outils, il est cependant possible d'en retirer un ensemble de

3. Appelé également *Entité-Relation*, ou *Individuel* ou *Entity-Relationship*. On utilise aussi le terme de modèle de classes (UML), emprunté aux approches orientées-objets, mais qui recouvre un concept similaire.

concepts et de principes qui lui seront précieux non seulement dans ses contacts avec les informaticiens, mais aussi pour l'aider à mener à bien ses propres développements. C'est la raison pour laquelle nous proposerons une version simplifiée, mais opérationnelle, du modèle Entité-association, ainsi qu'une démarche simple et intuitive de construction d'une base de données.

Un peu d'histoire

Dans les années 70, il était de règle de considérer que les modèles de données offerts par les SGBD constituaient un mode d'expression adéquat lors de la phase d'analyse du problème. C'est ainsi qu'on a assisté, durant toute cette décennie, à une grande concurrence entre le modèle réseau⁴, le modèle hiérarchique⁵ et le modèle relationnel, chacun étant proposé par ses supporters comme le formalisme conceptuel idéal. Il est cependant rapidement apparu que tant les modèles de SGBD (incomplets et trop techniques) que le modèle relationnel (trop pauvre) étaient inadéquats comme support de raisonnement abstrait. Sont alors apparus des formalismes de description de structures de données à la fois plus abstraits que les modèles des SGBD et plus riches que le modèle relationnel : les modèles conceptuels.

Malgré quelques tentatives d'extension du modèle relationnel [Codd, 1979], le modèle Entité-association s'est rapidement imposé comme modèle conceptuel. Développé dès le début des années 70, notamment en Europe [Deheneffe, 1974], et popularisé par [Chen, 1976], il propose une représentation explicite des entités du domaine d'application, de leurs associations et de leurs attributs. Son succès est aussi dû à l'existence d'une représentation graphique des concepts du domaine d'application. La plupart des méthodes de conception de bases de données, et plus généralement de systèmes d'information, telles que MERISE sur le marché français (et même francophone), ont adopté ce modèle. Des modèles tels que NIAM, basés sur des associations binaires entre objets dotées d'une interprétation linguistique, ont aussi leurs adeptes, bien que leur percée ait été plus modeste face au modèle Entité-Association. Le dernier avatar de ce dernier, le modèle de classes d'UML, bien que poussé par une vague commerciale sans précédent, pose quelques problèmes dont on discutera dans la section 9.10.

2.4 DESCRIPTION DE LA PREMIÈRE PARTIE

La première partie de cet ouvrage est constituée de quatre blocs.

- *Les concepts* : le chapitre 3 décrit les concepts fondamentaux des bases de données : tables, colonnes, identifiants et clés étrangères.
- *Les outils* : les chapitres 4 et 5 sont consacrés à un exposé des deux volets principaux du langage SQL : le DDL et le DML. Il suit une approche pédagogique

4. Principalement le modèle CODASYL, inspiré d'IDS (Honeywell), qui a donné naissance à de nombreux SGBD, dont certains sont toujours en activité (IDMS, IDS2, UDS, MDBS).

5. Principalement le modèle IMS d'IBM.

qui va de l'expression de requêtes simples vers celle de requêtes complexes, en mettant l'accent sur les difficultés les plus fréquentes. Le chapitre 6 décrit des techniques SQL avancées tandis que le chapitre 7 présente quatre applications pratiques particulièrement utiles, mais un peu plus complexes.

- *Les méthodes de conception* : les chapitres 8 à 11 décrivent les principes de la construction systématique et raisonnée d'une base de données : modèle Entité-association (chapitre 9), élaboration du schéma conceptuel (chapitre 10), production du schéma de la base de données (chapitre 11).
- *Des études de cas* : deux problèmes typiques sont discutés et résolus comme illustration de la méthode de conception (chapitre 12).

2.5 POUR EN SAVOIR PLUS

Le lecteur à la recherche d'une introduction plus substantielle aux principes des bases de données se tournera vers des ouvrages tels que [Delobel, 1981], [Pichat, 1990], [Date, 2001] ou [Bouzeghoub, 1998]. Les références en anglais ne sont pas à négliger, telles que [Date, 1999], [Elmasri, 2000] ou [Connolly, 2002]. Des ouvrages tels que [Bouzeghoub, 1997] et [Gardarin, 1999] proposent une description des nouvelles tendances dans les SGBD.

Le langage SQL fait l'objet d'innombrables monographies de tous niveaux, et pour tous les SGBD. Bornons-nous à citer [Delmal, 2001], et, en langue anglaise, [Date, 1997] et [Melton, 2002]. Le couplage des bases de données et XML est développé dans [Gardarin, 2003].

Dans le domaine méthodologique, on pourra recommander, parmi d'autres, [Batini, 1992], [Blaha, 1998], [Ceri, 1997]. Les ouvrages en langue française sont également nombreux : citons seulement [Nancy, 1996] comme référence de base en ce qui concerne la méthode MERISE. Le lecteur trouvera dans [Bodart, 1994], [Hainaut, 1986], [Bouzeghoub, 1990], [Rolland, 1991], [Akoka, 2001] quelques approches similaires ou alternatives. Les références [Habrias, 1988] et [Halpin, 1995] présentent les méthodes NIAM et ORM.

La lecture des articles qui sont à l'origine des avancées technologiques et méthodologiques majeures est toujours enrichissante. On recommandera en particulier [Codd, 1970] et [Chen, 1976] où le lecteur trouvera un exposé clair et argumenté de concepts considérés encore aujourd'hui comme novateurs.

Chapitre 3

Concepts des bases de données

Ce chapitre décrit les notions essentielles des bases de données. Les données se présentent sous la forme de tables formées de lignes et de colonnes. Chaque ligne représente une entité ou un fait du domaine d'application, tandis qu'une colonne représente une propriété de ces entités ou faits. Une table contient donc des informations similaires sur une population d'entités ou de faits. Certaines colonnes ont pour but d'identifier les lignes (identifiants), d'autres sont des références vers d'autres lignes (colonnes de référence et contraintes référentielles). On propose une représentation graphique de la structure des tables ainsi qu'un exemple de base de données servant à illustrer les concepts examinés dans d'autres chapitres. On étudiera aussi le phénomène de redondance interne, qui conduira au concept de table normalisée.

3.1 TABLE, LIGNE ET COLONNE

Les données d'une base de données sont organisées sous la forme d'une ou plusieurs tables. Une **table** contient une collection de lignes stockées sur un support externe, généralement un disque. Une **ligne** est elle-même une suite de (une ou) plusieurs **valeurs**, chacune étant d'un type déterminé. D'une manière générale, une ligne regroupe des informations concernant un objet, un individu, un événement, etc., c'est-à-dire un concept du monde réel (externe à l'informatique), que nous appelons parfois une **entité** ou un **fait**.

La figure 3.1 représente une table comportant huit lignes, décrivant chacune un client. On y trouve quatre valeurs représentant respectivement le NOM et l'ADRESSE du client, la LOCALITE où il réside ainsi que l'état de son COMPTE. L'une de ces lignes représente le fait suivant :

Il existe un client de nom AVRON, résidant 8, chaussée de la Cure à Toulouse, et dont le compte est débiteur d'un montant de 1 700¹.

Toutes les lignes d'une table ont même **format** ou **structure**. Cette propriété signifie que dans la table CLIENT, toutes les lignes sont constituées d'une valeur de NOM, d'une valeur d'ADRESSE, d'une valeur de LOCALITE ainsi que d'une valeur de COMPTE. L'ordre des lignes est indifférent².

CLIENT			
NOM	ADRESSE	LOCALITE	COMPTE
HANSENNE	23, a. Dumont	Poitiers	1.250,00
MERCIER	25, r. Lemaître	Namur	-2.300,00
TOUSSAINT	5, r. Godefroid	Poitiers	0,00
VANBIST	180, r. Florimont	Lille	720,00
GILLET	14, r. de l'Eté	Toulouse	8.700,00
FERARD	65, r. du Tertre	Poitiers	350,00
AVRON	8, ch. de la Cure	Toulouse	-1.700,00
NEUMAN	40, r. Bransart	Toulouse	0,00

Figure 3.1 - Structure et contenu de la table CLIENT

L'ensemble des valeurs de même type correspondant à une même propriété des entités décrites s'appelle une **colonne** de la table. On parlera de la colonne NOM ou de la colonne COMPTE de la table CLIENT. 'HANSENNE' est la **valeur** de la colonne NOM de la première ligne. On définira une colonne par son nom, le type de ses valeurs et leur longueur. Les valeurs de la colonne NOM sont constituées de 1 à 32 caractères et celles de la colonne COMPTE de 9 positions numériques, dont 2 après la virgule décimale (ou point décimal pour un logiciel anglo-saxon).

Il est possible d'*ajouter* des lignes à une table et d'*en supprimer*. Il est possible également de *modifier* la valeur d'une colonne d'une ligne, ou plus généralement d'un sous-ensemble des lignes. On pourrait par exemple, à titre de bonus, ajouter 5 % à la valeur de COMPTE des lignes dont COMPTE > 500.

Les trois tables décrites à la figure 3.2 constituent une petite base de données. La première table, de nom FOURNISSEUR, décrit des fournisseurs, dont elle spécifie le numéro (NUMF), le nom (NOMF) et la ville de résidence (VILLEF). La deuxième

1. Dans cet ouvrage, nous laisserons l'unité monétaire indéterminée.

2. On ne peut donc en principe parler de la première, de la troisième ou de la dernière ligne d'une table. Pour être plus précis, on dira que le contenu d'une table à un instant déterminé est défini comme un *ensemble* de lignes au sens mathématique du terme. Les éléments d'un ensemble sont non ordonnés et distincts. Cependant, afin d'illustrer certains concepts à l'aide des figures du texte, il nous arrivera de parler, par exemple, de *la première ligne* d'une table.

table, nommée *PIECE*, décrit des pièces de rechange, caractérisées par leur numéro (*NUMP*) et leur type (*TYPE*). La troisième table, de nom *OFFRE*, représente les offres des fournisseurs pour les pièces qu'ils peuvent livrer. Ces offres sont caractérisées par le numéro du fournisseur (*NUMFL*) pouvant effectuer la livraison, le numéro de la pièce livrable (*NUMPL*) et le prix (*PRIX*) auquel ce fournisseur propose cette pièce.

FOURNISSEUR		
NUMF	NOMF	VILLEF
46	GERCIN	Paris
81	DUMONT	Paris
152	MERCIER	Tours
174	CHARLES	Nevers
259	CHARLES	Liège
376	RENIER	Nevers

OFFRE		
NUMFL	NUMPL	PRIX
46	15	46
46	57	32
81	14	65
81	15	48
152	14	62
152	15	46
152	57	34
174	57	32

PIECE	
NUMP	TYPE
14	BOULON
15	BOULON
57	ECROU

Figure 3.2 - Un exemple de base de données

3.2 RÔLES D'UNE COLONNE

On admet qu'une ligne regroupe des informations sur une *entité* ou un *fait* du monde réel. Dans cette optique, la valeur d'une colonne représente une propriété de cette entité. Cependant, toutes les colonnes ne jouent pas le même rôle vis-à-vis des entités représentées par les lignes d'une table.

Une analyse plus précise de l'exemple de la figure 3.2 permet de déceler trois types de colonnes dans cette base de données, selon le rôle qu'elles y jouent.

3.2.1 Les identifiants

Ce premier type de colonne permet d'identifier une entité, et donc aussi la ligne qui la représente dans la table. Tel est le cas de *NUMF* pour les lignes de *FOURNISSEUR* et *NUMP* pour celles de *PIECE*. Une telle colonne est appelée l'**identifiant** de la table³. Déclarer que *NUMF* est l'identifiant de *FOURNISSEUR*, c'est imposer qu'à tout instant les lignes de cette table aient des valeurs distinctes de *NUMF*.

3. La terminologie standard des bases de données relationnelles propose le terme de *clé* ou *key* pour désigner ce concept. Etant donné le grand nombre d'acceptions de ce terme dans le domaine des bases de données, nous lui préférons celui d'identifiant.

3.2.2 Les clés étrangères

Une colonne du deuxième type est une copie de l'identifiant d'une autre table. Chacune de ses valeurs joue le rôle d'une référence à une ligne de cette table. On l'appellera **colonne de référence**, ou selon la terminologie standard, **clé étrangère** (*foreign key*). La table OFFRE contient deux clés étrangères : NUMFL, qui constitue une référence à une entité fournisseur (et donc aussi à une ligne de FOURNISSEUR), et NUMPL, qui est une référence à une pièce (et aussi à une ligne de PIECE). C'est par des clés étrangères qu'on peut mettre en relation des lignes dans des tables distinctes. Pour chaque offre, représentée par une ligne de OFFRE, il est possible de connaître les informations concernant le fournisseur (*via* NUMFL) et celles qui concernent la pièce (*via* NUMPL).

On notera que le nom d'une colonne formant une clé étrangère est indépendant de celui de l'identifiant qu'elle référence. Il arrivera souvent qu'on donne à une clé étrangère le nom de l'identifiant cible (NUMP dans OFFRE), mais on pourrait aussi lui donner celui de la table référencée (PIECE), ou encore celui du rôle que jouent les lignes référencées (PIECE_OFFERTE)

3.2.3 Les informations complémentaires

Le troisième type de colonne ne joue d'autre rôle que celui d'apporter une *information complémentaire* sur l'entité. Ainsi en est-il de NOMF, VILLEF, TYPE et PRIX.

A partir de ces éléments de base, nous mettrons encore en évidence quatre concepts importants : les identifiants et clés étrangères multicomposants, les identifiants primaires, les contraintes référentielles et les colonnes facultatives.

3.2.4 Les identifiants et clés étrangères multicomposants

Rien n'interdit que l'identifiant d'une table soit constitué de *plus d'une colonne*. On pourrait ainsi imposer que les colonnes (NUMFL, NUMPL) forment l'identifiant de la table OFFRE. Cette propriété revient à dire qu'on n'enregistre qu'une seule offre par fournisseur pour une pièce déterminée, ou encore qu'une pièce ne peut faire l'objet que d'une seule offre de la part d'un fournisseur déterminé.

Par voie de conséquence, une clé étrangère qui référence une table ayant un identifiant primaire multicomposant est elle-même multicomposant.

3.2.5 Les identifiants primaires

Rien n'interdit non plus qu'on impose plus d'un identifiant à une table. Par exemple, une table qui reprend les informations signalétiques d'une population dotée d'une couverture sociale pourrait inclure, entre autres, une colonne NUMERO-INSCRIPTION et une colonne NUMERO-CARTE-IDENTITE, chacune constituant un identifiant. Parmi les identifiants d'une table, l'un est choisi comme le plus représentatif. Il sera déclaré **identifiant primaire**⁴, les autres étant les identifiants **secondaires**⁵ de

la table. Toute table possède un identifiant primaire et un nombre quelconque (éventuellement nul) d'identifiants secondaires⁶. Ceci a pour conséquence importante que les lignes d'une table sont distinctes, en accord avec la définition du contenu d'une table comme un *ensemble* de lignes.

Remarque sur les identifiants. Tout ensemble de colonnes qui comprend un identifiant est encore un identifiant : (NUMF, NOMF) est un identifiant de FOURNISSEUR. Un identifiant dont on ne peut retirer aucun composant sans qu'il perde sa qualité d'identifiant est appelé *identifiant minimal*. Il est évident qu'on cherchera à ne définir que des identifiants minimaux.

Remarque sur les clés étrangères. Bien que, théoriquement, la cible d'une clé étrangère soit l'un des identifiants de la table référencée, on convient de se limiter à l'identifiant primaire.

3.2.6 Les contraintes référentielles

Une valeur d'une clé étrangère constituée de la colonne RA d'une table B (ou, étant donné l'existence d'identifiants multicomposants, constituée d'un groupe de colonnes de référence) est destinée à désigner une ligne d'une table A. Concrètement, on peut imposer que pour toute valeur de RA dans B, il y ait une ligne de la table A identifiée par cette valeur. On en déduit une propriété très importante, dénommée **contrainte référentielle**. Celle-ci stipule que l'ensemble des valeurs d'une clé étrangère est à tout instant une partie de l'ensemble des valeurs de l'identifiant primaire de la table référencée. On imposerait par exemple que toute valeur de NUMPL dans OFFRE soit présente dans la colonne NUMP de PIECE (ce qu'on notera $\text{OFFRE.NUMPL} \subseteq \text{PIECE.NUMFL}$), et que toute valeur de NUMFL dans OFFRE se retrouve comme valeur NUMF de la table FOURNISSEUR. Il est donc interdit d'introduire dans la table OFFRE une ligne telle que (NUMFL:93, NUMPL:57, PRIX:32), puisqu'il n'existe dans FOURNISSEUR aucune ligne dont l'identifiant primaire NUMF ait la valeur 93.

3.2.7 Les colonnes facultatives

Il se peut qu'une information ne soit pas connue au moment où on introduit la description d'une entité ou d'un fait dans une table. Par exemple, on apprend que le fournisseur 259 offre désormais des pièces 15, mais on n'en connaît pas encore le prix. On admet alors d'introduire dans la table OFFRE la ligne incomplète suivante :

4. Selon la terminologie standard, *clé primaire* ou *primary key*.

5. Le terme de *secondaire* n'est pas standard. La littérature utilisera le terme de clés candidates pour désigner tous les identifiants d'une table. Un identifiant secondaire est donc une *clé candidate non primaire*.

6. En conformité avec les recommandations énoncées dans [Codd, 1990] et [Date, 1992], on s'en tiendra à des structures de tables avec identifiants. Signalons cependant que le langage SQL, que nous étudierons dans la suite, permet de définir des tables sans identifiants, possibilité que nous ignorerons ici.

(NUMFL:259, NUMPL:15, PRIX:-). On dira que la valeur de PRIX de cette ligne est inconnue. Cette latitude donnée à une colonne d'accepter l'absence de valeur pour certaines lignes n'est pas valable pour toute colonne. Par exemple, on peut décider que le nom d'un fournisseur est indispensable, et donc que toute ligne de la table FOURNISSEUR doit avoir une valeur de NOMF valable. La colonne PRIX sera déclarée **facultative**, tandis que la colonne NOMF sera **obligatoire**. En fait, une valeur absente sera représentée par la valeur conventionnelle `null`, dont nous reparlerons dans la section 6.10.

On impose que les composants de tout identifiant primaire soient obligatoires. En ce qui concerne les colonnes de référence, le concept de colonne facultative demande que nous affinions la définition de la contrainte référentielle :

Si un groupe de colonnes de la table B est déclaré *clé étrangère* vers la table A, ces colonnes sont soit obligatoires, soit facultatives; dans ce dernier cas, elle sont coexistantes, c'est-à-dire qu'à tout instant, pour toute ligne de la table B,

- soit des valeurs existent pour chacune de ces colonnes et dans ce cas elles doivent identifier une ligne existante de A,
- soit il n'y a de valeurs pour aucune de ces colonnes.

On n'admet donc pas qu'il existe des lignes pour lesquelles certains composants d'une clé étrangère possèdent une valeur tandis que les autres n'en ont pas⁷.

3.3 STRUCTURE ET CONTENU D'UNE BASE DE DONNÉES

On distingue deux parties distinctes dans une base de données : son **schéma** et son **contenu**. Le *schéma* d'une base de données en définit la structure en termes de tables, de colonnes (avec le type de valeurs et le caractère obligatoire ou facultatif de chacune), d'identifiants primaires et secondaires, et de clés étrangères. Son *contenu* à un instant déterminé est l'ensemble des lignes. En supprimant les lignes de la figure 3.2, nous obtenons le schéma (encore partiel à ce stade) représenté à la figure 3.3.

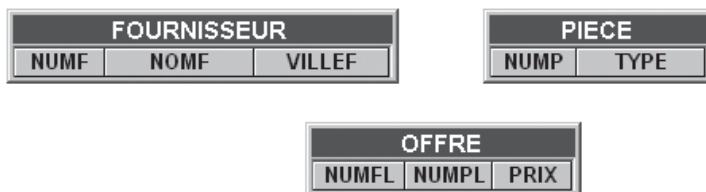


Figure 3.3 - Structure de la base de données de la figure 3.2

7. Il faut cependant noter que ce cas est généralement accepté par les SGBD. Nous l'excluons pour des raisons de simplicité et de sécurité.

Le contenu d'une base de données réelle est généralement volumineux (plusieurs millions de lignes) et susceptible d'évoluer constamment. Il est donc sans intérêt de le représenter dans un exposé comme celui-ci, sauf à titre illustratif. En revanche, le schéma comporte un nombre limité d'éléments (quelques dizaines à quelques milliers de tables en général) présentant une relative stabilité dans le temps : on ne modifie la structure d'une base de données que lorsque la structure de la réalité à représenter évolue. Notre intérêt se portera essentiellement sur la composition et les propriétés des schémas de bases de données.

3.4 REPRÉSENTATION GRAPHIQUE D'UN SCHÉMA

Nous pouvons à présent préciser et compléter les conventions graphiques de représentation d'un schéma. Nous proposons un premier jeu de conventions (figure 3.4).

- Une *table* et ses *colonnes*⁸ sont représentées par un cartouche contenant le nom de la table et celui de chaque colonne. Les colonnes sont placées dans un ordre quelconque. Cependant, afin de faciliter la compréhension, on veillera autant que possible à disposer de façon contiguë les colonnes d'un même identifiant de même que celles de chaque clé étrangère, et à placer à l'extrême gauche les colonnes de l'identifiant primaire.
- On soulignera d'un trait continu les noms des colonnes de l'*identifiant* primaire et d'un trait pointillé ceux de chaque identifiant secondaire.
- Le nom d'une colonne *facultative* sera entouré de parenthèses.
- Une clé étrangère constituée d'un groupe de colonnes de référence (clé étrangère multi-composant) sera représentée par une accolade.
- Une contrainte référentielle sera représentée par une flèche qui part du nom de la colonne de référence (ou de l'accolade en cas de groupe de référence) et qui pointe vers le cartouche de la table référencée.

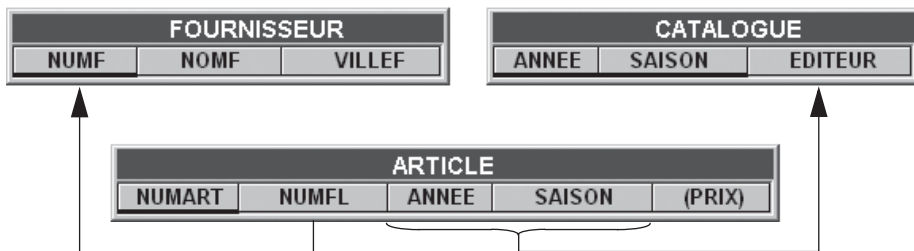


Figure 3.4 - Représentation des identifiants et des clés étrangères

8. On ne prévoit pas de convention particulière de représentation des types de valeurs.

3.5 UN EXEMPLE DE BASE DE DONNÉES

La base de données qui nous servira d'exemple dans cette section est structurée selon le schéma de la figure 3.5. On donnera une brève définition de la signification des tables et des colonnes

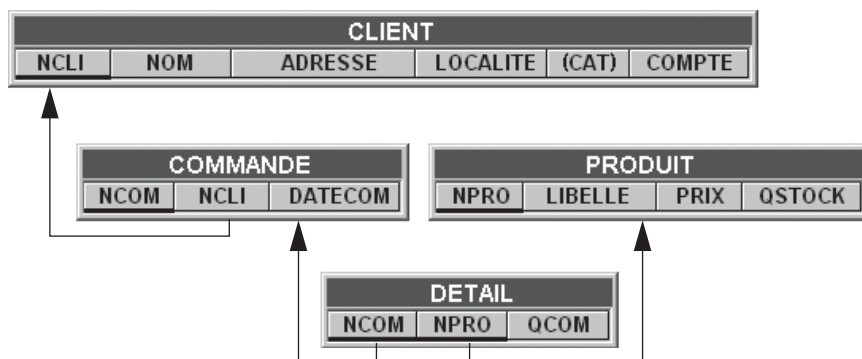


Figure 3.5 - Schéma de la base de données exemple

- Table **CLIENT** : chaque ligne décrit un client ; les colonnes décrivent successivement le numéro du client (NCLI), son nom (NOM), son adresse (ADRESSE), sa localité (LOCALITE), sa catégorie (CAT) et l'état de son compte (COMPTE). L'identifiant primaire est constitué de NCLI. La colonne CAT est facultative.
- Table **PRODUIT** : chaque ligne décrit un produit ; les colonnes décrivent successivement le numéro du produit (NPRO), son libellé (LIBELLE), son prix unitaire (PRIX) et la quantité restant en stock (QSTOCK). NPRO est l'identifiant primaire.
- Table **COMMANDE** : chaque ligne décrit une commande passée par un client ; les colonnes décrivent successivement le numéro de la commande (NCOM), le numéro du client qui a passé la commande (NCLI) et la date de la commande (DATECOM). NCOM est l'identifiant primaire de la table. NCLI est une clé étrangère vers la table CLIENT.
- Table **DETAIL** : chaque ligne représente un détail d'une commande ; les colonnes décrivent successivement le numéro de la commande à laquelle le détail appartient (NCOM), le numéro du produit commandé (NPRO) et la quantité commandée (QCOM). L'identifiant primaire est constitué de NCOM et NPRO. NCOM et NPRO sont en outre chacune une clé étrangère respectivement vers les tables COMMANDE et PRODUIT.

Le contenu de la base de données pourrait, à un instant donné, se présenter comme indiqué à la figure 3.6.

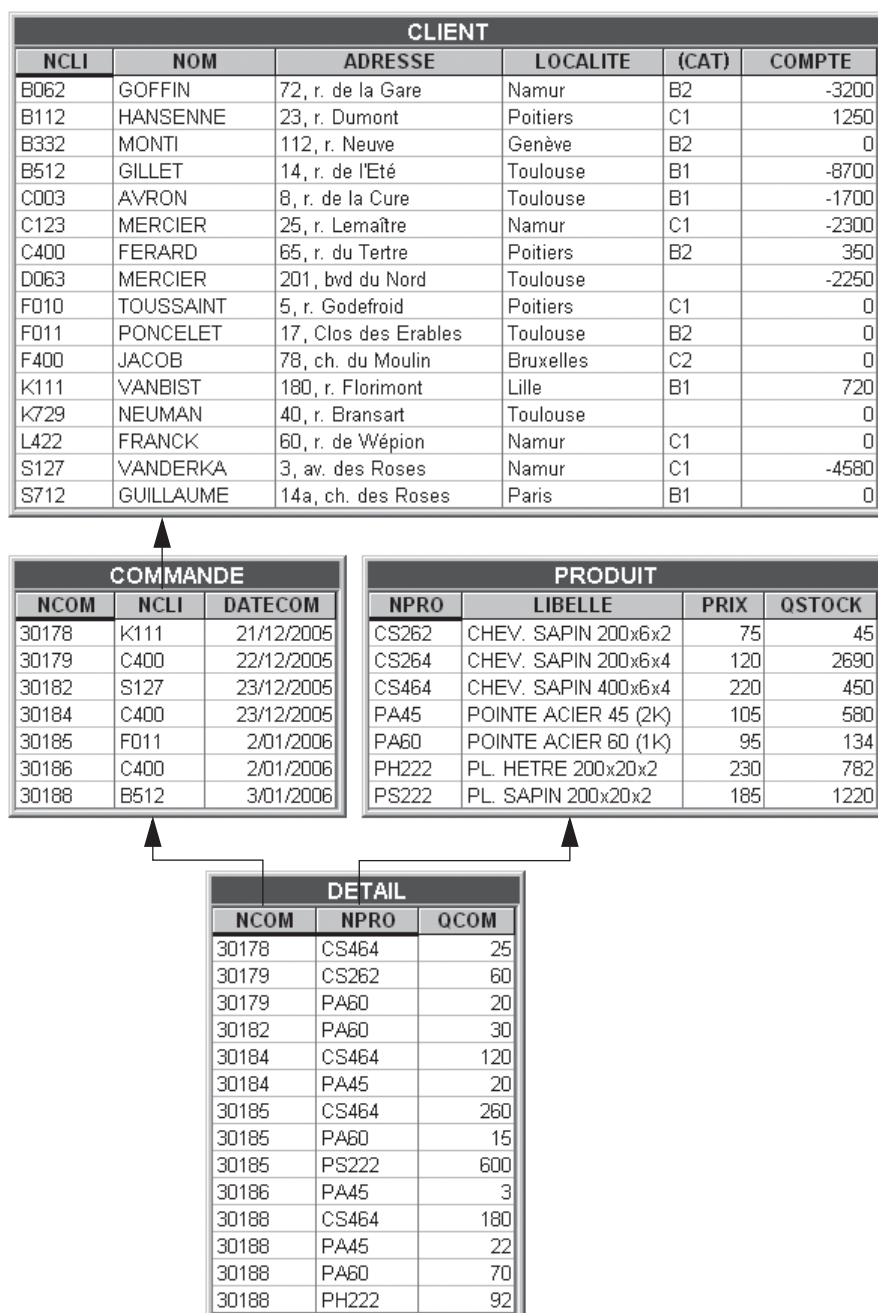


Figure 3.6 - Contenu typique de la base de données exemple

3.6 AUTRES NOTATIONS GRAPHIQUES

La notation de représentation adoptée dans ce chapitre est simple et naturelle. Elle ne conviendra cependant pas pour dessiner des schémas plus volumineux et plus complexes.

On pourrait adopter une disposition telle que celle d'Access de Microsoft (figure 3.7), selon laquelle les noms des colonnes sont présentés en liste verticale. L'identifiant primaire est indiqué en gras et les contraintes référentielles par un trait reliant la clé étrangère (symbole ∞) à l'identifiant primaire (symbole 1).

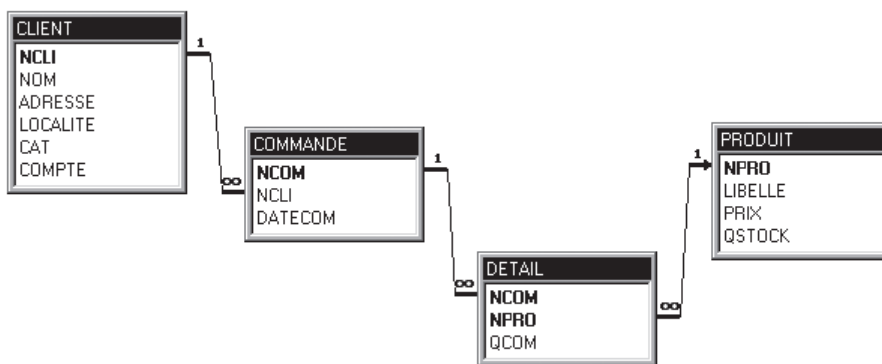


Figure 3.7 - Représentation graphique des schémas Microsoft Access

Ces symboles se lisent *un-à-plusieurs*, indiquant par là qu'à un élément du côté 1 correspondent plusieurs (∞) éléments du côté ∞ , et qu'à un élément du côté ∞ correspond un (1) élément du côté 1.

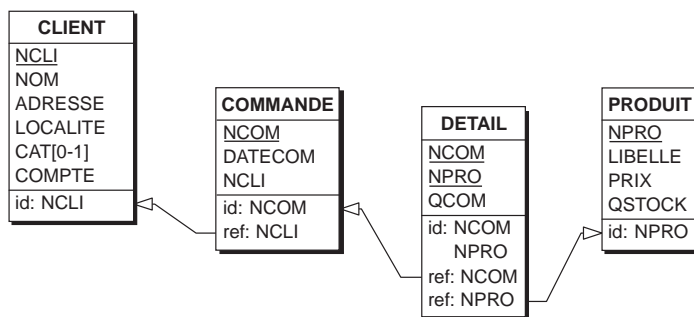


Figure 3.8 - Représentation graphique plus complète (atelier DB-MAIN)

La notation d'Access ne permet cependant pas de représenter de manière graphique certaines constructions qui nous seront utiles⁹, ce qui nous amènera à utiliser, lorsque cela sera nécessaire, une notation plus complète telle que celle qui est illus-

9. Les colonnes facultatives, les identifiants secondaires, les clés étrangères cycliques, des identifiants et/ou clés étrangères non disjoints, les contraintes d'existence, les structures objet-relationnelles, etc.

trée dans la figure 3.8, et qui est propre à l'atelier d'ingénierie DB-MAIN dont nous reparlerons plus loin. Les conventions de cette notation sont les suivantes :

- une table est représentée par une boîte à trois compartiments indiquant successivement le nom de la table, le nom de ses colonnes et les contraintes d'intégrité;
- une colonne facultative est caractérisée par le symbole [0-1] qui indique qu'une ligne possède de 0 à 1 valeur pour cette colonne¹⁰;
- l'identifiant primaire est spécifié par la clause "id:" qui énumère ses composants; en outre, ceux-ci sont soulignés dans le compartiment des colonnes;
- tout identifiant secondaire est spécifié par une clause similaire "id:";
- une clé étrangère est spécifiée par une clause "ref:" qui énumère ses composants; de cette clause est issue une flèche qui pointe vers l'identifiant référencé (celui-ci pourrait être secondaire, ce qu'on ignorera dans cet ouvrage);
- un groupe de colonnes qui forme à la fois un identifiant et une clé étrangère est simultanément noté "id:" (ou "id:") et "ref".

3.7 NOTE SUR LES CONTRAINTES RÉFÉRENTIELLES

Un schéma qui comporte un identifiant constitué de deux (ou plusieurs) clés étrangères doit faire l'objet d'une attention toute particulière lorsque cet identifiant est lui-même visé par une clé étrangère. Considérons par exemple (figure 3.9) une table COMPTE, dont chaque ligne représente le crédit ouvert par un client (table CLIENT) auprès d'un fournisseur (table FOURNISSEUR). Chaque achat du client (table ACHAT) est attaché à un compte.

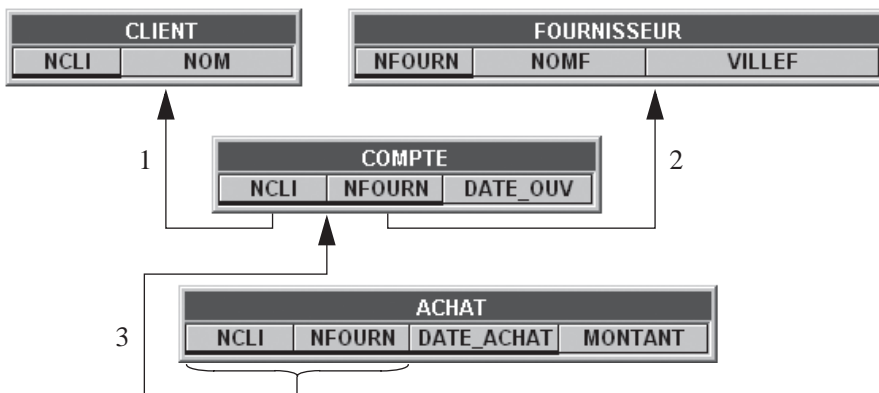


Figure 3.9 - Contraintes référentielles complexes

La contrainte référentielle relative à la clé étrangère de la table ACHAT stipule que *toute valeur de (NCLI, NFOURN) d'ACHAT doit se retrouver dans une ligne de*

10. En fait toute colonne possède une telle caractéristique; cependant la valeur la plus fréquente [1-1] n'est pas représentée pour alléger le dessin.

COMPTE (propriété 3). Étant donné que l'identifiant (NCLI, NFOURN) de COMPTE est constitué de deux clés étrangères, on a aussi les propriétés suivantes :

- toute valeur de NCLI de COMPTE est une valeur de NCLI de CLIENT (propriété 1),
- toute valeur de NFOURN de COMPTE est une valeur de NFOURN de FOURNISSEUR (propriété 2).

Des propriétés 1, 2 et 3, on déduit alors que (figure 3.10) :

- toute valeur de NCLI de ACHAT est une valeur de NCLI de CLIENT (propriété 4),
- toute valeur de NFOURN de ACHAT est une valeur de NFOURN de FOURNISSEUR (propriété 5).

Cependant, les propriétés 4 et 5 ne peuvent jamais se substituer à la propriété 3, comme le suggère le schéma de la figure 3.10.

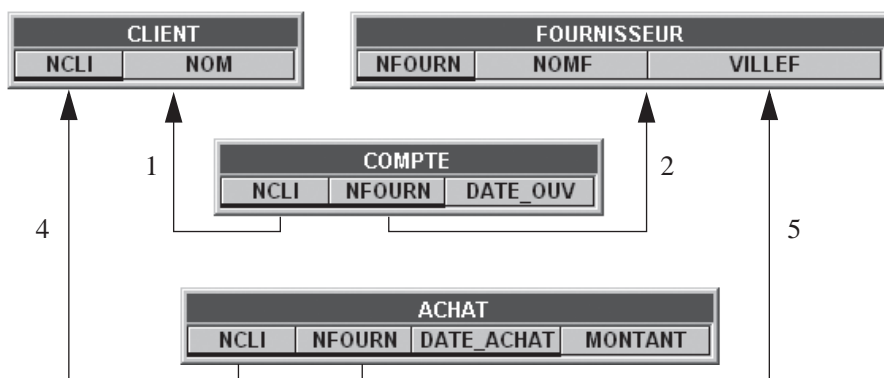


Figure 3.10 - Une version incorrecte du schéma 3.9

Par exemple, un achat représenté par (C123, F445, 14/04/2006, ...) est valide, non pas par l'existence d'un client n° C123 et d'un fournisseur n° F445, mais parce qu'il existe un compte identifié par (C123, F445), auquel il est attaché¹¹. Il s'agit d'une erreur qui apparaît fréquemment chez les *modélisateurs* débutants.

11. Plus précisément, cette discussion se base sur deux propriétés importantes concernant les règles d'inclusion, et donc les clés étrangères. La première (dite de *décomposition*) veut que si $U.(A,B) \subseteq T.(A,B)$, alors on a aussi que $U.A \subseteq T.A$ et $U.B \subseteq T.B$; l'inverse n'est cependant pas vraie. La seconde (dite de *transitivité*) stipule que si $U.A \subseteq T.A$ et $T.A \subseteq R.A$, alors on a aussi $U.A \subseteq R.A$. Étant donné les propriétés ($T.A \subseteq R.A$) et ($T.B \subseteq S.B$), la propriété ($U.(A,B) \subseteq T.(A,B)$) permet d'inférer ($U.A \subseteq R.A$) et ($U.B \subseteq S.B$). Mais, une fois encore, l'inverse n'est pas vraie. Dans cette formulation, $R(A,...)$, $S(B,...)$, $T(A,B,...)$ et $U(A,B,...)$ sont des schémas de tables, $R.A$ désigne l'ensemble des valeurs de A dans R, et $U.(A,B)$ désigne l'ensemble des couples de valeurs de (A,B) dans U.

3.8 MODIFICATION ET CONTRAINTES D'INTÉGRITÉ

Les propriétés structurelles (identifiant, contrainte référentielle, colonne obligatoire/facultative) associées aux données doivent être respectées à tout instant; elles constituent donc des contraintes imposées aux opérations de modification de ces données. Ajouter une ligne, supprimer une ligne ou modifier une valeur de colonne d'une ligne sont des opérations qui ne sont autorisées que si ces propriétés sont toujours respectées par les données après l'opération. Si ces propriétés sont violées, on dira que les données ont perdu leur intégrité. Ces propriétés constituent dès lors des **contraintes d'intégrité**. Nous examinerons brièvement l'impact de ces contraintes sur les opérations de modification du contenu d'une base de données.

3.8.1 Les contraintes d'unicité (identifiants)

Un identifiant constitue une contrainte d'unicité imposant qu'à tout instant les lignes d'une table possèdent des valeurs distinctes pour une ou plusieurs colonnes. Il ne peut exister, à aucun moment, plus d'une ligne de CLIENT ayant la même valeur de NCLI.

- *Création d'une ligne* : il ne peut exister de ligne possédant la même valeur de l'identifiant.
- *Suppression d'une ligne* : pas de contrainte.
- *Modification de l'identifiant d'une ligne* : il ne peut exister de ligne possédant la nouvelle valeur de l'identifiant.

3.8.2 Les contraintes référentielles (clés étrangères)

Une contrainte référentielle précise que certaines colonnes d'une table, appelées clé étrangère, doivent à tout instant, pour chaque ligne, contenir des valeurs qu'on retrouve comme identifiant primaire d'une ligne dans une autre table¹². C'est ainsi que la table DETAIL est soumise à deux contraintes référentielles. La première indique que toute valeur de NCOM doit identifier une ligne de COMMANDE. La seconde indique que toute valeur de NPRO doit identifier une ligne de PRODUIT.

Les règles qui régissent les opérations de modification des données sont plus complexes que pour les autres contraintes. Nous raisonnerons sur les tables CLIENT et COMMANDE (figure 3.11), cette dernière faisant l'objet d'une contrainte référentielle, puisque la valeur de NCLI de toute ligne de COMMANDE doit être présente dans la colonne NCLI d'une ligne CLIENT (en termes de la réalité de l'entreprise, *toute commande doit appartenir à un client enregistré*). On ignorera dans cette analyse l'impact des autres contraintes référentielles du schéma sur les opérations.

12. Il peut d'ailleurs s'agir de la même table.

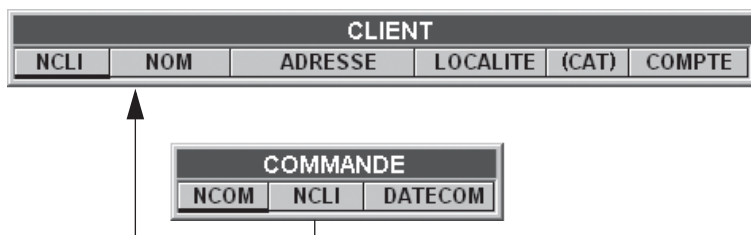


Figure 3.11 - Schéma d'étude de l'intégrité référentielle

*Création d'une ligne de **COMMANDE***

La valeur de NCLI de cette ligne doit être présente dans la colonne NCLI d'une ligne de CLIENT (si la colonne NCLI de COMMANDE avait été déclarée facultative, alors la valeur de NCLI de la ligne pourrait être absente).

*Suppression d'une ligne de **COMMANDE***

Effectuée sans restriction¹³.

*Modification de la valeur de NCLI d'une ligne de **COMMANDE***

La nouvelle valeur de NCLI de la ligne doit être présente dans la colonne NCLI d'une ligne de CLIENT (si NCLI de COMMANDE avait été déclarée facultative, alors on aurait pu effacer la valeur de NCLI de la ligne).

*Création d'une ligne de **CLIENT***

Effectuée sans restriction.

*Suppression d'une ligne de **CLIENT***

Le problème se pose lorsque ce client possède des commandes; il existe alors des lignes dans COMMANDE qui référencent la ligne de CLIENT à supprimer. On définit trois comportements possibles qui laissent la base de données dans un état correct.

- **Blocage.** Le premier consiste à refuser la suppression de la ligne de CLIENT afin d'éviter de laisser dans la base de données des lignes de COMMANDE *orphelines*.
- **Propagation ou cascade.** Le deuxième consiste à supprimer non seulement la ligne de CLIENT, mais aussi toutes les lignes de COMMANDE qui la référencent¹⁴.

13. Pour être tout à fait précis, n'oublions pas que DETAIL est aussi lié à COMMANDE par une contrainte référentielle. La suppression d'une ligne de COMMANDE n'a pas d'impact sur sa relation avec CLIENT, mais elle en a sur celle qui concerne DETAIL, que nous ignorons ici pour simplifier le raisonnement.

14. Et bien sûr aussi les lignes de DETAIL qui référencent les lignes de COMMANDE ainsi supprimées.

- **Indépendance.** Le troisième est possible lorsque la clé étrangère est constituée de colonnes facultatives. Il consisterait ici, si NCLI de COMMANDE avait été déclarée facultative, à effacer la valeur de la colonne NCLI des lignes de COMMANDE qui référencent la ligne de CLIENT à supprimer. De la sorte, ces commandes n'appartiennent plus à aucun client après l'opération.

Modification de NCLI d'une ligne CLIENT

Si aucune ligne de COMMANDE ne référence cette ligne de CLIENT, la contrainte référentielle n'impose pas de restriction. Si au contraire de telles lignes existent, on admet trois comportements similaires à ceux de l'opération de suppression : refus de modification, modification des valeurs de clés étrangères qui référencent cette ligne, effacement des valeurs de clés étrangères qui référencent cette ligne.

On voit donc que l'impact d'une contrainte référentielle n'est pas unique et que, selon la réalité à représenter et la politique de gestion des données, on sera amené à choisir l'un ou l'autre des comportements décrits.

3.8.3 Les colonnes obligatoires

Si une colonne est déclarée obligatoire, chaque ligne doit en posséder une valeur. Lors des opérations de création et de modification de lignes, cette colonne devra recevoir une valeur significative. Par exemple, il est permis de créer une ligne de CLIENT sans valeur pour CAT, mais pas sans valeur pour LOCALITE.

3.9 LA NORMALISATION

Les tables que nous avons rencontrées jusqu'ici représentaient chacune un ensemble d'entités clairement identifié : des fournisseurs, des clients, des commandes, des comptes ou des achats. Certaines tables peuvent présenter une structure plus complexe, généralement considérée comme indésirable. Tel est le cas de la table de la figure 3.12, que nous allons examiner de plus près.

LIVRE					
NUMERO	TITRE	AUTEUR	ISBN	DATE_ACHAT	EMPL
1029	L'humanité perdue	Finkelkraut A.	2 02 033300 7	14/10/2005	F3
1030	L'humanité perdue	Finkelkraut A.	2 02 033300 7	14/10/2005	F3
1032	Mercure	Nothomb A.	2 253 14911 X	14/10/2005	G5
1045	Eva Luna	Allende I.	2 253 05354 6	22/2/2006	F3
1067	Mercure	Nothomb A.	2 253 14911 X	24/2/2006	G5
1022	Mercure	Nothomb A.	2 253 14911 X	3/10/2005	G6

Figure 3.12 - La table LIVRE enregistre les informations sur les livres disponibles dans une bibliothèque

3.9.1 Le phénomène de redondance interne

Le propriétaire de la table LIVRE désire manifestement y enregistrer les livres de la bibliothèque dont il est responsable. Pour chaque livre, il a repris le numéro, le titre, l'auteur, le code ISBN, la date d'achat et son emplacement dans les rayonnages. Un livre qui fait l'objet d'une demande importante de la part des lecteurs peut être acquis en plusieurs exemplaires, qui font chacun l'objet d'une ligne distincte de la table.

Pour naturelle qu'elle puisse paraître, cette représentation pose cependant un problème : lorsqu'un livre existe en plusieurs exemplaires, les lignes décrivant ceux-ci reprennent les mêmes valeurs du titre, de l'auteur et du code ISBN.

On nomme ce phénomène *redondance d'information*, puisqu'une même information est enregistrée plusieurs fois. Si par inadvertance nous effaçons le titre du livre 1067, il serait aisé de le restaurer par une recherche d'un autre livre qui posséderait le même code ISBN, et qui aurait aussi le même titre. Une telle situation viole le principe fondateur des bases de données : *tout fait pertinent du domaine d'application doit y être enregistré une et une seule fois*. Sur le plan pratique, la redondance n'est pas sans inconvénients.

1. Avant tout, la table occupe un espace excessif.
2. Ensuite, la modification ultérieure du titre ou de l'auteur d'un livre exigera la même modification des lignes de tous les exemplaires de ce livre, à défaut de quoi les données deviendraient incohérentes.
3. Si l'enregistrement d'un premier exemplaire d'un livre peut se faire librement, celui des exemplaires suivants doit être conforme aux informations déjà présentes.
4. Plus subtilement encore, l'effacement du seul exemplaire d'un livre entraînerait la perte définitive des informations sur son titre et son auteur.

Cette analyse montre donc que la table est soumise à une **contrainte d'intégrité** d'un type nouveau : *si deux lignes ont la même valeur d'ISBN, alors elles ont les mêmes valeurs de TITRE et d'AUTEUR*. On dira aussi que *la valeur de TITRE (et d'AUTEUR) est principalement fonction de celle d'ISBN*, colonne qui n'est pas l'identifiant de la table.

3.9.2 Normalisation par décomposition

L'observation attentive des données contenues dans la table LIVRE montre que celle-ci contient des renseignements sur **deux** catégories d'entités : les livres eux-mêmes et les ouvrages dont les livres sont des exemplaires multiples. Il faudrait par exemple distinguer l'ouvrage *Mercure* d'A. Nothomb (ISBN 2 253 14911 X) des trois exemplaires de celui-ci que constituent les livres 1032, 1067 et 1022. Les données concernant un ouvrage sont enregistrées autant de fois que cet ouvrage a d'exemplaires.

La résolution de ce problème passe par la **décomposition** de la table LIVRE en deux tables distinctes, auxquelles nous donnerons de nouveaux noms, pour éviter toute ambiguïté (figure 3.13). La première, OUVRAGE, contient la description des ouvrages. On y reprend le code ISBN, le titre et l’auteur de chacun de ceux-ci. La seconde table, EXEMPLAIRE, décrit les exemplaires, éventuellement multiples, de ces ouvrages. On y indique, pour chacun d’eux, le numéro d’exemplaire, le code ISBN de l’ouvrage (référence à OUVRAGE), la date d’achat et l’emplacement dans les rayonnages.

Ce processus de décomposition en vue d’éliminer les redondances internes porte le nom de *normalisation*. Les tables de la figure 3.13 sont dites *normalisées*, alors que la table LIVRE ne l’est pas.

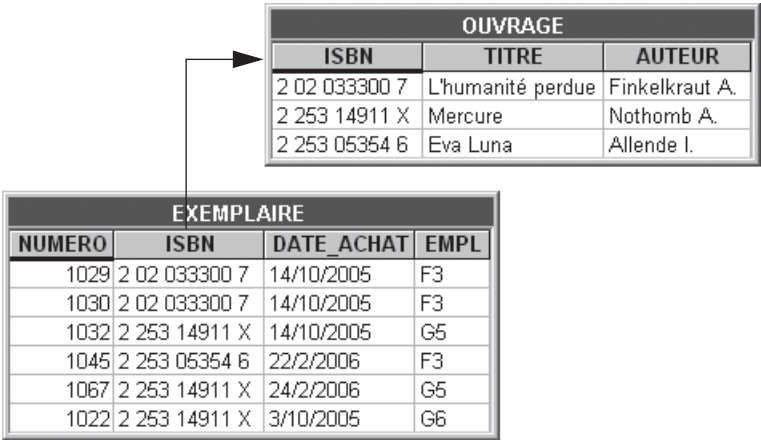


Figure 3.13 - Dans une base de données normalisée, on distingue les *exemplaires* des *ouvrages* dont ils sont la matérialisation. Les redondances présentes dans la table de la figure 3.12 sont ainsi éliminées

3.9.3 Analyse du phénomène

La propriété qui veut que *deux lignes qui ont la même valeur d’ISBN ont les mêmes valeurs de TITRE et d’AUTEUR* porte le nom de **dépendance fonctionnelle**, et se note

ISBN → TITRE, AUTEUR

en vertu du fait que cette propriété spécifie simplement une *fonction*, au sens mathématique du terme (à une valeur d’ISBN correspond *une seule* valeur de TITRE et d’AUTEUR). La partie gauche se nomme le *déterminant* de la dépendance et la partie droite le *déterminé*.

On notera deux propriétés importantes : (1) un identifiant d’une table est un déterminant de chacune des (autres) colonnes de la table, (2) inversement, toute colonne, ou groupe de colonnes, qui est un déterminant pour chacune des (autres) colonnes de la table est un identifiant. En particulier, on a :

NUMERO → TITRE, AUTEUR, ISBN, DATE_ACH, PLACE

Ces définitions et observations nous permettent de répondre à trois questions importantes.

1. Comment décomposer une table ?

La décomposition d'une table doit se faire *selon une dépendance fonctionnelle*, à défaut de quoi l'opération entraîne une perte de données, en ce sens qu'une recombinaison (qu'on appellera plus tard *jointure*) ne restitue pas les données initiales. Elle doit obéir au canevas de la figure 3.14, qui montre que la table R peut être remplacée par la table S, qui regroupe le déterminant et le déterminé de la dépendance, et d'une variante réduite de la table R, de laquelle le déterminé a été retiré. Les colonnes A, B et C sont en toute généralité des groupes de colonnes.

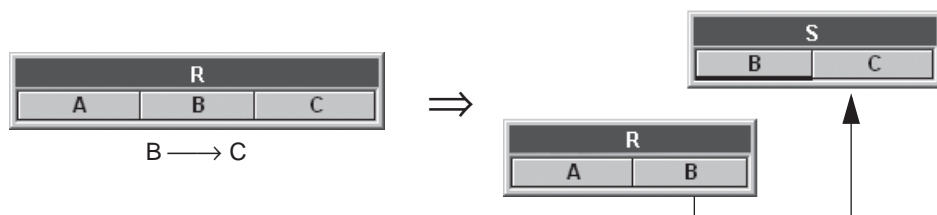


Figure 3.14 - Décomposition sans pertes selon une dépendance fonctionnelle

2. Qu'est-ce qu'une table normalisée ?

Une table est normalisée si *tout déterminant y est un identifiant*. À l'inverse, si une table est le siège d'une dépendance fonctionnelle *anormale*, c'est-à-dire dont le déterminant n'est pas un identifiant, alors cette table n'est pas normalisée, et est susceptible de contenir des données redondantes.

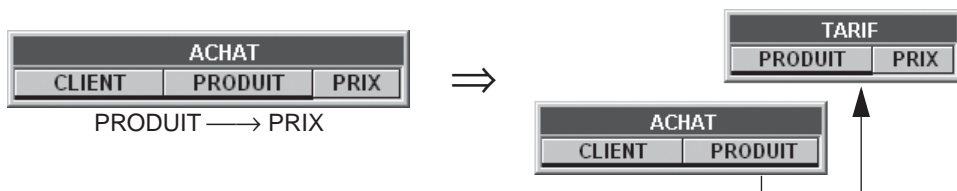
3. Comment traiter une table non normalisée ?

Toute table qui est le siège d'une dépendance anormale doit être décomposée selon cette dépendance, selon le canevas de la figure 3.14. Si cette table contient plusieurs dépendances anormales, on la décomposera itérativement en traitant, à chaque stade, celles dont le déterminé n'est pas un déterminant.

On observera que la décomposition de la table LIVRE en EXEMPLAIRE et OUVRAGE obéit à ces principes (ISBN est un déterminant mais n'est pas un identifiant). Examinons le traitement de deux exemples supplémentaires.

Les achats

Chaque ligne (c, p, x) de la table ACHAT ci-dessous représente le fait que le client c achète le produit p au prix x. On sait que le prix d'un produit est constant, quel que soit le client qui l'achète.



Cette table est donc le siège de la dépendance fonctionnelle $\text{PRODUIT} \longrightarrow \text{PRIX}$. Elle n'est pas normalisée, puisque son identifiant ($\text{CLIENT}, \text{PRODUIT}$) est différent du déterminant de la dépendance. Il est donc nécessaire de la décomposer en une table TARIF et une table ACHAT .

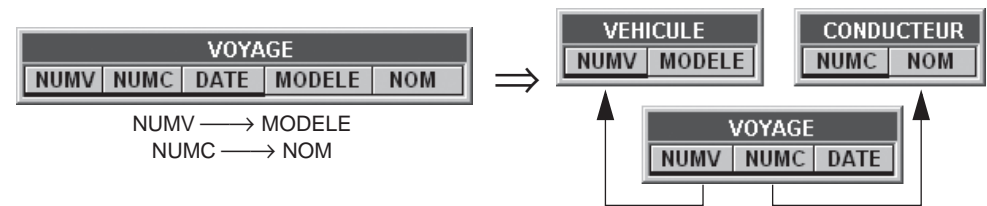
Les voyages

Dans le second exemple, la table VOYAGE décrit des véhicules (NUMV) d'un certain modèle (MODELE) effectuant des voyages à une certaine date (DATE), sous la responsabilité d'un conducteur (NUMC) dont on enregistre le nom (NOM). L'identifiant est constitué des colonnes ($\text{NUMV}, \text{NUMC}, \text{DATE}$).

Tout conducteur a un nom et tout véhicule est d'un certain modèle, caractéristiques qui ne dépendent pas des autres informations de la table. Ces deux propriétés se traduisent par les dépendances fonctionnelles suivantes :

$$\begin{aligned} \text{NUMC} &\longrightarrow \text{NOM} \\ \text{NUMV} &\longrightarrow \text{MODELE} \end{aligned}$$

La table VOYAGE est le siège de deux dépendances anormales, et n'est donc pas normalisée. On la traitera comme décrit ci-dessous.



La question de la dénomination des deux fragments est assez simple à résoudre. La table réduite conserve son nom. Elle représente en effet les mêmes faits que la table d'origine, débarrassée des informations problématiques (déterminé). Ces dernières sont regroupées, avec le déterminant, dans une seconde table, dont il faut préciser la signification. Celle-ci s'obtient en répondant à la question, relative à la figure 3.14 : *quels sont les objets ou les faits qui sont **identifiés** par B et **caractérisés** par C ?*

3.9.4 Remarques

L'étude des dépendances fonctionnelles, des formes normales et des techniques de normalisation constitue une partie essentielle du domaine des bases de données. Elle est plus riche et plus complexe que ce que nous avons discuté dans cette section. En particulier, il existe d'autres types de dépendances et d'autres formes normales. Celle que nous avons présentée, qui veut que *tout déterminant soit un identifiant*, est dénommée *forme normale de Boyce-Codd*, mais est souvent appelée (improprement) *3ème forme normale*. Le lecteur intéressé par cette question consultera, par exemple, [Bouzeghoub, 1998] ou [Date, 2001].

Les SGBD ignorent les dépendances fonctionnelles, sauf celles dont le déterminant est un identifiant de la table. Ils sont donc incapables de gérer les redondances internes d'une table non normalisée (sinon via des *déclencheurs*, étudiés au Chapitre 6). Il est donc important de n'admettre que des tables normalisées.

3.10 LES STRUCTURES PHYSIQUES

A la forme tabulaire des données correspond sur le support une structure physique relativement complexe qui garantit de bonnes performances lors de l'exécution des requêtes. On décrira brièvement certaines de ces structures : *index*, *espaces de stockage* et *agrégats physiques*. Il est important de noter que l'utilisateur qui consulte et modifie les données ignore la présence de ces structures physiques lors de la formulation d'une requête. Par exemple, toute requête est exécutable par le SGBD, qu'il existe ou non des index permettant d'accélérer l'accès et le tri des données.

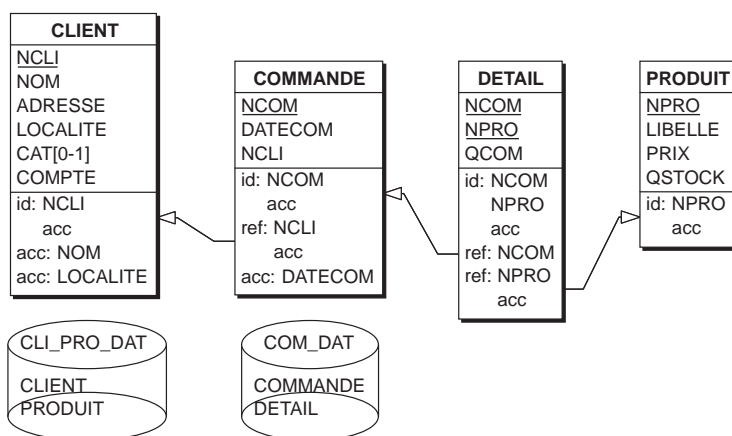


Figure 3.15 - Schéma physique d'une base de données spécifiant les index et les espaces de stockage

- Un **index** est une structure associée à une table, définie sur une ou plusieurs colonnes, permettant d'accéder rapidement, et de manière sélective, aux lignes qui possèdent des valeurs déterminées de ces colonnes; il permet également d'accéder aux lignes de la table dans l'ordre des valeurs (dé)croissantes de ces colonnes. Dans le schéma de la figure 3.15, un index est représenté par un groupe de colonnes préfixé du symbole **acc**, ou, si ce groupe est un identifiant ou une clé étrangère, par le suffixe **acc**. Ainsi, la table CLIENT est dotée de trois index : (NCLI), (NOM) et (LOCALITE).

Il existe plusieurs techniques de réalisation d'un index. L'une d'elles comporte une table de correspondance qui associe à chaque valeur de la colonne, ou des colonnes, la liste des numéros des lignes correspondantes. Dans cette table de correspondance, les valeurs des colonnes sont rangées par ordre croissant ou décroissant. Sur chaque table de données peuvent être déclarés un nombre quelconque d'index. On peut à tout instant ajouter ou retirer un index.

- Les lignes des tables sont rangées dans les **espaces de stockage** (*dbspace*, *tablespace*, *space* ou plus simplement *fichier*), qui constituent des *conteneurs* implantés dans une mémoire secondaire (disque magnétique, carte mémoire, CD-ROM, DVD). Un tel conteneur est découpé en *pages* de taille fixe (typiquement 4096

octets) et peut accueillir des lignes de plusieurs tables. Nous noterons graphiquement un espace de stockage par un cylindre comprenant la liste des noms des tables (figure 3.15).

- Un **agrégat physique** (souvent dénommé *cluster*) est formé d'un ensemble de lignes qui sont souvent extraites simultanément lors de l'exécution des requêtes. Cet agrégat est stocké de manière contiguë dans la mémoire secondaire (dans la même page par exemple) de manière à réduire le temps d'accès à l'ensemble de ces lignes. Par exemple, on constituera des agrégats formés chacun d'une ligne de COMMANDE et des lignes de DETAIL correspondantes. Ainsi, l'accès à une ligne de COMMANDE entraînera automatiquement le chargement en mémoire centrale de ses lignes de DETAIL.

3.11 LES SYSTÈMES DE GESTION DE DONNÉES

La gestion d'une base de données, sa consultation et, d'une manière générale, la manipulation des données qu'elle contient, constituent des opérations dont la réalisation technique est souvent très complexe, ainsi qu'on l'a montré dans les sections précédentes. C'est la raison pour laquelle on fera appel à des logiciels spécialisés appelés **systèmes de gestion de bases de données** (SGBD). Ces logiciels offrent un ensemble de fonctions permettant la définition, l'exploitation et la gestion de tables et de leur contenu.

Les SGBD s'adressent à la fois à l'utilisateur non spécialisé ou occasionnel et au développeur d'applications. Ils permettent de définir des tables et leur structure, de consulter des données extraites d'une ou plusieurs tables et sélectionnées selon des critères simples ou complexes, de modifier le contenu d'une table (ajouter, supprimer des lignes, modifier les valeurs de colonnes de lignes sélectionnées). En outre, ils disposent souvent d'un langage de programmation propre qui permet le développement rapide de programmes complexes. Ces logiciels exigent des ressources en matériel (mémoire centrale, vitesse du processeur, disques rapides et à grande capacité) importantes. Ils réclament aussi une formation spécialisée de la part de l'utilisateur qui désire en utiliser intensivement les fonctions. Une base de données complexe peut comporter plusieurs milliers de tables et plusieurs dizaines de milliers de colonnes.

Microsoft Access (MS Access) jouit d'une position un peu particulière. Il s'agit essentiellement d'un environnement de développement d'applications légères travaillant sur une base de données. Des outils graphiques permettent à un utilisateur de construire rapidement et intuitivement de petites applications. Il est possible d'utiliser le langage SQL, mais l'utilisateur doit alors faire preuve d'un acharnement digne d'un cochon truffier (nous en reparlerons en 4.1).

La suite de cette section présente succinctement les fonctions d'organisation et de gestion des données offertes par les SGBD relationnels.

a) Organisation des données

Les données d'une base de données SQL sont principalement organisées sous la forme de tables (*tables*), de colonnes (*columns*), d'identifiants primaires (*primary keys*) et de colonnes de référence (*foreign keys*). SQL permet également d'ajouter et de supprimer une table dans une base de données. Il permet d'ajouter, modifier et supprimer une colonne dans une table.

b) Gestion des données

Il est possible d'ajouter et de retirer des lignes dans une table. Il est possible de modifier les valeurs d'une colonne dans certaines lignes d'une table. Le SGBD garantit le respect des contraintes d'intégrité qui ont été déclarées.

c) Accès aux données

L'accès aux données et leur manipulation s'effectuent à l'aide du langage SQL. Celui-ci permet de décrire des ensembles de données en n'en définissant que les propriétés, c'est-à-dire sans faire référence aux techniques d'accès qui seront utilisées pour atteindre ces données dans la base. Cette désignation d'ensembles de données prend la forme de requêtes SQL. Le résultat d'une requête se présente sous la forme d'une table, éventuellement d'une seule ligne et/ou d'une seule colonne. Les données de cette table sont généralement extraites de la base de données. Cette table est alors soit un sous-ensemble (certaines colonnes, certaines lignes) d'une table existante, soit une table construite à partir d'extraits de plusieurs tables, mises en correspondance sur la base de valeurs communes dans certaines colonnes (jointure).

Les données de cette table peuvent également être déduites ou calculées : totaux, comptages, moyennes, etc. Le résultat d'une requête peut être ou non stocké dans la base de données.

d) Présentation des données extraites

Les données extraites seront généralement affichées à l'écran ou imprimées sur papier. Dans ces cas les données sont présentées sous forme d'un rapport. Ce rapport peut être simplement une copie des données telles qu'elles sont extraites. Ses lignes peuvent également être triées selon plusieurs colonnes. Elles peuvent être regroupées pour les mêmes valeurs d'une ou plusieurs colonnes. On peut demander d'éliminer les lignes en double. On peut enfin commander la disposition des données sur écran ou sur papier.

e) Les privilèges et le contrôle d'accès

L'accès aux données et leur manipulation sont soumis à autorisation. Chaque utilisateur a le droit de demander l'exécution de certaines opérations sur certains objets de la base de données. Ce droit est appelé un privilège. Le droit d'accorder un privilège est lui-même un privilège particulier; il permet aussi de retirer un privilège accordé.

f) Accès par programme

Les commandes SQL de définition, d'extraction et de modification de données peuvent être exécutées soit à partir d'un terminal, soit à partir d'un programme d'application.

g) Autres fonctions

Les SGBD SQL offrent d'autres fonctions indispensables dans un contexte industriel, telles que les suivantes :

- la *définition de vues*, qui permet à une classe d'utilisateurs de visualiser les données sous une forme personnalisée,
- la *protection contre les incidents*, qui garantit que les données ne seront pas corrompues à la suite d'incidents affectant le fonctionnement de la configuration informatique (serveurs, réseau, postes de travail, logiciels),
- la *gestion des accès concurrents*, qui permet à plusieurs utilisateurs de consulter et de modifier simultanément les mêmes données,
- le *dictionnaire de données*, qui contient une description précise des structures d'une base de données.

3.12 SQL ET LES BASES DE DONNÉES

Les SGBD (Systèmes de Gestion de Bases de Données) qui présentent les données sous la forme de tables proposent un langage de requête dénommé SQL (*Structured Query Language*). Présenté pour la première fois en 1973 par une équipe de chercheurs d'IBM [Chamberlin, 1974], ce langage a rapidement été adopté comme standard potentiel, et pris en charge par les organismes de normalisation ANSI et ISO¹⁵. La première norme a été publiée en 1986 (SQL-86). La suivante, SQL-89, introduit notamment l'intégrité référentielle et constitue la base de tous les SGBD actuels. La norme SQL-92, aussi dénommée SQL2, introduit de nombreuses extensions opérationnelle dans les SGBD d'aujourd'hui. La norme SQL:1999 (SQL3) est désormais disponible. Cependant, certains SGBD en ont déjà intégré certains composants. Alors que la norme 89 couvrait 110 pages seulement, la norme 92 compte 580 pages et la norme 1999 en exige plus de 3.000. Nous examinerons dans le chapitre 6 quelques extensions qu'apporte la norme 1999 par rapport à celle de 92, qui est celle qui est le plus utilisée actuellement.

Malheureusement, les éditeurs de SGBD montrent peu d'empressement à respecter les normes successives, bien qu'ils en aient été les principaux auteurs. Chacun des SGBD ne reprend qu'un sous-ensemble des spécifications, modifie la syntaxe, voire l'interprétation des concepts retenus, et ajoute ses propres fonctions.

15. Le lecteur intéressé par l'historique et l'évolution du langage SQL et de ses normes consultera par exemple [Melton, 2002]. Il pourra se procurer la définition des normes auprès des organismes nationaux de normalisation tels que l'AFNOR (11, avenue Francis de Pressensé, F-93571 Saint-Denis La Plaine Cedex, France, <http://www.afnor.fr>).

SQL est disponible sur tous les types de matériel, depuis les PDA jusqu'aux plus gros ordinateurs. On recensait en 1992 plus de 150 logiciels offrant le langage SQL [Khoshafian, 1992]. Aujourd'hui, l'offre s'est restructurée autour d'un petit nombre de grands SGBD, essentiellement DB2 d'IBM, ORACLE d'Oracle Corp., SQL-Server de Microsoft, mais aussi, dans une moindre mesure SYBASE de PowerSoft, INFORMIX d'Informix (actuellement IBM), INGRES de Computer Associates. Il existe cependant un plus grand nombre de systèmes moins répandus, mais en général plus faciles d'accès, tant sur le plan du coût que de la simplicité d'utilisation. L'environnement Access de Microsoft propose également une interface SQL. Elle est cependant plus délicate à utiliser, notamment par son caractère non standard, ses lacunes et la difficulté de trouver une documentation précise sur le langage.

Le monde Open Source propose également des SGBD relationnels de bonne qualité. On citera en particulier PostgreSQL (<http://www.postgresql.org>), Firebird (<http://firebird.sourceforge.net>), ou son grand frère InterBase 6 (<http://www.borland.com/interbase>), et MySQL (<http://www.mysql.com>). Ce dernier, malgré sa popularité auprès des développeurs d'applications internet, présente cependant des lacunes qui le rendent moins approprié pour l'enseignement.

3.13 EXERCICES

- 3.1 Vérifier si le schéma ci-dessous est normalisé. Si nécessaire, le décomposer en tables normalisées.

VENTE						
NPRO	CLIENT	DATE	QUANTITE	ADRESSE	DELEGUE	REGION

CLIENT —→ ADRESSE, DELEGUE
 DELEGUE —→ REGION
 CLIENT —→ ADRESSE, DELEGUE

- 3.2 Décomposer si nécessaire la table ci-dessous.

COMMANDE					
NCOM	NCLI	NOM	DATE	NPRO	LIBELLE

NCOM —→ NCLI
 NCLI —→ ADRESSE

- 3.3 Décomposer si nécessaire la table ci-dessous.

PRODUIT			
NPRO	DATE_INTRO	IMPORTATEUR	AGREATION

DATE_INTRO, IMPORTATEUR —→ AGREATION

Chapitre 4

Le langage SQL DDL

Ce chapitre ainsi que le suivant présentent, selon une approche progressive et raisonnée, le langage de base de données SQL, et en particulier les deux parties importantes que sont les sous-langages DDL (*Data Description Language*) et DML (*Data Manipulation Language*). On examine dans ce chapitre les fonctions de création des principales structures de données étudiées dans le chapitre 3, et la manière dont ces fonctions sont activées via le langage SQL DDL.

4.1 INTRODUCTION

Ce chapitre et le suivant sont consacrés à une introduction au langage SQL et, à travers lui, à une première approche des SGBD et de leurs fonctions. La démarche suivie est essentiellement pédagogique et se base sur une analyse des principales difficultés que rencontrent les étudiants débutants en cette matière. Cet exposé, bien que plus large que celui de [Hainaut, 1994], n'est pas complet, le lecteur étant renvoyé à la littérature spécifique, largement disponible dans toutes les librairies. Nous nous servons pour l'essentiel de la norme SQL2 du langage. A certaines occasions, nous citerons quelques extensions présentes soit dans la norme SQL3 (ou SQL:1999), soit dans certains SGBD. Le lecteur trouvera sur le site Web de l'ouvrage la syntaxe précise des principales instructions rencontrées dans ce chapitre et dans le suivant.

Tant la définition des structures de données (composants du schéma) que la manipulation du contenu (les données elles-mêmes), en passant par les autres fonctions, s'effectuent exclusivement au moyen du langage SQL. Une instruction SQL cons-

titue une **requête**, c'est-à-dire la description d'une opération que le SGBD doit exécuter. Une requête peut être introduite au terminal, auquel cas le résultat éventuel (dans le cas d'une consultation de données par exemple) de l'exécution de la requête apparaît à l'écran. Cette requête peut également être envoyée par un programme (écrit en Pascal, C, COBOL, Basic ou Java) au SGBD. Dans ce cas, le résultat de la requête est rangé par le SGBD, ligne par ligne, dans des variables du programme. Nous développerons plus particulièrement la formulation interactive des requêtes SQL, plus intuitive pour un lecteur novice en la matière.

Il est vivement recommandé de travailler les matériaux de cet ouvrage en les expérimentant au moyen d'un SGBD. Si lecteur n'a pas accès à un moteur *lourd* tel qu'Oracle ou DB2 (versions *Education* gratuites), on pourra recommander l'interface SQL de MS Access, InterBase, livré avec Delphi de Borland ou Firebird. Le site Web de l'ouvrage contient des exemples rédigés en InterBase 6 (Firebird) et en MS Access.

Remarque. Le langage SQL n'est pas immédiatement disponible dans MS-Access. On y accédera de la manière suivante, la fenêtre de la base de données étant ouverte : (1) sélectionner l'onglet Requetes; cliquer sur le bouton Nouveau; (3) sélectionner Mode Création; (4) fermer la fenêtre Ajouter une table; (5) dans la barre d'outils, sélectionner le mode d'affichage Mode SQL; (6) rédiger la requête SQL dans la fenêtre, l'exécuter (bouton "!") puis la sauver si nécessaire. Contrairement à InterBase et Firefox, MS-Access ne permet pas d'exécuter des scripts SQL (suite de requêtes). On trouvera sur le site Web de l'ouvrage un interpréteur de commandes SQL, développé en Visual Basic, qui permet d'exécuter de tels scripts.

4.2 LE LANGAGE SQL DDL

Le langage SQL offre des commandes de définition et de modification des structures. Ces commandes permettent de définir (créer) une table, de supprimer une table existante et d'ajouter une colonne à une table existante. L'ensemble de ces commandes forme un sous-langage de SQL dénommé SQL-DDL (*Data Definition Language*). Nous en examinerons ci-après les principales formes.

4.3 CRÉATION D'UN SCHÉMA

Une base de données est définie par son schéma. SQL propose donc de créer ce schéma avant de définir ses composants :

```
create schema CLICOM
```

Cette instruction sera accompagnée de divers paramètres spécifiant notamment les conditions d'autorisation d'accès. Les schémas sont rassemblés dans un catalogue qui représente donc un ensemble de bases de données. Un site peut contenir plusieurs catalogues.

Connexions et sessions

Nous ferons remarquer, sans insister sur les aspects techniques, que l'approche client/serveur propre aux SGBD SQL implique quelques démarches administratives préalables au travail sur une base de données. D'abord, l'utilisateur qui a l'intention de travailler sur la base de données (schéma ou données) doit établir une (ou plusieurs) connexion(s) avec le serveur sur lequel elle réside :

```
connect to ADMIN\SERV01 as CON_21Dec2005_002 user "j1h"
```

A la fin du travail, on demande la déconnexion :

```
disconnect CON_21Dec2005_002
```

Durant une connexion avec un serveur, l'utilisateur peut exécuter des instructions SQL de création de structures et de manipulation de données. Dans un contexte purement local, la procédure peut être simplifiée, comme dans InterBase :

```
connect "D:\-j1h-\DONNEES\clicom.gdb"  
user "j1h" password "j1hclicom";
```

4.4 CRÉATION D'UNE TABLE

Cette opération produit une table vide (c'est-à-dire sans lignes). On spécifie le nom de la table et la description de ses colonnes. On spécifiera pour chaque colonne son nom et le type de ses valeurs.

```
create table CLIENT ( NCLI      char(10),  
                     NOM       char(32),  
                     ADRESSE   char(60),  
                     LOCALITE  char(30),  
                     CAT       char(2),  
                     COMPTE    decimal(9,2))
```

a) Les colonnes et leurs types

SQL admet divers types de données dans la déclaration d'une colonne d'une table. On citera les principaux :

- SMALLINT : entiers signés courts (p. ex. 16 bits);
- INTEGER (ou INT) : entiers signés longs (p. ex. 32 bits);
- NUMERIC(p,q) : nombres décimaux de p chiffres dont q après le point décimal; si elle n'est pas mentionnée, la valeur de q est 0;
- DECIMAL(p,q) : nombres décimaux d'au moins p chiffres dont q après le point décimal; si elle n'est pas mentionnée, la valeur de q est 0;

- `FLOAT(p)` (ou `FLOAT`) : nombres en virgule flottante d'au moins `p` bits significatifs;
- `CHARACTER(p)` (ou `CHAR`) : chaîne de longueur fixe de `p` caractères;
- `CHARACTER VARYING` (ou `VARCHAR(p)`) : chaîne de longueur variable d'au plus `p` caractères;
- `BIT(p)` : chaînes de longueur fixe de `p` bits;
- `BIT VARYING` : chaînes de longueur variable d'au plus `p` bits;
- `DATE` : dates (année, mois et jour);
- `TIME` : instants (heure, minute, seconde, éventuellement 1000ème de seconde);
- `TIMESTAMP` : date + temps,
- `INTERVAL` : intervalle en années/mois/jours entre dates ou en heures/minutes/secondes entre instants¹.

Chaque SGBD ajoutera de sa propre initiative d'autres types de données. Citons-en deux généralement disponibles sous une forme ou sous une autre dans tous les moteurs SQL :

- les *Binary Large Objects* (BLOB), sorte de contenants génériques pouvant accueillir des chaînes de bits de longueur illimitée telles que des images, séquences vidéo, séquence sonores ou musicales. Les *Character Large Objects* (CLOB) sont similaires, mais considérés comme étant formés de caractères.
- les *générateurs de valeurs*, qui attribuent automatiquement la valeur suivante à leur colonne lors de l'insertion d'une ligne; ils permettent d'assigner des valeurs uniques à des colonnes servant d'identifiant technique.

Outre ces types de base abstraits et peu informatifs, il est possible de définir des domaines de valeurs qui rendront le schéma plus lisible et plus facile à modifier².

```
create domain MONTANT decimal(9,2)
create domain MATRICULE char(10)
create domain LIBELLE char(32)

create table CLIENT ( NCLI      MATRICULE,
                     NOM       LIBELLE,
                     ADRESSE   char(60),
                     LOCALITE  LIBELLE,
                     CAT        char(2),
                     COMPTE     MONTANT)
```

1. SQL distingue deux types d'intervalles car il n'est pas possible de calculer le nombre de jours compris dans `p` mois sans connaître ces mois.

2. Comme nous le verrons également, il est possible d'attacher des contraintes d'intégrité simples à un domaine.

On peut enfin définir la valeur, dite *valeur par défaut*, que le SGBD assignera automatiquement à une colonne (éventuellement via son domaine) lorsque l'utilisateur omettra d'en fournir une lors de la création d'une ligne :

```
create domain MONTANT decimal(9,2) default 0.0

CAT char(2) default 'AA'
```

En SQL:1999, les domaines tendent à être remplacés par le concept plus puissant de *type*.

b) Expression de l'identifiant primaire

On complétera la déclaration de la table par la clause *primary key* :

```
create table CLIENT ( NCLI      char(10),
                     NOM       char(32),
                     ADRESSE   char(60),
                     LOCALITE  char(30),
                     CAT       char(2),
                     COMPTE    decimal (9,2),
                     primary key (NCLI) )

create table DETAIL ( NCOM      char(12),
                     NPRO      char(15),
                     QCOM      decimal(8),
                     primary key (NCOM,NPRO) )
```

c) Expression d'un identifiant secondaire

Les autres identifiants seront déclarés par une clause *unique* :

```
create table ASSURE ( NUM_AFFIL char(10),
                     NUM_IDENT char(15),
                     NOM       char(35),
                     primary key (NUM_AFFIL),
                     unique (NUM_IDENT) )
```

d) Expression d'une contrainte référentielle

On déclarera la clé étrangère et la table référencée par une clause *foreign key*³ :

3. Certains éditeurs choisissent délibérément d'écarter cette contrainte avec des arguments tels que les suivants : *There are so many problems with foreign keys that we don't know where to start*, ou encore : *The only nice aspect of foreign key is that it gives ODBC and some other client programs the ability to see how a table is connected and to use this to show connection diagrams and to help in building applications*. MySQL Reference Manual, Version 3.23.2-alpha 8 August 1999. On avance aussi l'argument péremptoire qu' "il suffit de faire attention à ne pas introduire des données erronées". On pourrait, en suivant le même raisonnement, prôner l'inutilité des antibiotiques et des logiciels anti-virus. Les dernières versions de MySQL montrent que les concepteurs ont fait de louables efforts dans le bon sens.


```
create table DETAIL (  NCOM      char(12) not null,
                      NPRO      char(15) not null,
                      QCOM      decimal(8) not null,
                      primary key (NCOM,NPRO),
                      foreign key (NCOM) references COMMANDE,
                      foreign key (NPRO) references PRODUIT)
```

Une table contient au moins une colonne. Au moment de sa création, elle ne contient aucune ligne. Il est possible de créer de nouvelles tables à tout instant.

g) Forme synthétique des contraintes

Les contraintes impliquant une seule colonne peuvent être considérées comme des contraintes de colonnes (identifiants primaires et secondaires, clés étrangères). Elles pourront alors déclarées comme complément de la définition de cette colonne :

```
create table COMMANDE (
    NCOM char(12) not null primary key,
    NCLI char(10) not null references CLIENT,
    DATECOM date not null)
```

4.5 SUPPRESSION D'UNE TABLE

Toute table peut être supprimée. Elle est désormais inconnue et son contenu est perdu.

```
drop table DETAIL
```

Si, au moment de la suppression, la table contenait des lignes, celles-ci sont préalablement supprimées, opération qui est soumise aux contraintes référentielles qui concernent la table. Par exemple, la suppression de la table COMMANDE pourrait entraîner la suppression de toutes les lignes de DETAIL (mais pas de la table). On se reportera à la discussion de la section 3.7.2 sur l'impact des contraintes d'intégrité sur les opérations de modification des données.

4.6 AJOUT, RETRAIT ET MODIFICATION D'UNE COLONNE

La commande suivante ajoute la colonne POIDS à la table PRODUIT :

```
alter table PRODUIT
add column POIDS smallint
```

Après l'exécution de cette commande, la table possède une nouvelle colonne qui ne contient que des valeurs *null* pour toutes les lignes. On ne peut ajouter une colonne obligatoire que si la table est vide, ou si cette colonne possède une valeur par défaut.

L'élimination d'une colonne d'une table s'effectue à l'aide d'une commande similaire :

```
alter table PRODUIT
drop column PRIX
```

Il est également possible de modifier ou de supprimer un domaine (on modifie ici la valeur par défaut) :

```
alter table CLIENT
alter column CAT set '00'
```

Il est également possible de modifier ou supprimer un domaine :

```
drop domain MATRICULE
```

4.7 AJOUT ET RETRAIT D'UNE CONTRAINTE

Nous avons rencontré jusqu'ici quatre types de contraintes : les identifiants primaires, les identifiants secondaires, les colonnes obligatoires et les clés étrangères. Ces contraintes, ou propriétés, sont généralement déclarées lors de la création de la table qui y est soumise. Il est cependant possible de les ajouter et même de les retirer a posteriori par une commande `alter table`.

La demande d'ajout d'un identifiant sera refusée si les données que la table contient déjà violent cette propriété :

```
alter table CLIENT
add primary key (NCLI)
```

Il en va de même pour les identifiants secondaires déclarés via une clause unique :

```
alter table CLIENT
add unique (NOM,ADRESSE,LOCALITE)
```

Une colonne obligatoire peut être redéclarée facultative et inversement (si les données le permettent) :

```
alter table CLIENT
modify CAT not null
```

```
alter table CLIENT
modify ADRESSE null
```

Enfin, une clé étrangère peut être définie (si les données le permettent) ou retirée après création de la table source. Cette possibilité est indispensable lorsque le SGBD n'accepte pas la déclaration d'une clé étrangère vers une table non encore définie (ce qu'on appelle une *référence en avant*).

```
alter table COMMANDE
add foreign key (NCLI) references CLIENT
```

Lors de la déclaration d'une contrainte, que ce soit dans la définition ou dans la modification de la table concernée, il est possible de donner un nom à cette contrainte par la clause préliminaire `constraint <nom>` :

```
create table DETAIL (NCOM char(12) not null,
                    NPRO char(15) constraint C1 not null,
                    QCOM decimal(8),
                    constraint C2 primary key (NCOM,NPRO),
                    constraint C3 foreign key (NPRO) references PRODUIT)
```

```
alter table CLIENT
add constraint C_CLI_U unique (NOM,ADRESSE,LOCALITE)

alter table COMMANDE
add constraint C5 foreign key (NCLI) references CLIENT
```

Il est possible de supprimer une contrainte existante :

```
alter table DETAIL
drop constraint C2
```

On notera que le langage SQL DDL de MS Access exige que les identifiants primaires et les clés étrangères soient déclarés sous forme de contraintes nommées.

4.8 LES STRUCTURES PHYSIQUES

Les structures physiques décrites dans la section 3.10 peuvent être définies par des requêtes spécifiques. Nous décrirons brièvement le traitement des index et des espaces de stockage.

Un **index** est créé par la commande `create index`. On spécifie son nom, la table à laquelle il est associé, les colonnes qui le composent, et l'ordre de rangement des valeurs dans l'index (`asc` = ascendant, `desc` = descendant). L'ordre par défaut est *ascendant*.

```
create index XCLILOC
on CLIENT (LOCALITE)
```

Si les colonnes de l'index forment également un identifiant, on définira un index unique :

```
create unique index XCLI_NCLI
on CLIENT (NCLI desc)

create unique index XDET1
```

```
on DETAIL (NCOM asc, NPRO desc)
```

Il est en outre possible de supprimer un index à tout instant par la commande,

```
drop index XDET1
```

Un **espace de stockage** est créé comme tout objet SQL :

```
create dbspace CLI_PRO_DAT
```

On spécifiera ensuite pour chaque table dans quel espace ses lignes doivent être stockées :

```
create table CLIENT ( ... ) in CLI_PRO_DAT
```


Chapitre 5

Le langage SQL DML

Ce chapitre décrit la deuxième partie importante du langage SQL : le DML (*Data Manipulation Language*). On y examine les principes de l'extraction de données d'une ou plusieurs tables ainsi que la modification des données.

5.1 INTRODUCTION

SQL DML est certainement la partie la plus spectaculaire du langage SQL. Il comporte deux grandes classes de fonctions : l'extraction de données et la modification de données.

L'extraction fait l'objet d'une seule commande : la requête `select`, qui nous permet d'extraire d'une base de données les données répondant à des questions que nous pourrions nous poser. Sa forme est cependant d'une telle puissance et d'une telle richesse que son étude exigera la plus grande partie de ce chapitre. Bien que la connaissance du langage SQL soit l'un des objectifs de celui-ci, nous voudrions surtout insister sur l'expression rigoureuse de raisonnements corrects, ce que le langage courant est souvent loin de permettre.

La modification du contenu d'une base de données est simple en apparence : ajouter (`insert`), supprimer(`delete`) et modifier (`update`) les lignes d'une table. Nous verrons cependant que la prise en compte des contraintes d'intégrité nous amènera à considérer ces opérations avec circonspection.

Dans la suite de cet exposé, nous aborderons successivement :

1. l'extraction de données d'une seule table, indépendamment des autres tables (section 5.2)

2. l'extraction de données d'une seule table, mais en corrélation avec les données d'une autre table - les sous-requêtes (section 5.3)
3. l'extraction de données de plusieurs tables - les jointures (section 5.4)
4. l'extraction de données groupées (section 5.5)
5. la modification des données (section 5.8)

5.2 CONSULTATION ET EXTRACTION DE DONNÉES DANS UNE TABLE

Les nombreuses variantes de l'instruction `select` d'SQL permettent d'extraire des données d'un ensemble de tables et de les présenter, toujours sous la forme d'une table, soit à l'utilisateur au terminal, soit au programme d'application qui en a demandé l'exécution. La présentation qui en sera faite suppose une utilisation interactive : l'utilisateur introduit la requête au terminal, en demande l'exécution au SGBD qui affiche les données extraites à l'écran.

Sauf mention contraire, tous les exemples de requêtes et leurs résultats sont relatifs à la base de données illustrée à la figure 3.6.

5.2.1 Principes

L'exécution d'une requête `select` produit un résultat qui est une table. Ce fait est important, car il implique naturellement qu'on devrait pouvoir écrire une requête `select` partout où on utilise le nom d'une table. Ceci n'est vrai, avec quelques restrictions, qu'en SQL:1999 mais on trouve déjà dans SQL2, quelques possibilités dans ce sens.

D'une manière générale, une requête simple contient trois parties principales :

- la clause **select** précise les valeurs (nom des colonnes, valeurs dérivées) qui constituent chaque ligne du résultat,
- la clause **from** indique les tables desquelles le résultat tire ses valeurs,
- la clause **where** donne la condition de sélection que doivent satisfaire les lignes qui fournissent le résultat.

Le résultat d'une requête `select`, ou plus généralement SFW (select-from-where) est une table fictive qui, dans cet exposé, sera considérée comme s'affichant à l'écran. Dans d'autres circonstances que nous rencontrerons plus loin (5.8.1), ce résultat pourra être matérialisé sous la forme d'une table permanente.

Les expressions SFW seront introduites progressivement, par ordre de complexité croissante.

5.2.2 Extraction simple

La requête la plus simple consiste à demander d'afficher les valeurs de certaines colonnes de lignes d'une table. La requête suivante demande les valeurs de `NCLI`, `NOM` et `LOCALITE` des lignes de la table `CLIENT`.

```
select NCLI, NOM, LOCALITE
from CLIENT
```

La réponse à cette requête se présenterait comme suit à l'écran.

NCLI	NOM	LOCALITE
B062	GOFFIN	Namur
B112	HANSENNE	Poitiers
B332	MONTI	Genève
B512	GILLET	Toulouse
C003	AVRON	Toulouse
C123	MERCIER	Namur
C400	FERARD	Poitiers
D063	MERCIER	Toulouse
F010	TOUSSAINT	Poitiers
F011	PONCELET	Toulouse
F400	JACOB	Bruxelles
K111	VANBIST	Lille
K729	NEUMAN	Toulouse
L422	FRANCK	Namur
S127	VANDERKA	Namur
S712	GUILLAUME	Paris

Si on demande les valeurs de toutes les colonnes, la clause `select` peut se simplifier comme suit¹ :

```
select *
from CLIENT
```

5.2.3 Extraction de lignes sélectionnées

Extrayons à présent les informations NCLI et NOM des lignes de la table CLIENT qui concernent les clients de Toulouse.

```
select NCLI, NOM
from CLIENT
where LOCALITE = 'Toulouse'
```

Le résultat est la table suivante :

1. Cette forme synthétique est déconseillée pour une requête encapsulée dans un programme d'application (ou une procédure SQL, un déclencheur ou dans la définition d'une vue (chapitre 6)). En effet, elle produit une table dont le nombre de colonnes est celui de la table source au moment de l'exécution. Si ce schéma évolue, il est nécessaire de modifier les programmes.

NCLI	NOM
B512	GILLET
C003	AVRON
D063	MERCIER
F011	PONCELET
K729	NEUMAN

La relation d'égalité apparaissant dans la condition de sélection n'est qu'un exemple de comparateur; on dispose en fait des relations suivantes :

égal à :	=
plus grand que :	>
plus petit que :	<
différent de :	<>
plus grand ou égal :	>=
plus petit ou égal :	<=

L'interprétation de ces relations est évidente pour les valeurs numériques et temporelles. En ce qui concerne les chaînes de caractères, $A < B$, pour toutes chaînes A et B , s'interprète comme A est avant B selon l'ordre lexicographique (celui du dictionnaire). Attention, selon cet ordre, "a" n'est pas égal à "A". Plus précisément, "Z" < "a". D'autres relations seront discutées plus tard. Notons encore la formulation des principales constantes :

- numériques et décimales : suite de caractères décimaux, éventuellement signée et pouvant inclure un point décimal; exemples : 123, -0.003, 7.12;
- chaînes de caractères : valeur entre ' et ' (exemple : 'Mercier Jean'); la présence du caractère ' dans la chaîne se représente par son redoublement (exemple : 'rue de l''Eté');
- dates : '2005-02-14' (standard SQL2²); autres variantes selon les SGBD : '14-02-2005', '14-FEB-2005', '14/02/2005'.

5.2.4 Lignes dupliquées dans le résultat

En principe, le résultat d'une requête monotable contient autant de lignes qu'il y a, dans la table de départ, de lignes vérifiant la condition de sélection. Il se peut donc, dès qu'aucun identifiant n'est repris entièrement dans la clause `select`, que le résultat contienne plusieurs lignes identiques.

La requête suivante affiche les localités où habitent des clients de catégorie C1 :

```
select LOCALITE
```

2. En fait, le standard préconise la forme `date '2005-02-14'`; pour des raisons de concision, nous adopterons la forme courte. En MS Access, une date se note `#2005-02-14#`.

```
from CLIENT
where CAT = 'C1'
```

Nous obtenons le résultat suivant :

<u>LOCALITE</u>
Poitiers
Namur
Poitiers
Namur
Namur

La réponse contient autant de lignes que la table originale CLIENT en contient qui satisfont la sélection. On pourra éliminer les lignes en double par la clause `distinct` comme suit³ :

```
select distinct LOCALITE
from CLIENT
where CAT = 'C1'
```

Nous obtenons le résultat suivant, qui ne contient plus que des lignes distinctes :

<u>LOCALITE</u>
Namur
Poitiers

Ainsi qu'on l'a indiqué dans le chapitre 3, nous travaillerons sur des tables initiales dotées chacune d'un identifiant primaire, c'est-à-dire dont les lignes sont distinctes. On vient de voir cependant que le résultat d'une requête peut contenir des lignes dupliquées. Il faut être très prudent lorsqu'on exploite de tels résultats. Par exemple, la requête suivante donne les numéros des clients qui ont passé au moins une commande.

```
select NCLI
from COMMANDE
```

Cependant, le nombre d'éléments du résultat n'est pas égal à celui des clients qui ont passé une commande, mais bien au nombre de commandes. On écrira plutôt :

```
select distinct NCLI
from COMMANDE
```

3. En fait, l'expression "select LOCALITE ..." est une forme abrégée de "select **all** LOCALITE ...".

L'existence de lignes en double dans une table pose de nombreux problèmes logiques dont le lecteur trouvera le développement dans [Codd, 1990] et [Date, 1992] par exemple.

Unicité des lignes par construction

Si la clause `select` cite *tous les composants de l'un des identifiants* de la table, l'unicité des lignes du résultat est garantie. Il est donc inutile⁴ d'inclure `distinct`.

5.2.5 Des conditions de sélection plus complexes

a) Autres conditions élémentaires

Une condition élémentaire peut porter sur la présence de la valeur `null` :

```
CAT is null
CAT is not null
```

On pourra s'étonner qu'il faille une forme spéciale (`is`, `is not`) pour exprimer l'égalité ou la non égalité dans le cas des valeurs `null`. Pourquoi en effet n'a-t-on pas utilisé la forme classique : `CAT = null` et `CAT <> null` ? Nous justifierons ce choix lorsque nous examinerons plus en détail la question de l'information incomplète (Section 6.10). Nous y verrons que ces deux conditions, quelle que soit la valeur de `CAT`, ne pourront jamais avoir la valeur vrai.

Une condition peut aussi porter sur l'appartenance à une liste :

```
CAT in ('C1', 'C2', 'C3')
LOCALITE not in ('Toulouse', 'Namur', 'Breda')
```

ou à un intervalle :

```
COMPTE between 1000 and 4000
CAT not between 'B2' and 'C1'
```

ou encore sur la présence de certains caractères dans une valeur :

```
CAT like '_1'
ADRESSE like '%Neuve%'
```

Ces dernières conditions utilisent un **masque** qui décrit la structure générale des valeurs désignées. Dans ce masque, le signe `"_"` désigne un caractère quelconque et `"%"` désigne toute suite de caractères, éventuellement vide; tout autre caractère doit être présent là où il apparaît. La première expression est satisfaite dans la table `CLIENT` pour les valeurs de `CAT` constituées d'un caractère quelconque suivi du caractère `"1"`, soit l'une des valeurs `"A1"`, `"B1"`, `"C1"`. La seconde est satisfaite par

4. et d'ailleurs déconseillé du point de vue des performances.

toute valeur de ADRESSE contenant le mot "Neuve". Un petit problème se pose : comment rechercher les caractères % et _ dans les données ? Il suffit de les préfixer dans le masque par un caractère spécial qu'on définit dans une clause `escape`. Dans l'exemple ci-dessous, on recherche la présence de la chaîne '_CHENE' dans les valeurs de LIBELLE:

```
LIBELLE like '%$_CHENE%' escape '$'
```

Un masque peut aussi s'appliquer à une date. La forme suivante retient les dates de commande tombant en 2005

```
DATECOM like '%2005%'
```

Cette condition admet une forme négative, dont l'interprétation est évidente :

```
ADRESSE not like '%Neuve%'
```

Note

L'effet d'une condition `like` portant sur le dernier caractère d'une valeur dépend du type `char` ou `varchar` de cette valeur. Une colonne de type `char(n)` contient des valeurs composées de `n` caractères exactement. Si la valeur qu'on y range comporte moins de `n` caractères, ceux-ci sont considérés comme étant complétés par des espaces. Donc, la condition `CAT like '_1'` reconnaîtra 'B1 si CAT est déclarée `char(2)` mais pas si on l'avait déclarée `char(3)`. Le type `varchar(n)` définit des valeurs dont la longueur est le nombre de caractères effectivement introduits (à concurrence de `n`). Cette condition se comporterait donc selon l'intuition si CAT avait été déclarée `varchar(3)`.

b) Expressions composées

La condition de sélection introduite par la clause `where` peut être constituée d'une expression booléenne de conditions élémentaires (opérateurs `and`, `or`, `not` et parenthèses). La requête suivante retient les lignes pour lesquelles, simultanément (`and`), LOCALITE a pour valeur 'Toulouse' et COMPTE a une valeur négative.

```
select NOM, ADRESSE, COMPTE
from   CLIENT
where  LOCALITE = 'Toulouse' and COMPTE < 0
```

Etant donné les conditions P et Q relatives aux lignes de la table T, la clause

- `where P and Q` sélectionne les lignes de T qui vérifient simultanément P et Q;
- `where P or Q` sélectionne les lignes de T qui vérifient P ou Q ou les deux;
- `where not P` sélectionne les lignes de T qui ne vérifient pas P.

L'usage de parenthèses permet de former des conditions plus élaborées encore :

```
where COMPTE > 0
and    (CAT = 'C1' or LOCALITE = 'Paris')
```

Les parenthèses peuvent être utilisées même lorsqu'elles ne sont pas indispensables, par exemple pour rendre plus lisible une condition composée :

```
where (LOCALITE = 'Toulouse') and (COMPTE < 0)
```

5.2.6 Un peu de logique

Il n'est peut-être pas inutile de rappeler quelques principes et propriétés de la logique classique. Celle-ci va en effet nous servir à comprendre, exprimer et simplifier les conditions de sélection qui apparaissent dans la clause `where`⁵.

Sur le plan pratique, outre la *négation* (communément notée \neg et dénommée `not` en SQL), deux opérateurs logiques à deux arguments (les deux conditions `P` et `Q` par exemple) nous permettront de construire des conditions composées à partir de conditions plus simples :

- la *conjonction*, notée $P \wedge Q$ (`P` et `Q` en français, `P and Q` en SQL); cette expression est vraie si les arguments sont vrai;
- la *disjonction*, notée $P \vee Q$ (`P` ou `Q` en français, `P or Q` en SQL); cette expression est vraie si au moins un des arguments est vrai.

Lors de l'évaluation d'une expression complexe, la négation (`not`) a priorité sur la conjonction (`and`), qui elle-même a priorité sur la disjonction (`or`). On peut imposer un ordre d'évaluation différent à l'aide de parenthèses. Soit, par exemple, l'expression :

```
LOCALITE = 'Toulouse' and COMPTE < 0 or CAT = 'C1'
```

Elle indique qu'une ligne de `CLIENT` est sélectionnée si (`LOCALITE = 'Toulouse'` et `COMPTE < 0`) ou si (`CAT = 'C1'`) ou si les deux conditions sont vérifiées. On aurait pu écrire, de manière équivalente :

```
(LOCALITE = 'Toulouse' and COMPTE < 0) or CAT = 'C1'
```

En revanche, l'expression :

```
LOCALITE = 'Toulouse' and (COMPTE < 0 or CAT = 'C1')
```

a une toute autre interprétation. Une ligne de `CLIENT` est sélectionnée si elle vérifie simultanément deux conditions : d'une part, (`LOCALITE = 'Toulouse'`), et d'autre

5. On trouvera aussi de telles conditions dans les clauses *check*, dans les *triggers* et dans les *procédures SQL*, comme nous le verrons dans le chapitre 5.

part, ($\text{COMPTE} < 0$, ou bien $\text{CAT} = \text{'C1'}$, ou encore les deux conditions simultanément). Il sera prudent, même lorsqu'elles ne sont pas strictement nécessaires, d'utiliser de parenthèses dans l'écriture d'expressions complexes afin d'éviter toute ambiguïté d'interprétation.

A l'occasion, trois autres opérateurs pourront être utiles :

- la *disjonction exclusive*, notée $P \oplus Q$ (ou *exclusif* en français, absent d'SQL); cette expression est vraie si un et un seul des arguments est vrai;
- l'*implication*, notée $P \Rightarrow Q$ (implique en français, absent d'SQL); cette expression est vraie sauf si P est vrai et Q est faux;
- l'*équivalence*, notée $P \Leftrightarrow Q$ (équivalent à en français, absent d'SQL); cette expression est vraie lorsque P et Q ont la même valeur.

Un opérateur logique est complètement défini par sa *table de vérité*, qui indique la valeur (vrai ou faux) de l'expression pour chaque combinaison de valeurs des arguments. On rappelle les tables des opérateurs mentionnés ci-dessus.

P	Q	$\neg P$	$\neg Q$	$P \wedge Q$	$P \vee Q$	$P \oplus Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
vrai	vrai	faux	faux	vrai	vrai	faux	vrai	vrai
vrai	faux	faux	vrai	faux	vrai	vrai	faux	faux
faux	vrai	vrai	faux	faux	vrai	vrai	vrai	faux
faux	faux	vrai	vrai	faux	faux	faux	vrai	vrai

Son interprétation est la suivante : pour toute combinaison de valeurs de P et Q, la ligne correspondante donne la valeur de vérité de chaque opérateur. Par exemple, si $P = \text{vrai}$ et $Q = \text{faux}$, alors $P \vee Q = \text{vrai}$ et $P \Rightarrow Q = \text{faux}$.

Soient P, Q et R trois conditions élémentaires ou composées. On a les équivalences suivantes, qu'on démontre aisément en constatant que les tables de vérité des deux membres sont identiques.

Un Ricard, sinon rien (principe du tiers exclu)

Il n'y a pas d'autres valeurs que vrai et faux : ce qui n'est pas vrai est faux et inversement.

- $\neg \text{vrai} \equiv \text{faux}$ r1
- $\neg \text{faux} \equiv \text{vrai}$ r2

Le beurre et l'argent du beurre (complémentarité)⁶

- $(\neg P) \wedge P \equiv \text{faux}$ r3
- $(\neg P) \vee P \equiv \text{vrai}$ r4

6. Pour le sourire de la crémère, il faudra recourir à la logique ternaire, qui sera examinée à la Section 6.10.

Vous n'êtes pas sans "ignorer" (double négation)

- $\neg (\neg P) \equiv P$ r5

Schtroumpf vert et vert Schtroumpf (commutativité)

- $P \vee Q \equiv Q \vee P$ r6
- $P \wedge Q \equiv Q \wedge P$ r7

Jules et Jim (associativité)

- $(P \vee Q) \vee R \equiv P \vee (Q \vee R)$ r8
- $(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$ r9

Les Restos du coeur (distributivité)

- $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$ r10
- $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$ r11

Purification ethnique (lois de de Morgan)

- $\neg (P \wedge Q) \equiv (\neg P) \vee (\neg Q)$ r12
- $\neg (P \vee Q) \equiv (\neg P) \wedge (\neg Q)$ r13

Ariel ou eau de Javel ? (éléments neutres et absorbants)

- $P \wedge \text{vrai} \equiv P$ r14
- $P \wedge \text{faux} \equiv \text{faux}$ r15
- $P \vee \text{vrai} \equiv \text{vrai}$ r16
- $P \vee \text{faux} \equiv P$ r17

Les Faussaires

L'implication (\Rightarrow), la disjonction exclusive (\oplus) et l'équivalence (\Leftrightarrow) n'existent pas en SQL, et doivent donc être remplacées par des expressions équivalentes.

- $P \Rightarrow Q \equiv (\neg P) \vee Q$ r18
- $P \oplus Q \equiv (P \wedge \neg Q) \vee (\neg P \wedge Q)$ r19
- $P \oplus Q \equiv (P \vee Q) \wedge \neg (P \wedge Q)$ r20
- $P \Leftrightarrow Q \equiv (P \wedge Q) \vee \neg (P \vee Q)$ r21

Ces tables et ces règles définissent la logique *binaire* (à deux valeurs, vrai et faux) ou *cartésienne* ou encore *du tiers exclu*. Nous verrons plus tard que certains aspects du langage SQL s'appuient également sur une logique *ternaire*, utilisant trois valeurs : vrai, faux et inconnu (section 6.10). Les règles ci-dessus ne sont plus d'application dans cette logique.

Appliquons ces règles à quelques exemples concrets, qu'on exprimera selon la syntaxe SQL.

► La **négation** d'une condition complexe

Considérons d'abord les *clients de Toulouse dont le compte est négatif*. Les renseignements les concernant peuvent être obtenus par la condition composée :

```
LOCALITE = 'Toulouse' and COMPTE < 0
```

Les clients qui ne tombent pas dans cette catégorie sont caractérisés par la condition inverse, soit :

```
not (LOCALITE = 'Toulouse' and COMPTE < 0)
```

ou encore, par les lois de de Morgan (r12) :

```
not (LOCALITE = 'Toulouse') or not (COMPTE < 0)
```

ou, en simplifiant :

```
LOCALITE <> 'Toulouse' or COMPTE >= 0
```

► L'opérateur d'implication

L'opérateur d'implication ($P \Rightarrow Q$) est plus délicat à manier, en particulier parce que sa définition, telle que précisée dans la table de vérité ci-dessus, ne semble pas toujours conforme à l'intuition⁷, qui interprète l'implication de manière plus restrictive. Comme il n'est pas disponible en SQL, nous devons le traduire à l'aide d'opérateurs plus classiques. A titre d'illustration, recherchons les clients qui, *s'ils ont un compte négatif, alors sont aussi de catégorie B1*. On peut écrire, en SQL étendu :

```
(COMPTE < 0)  $\Rightarrow$  (CAT = 'B1')
```

L'interprétation est la suivante : si un client a un compte négatif et est de catégorie B1, alors il est sélectionné; bien que ceci soit moins intuitif, on admet aussi que, si son compte n'est pas négatif, alors il est sélectionné quelle que soit sa catégorie. Ou encore : les clients sont sélectionnés s'ils ont un compte non négatif ou s'ils sont de catégorie B1 (règle r18). On peut donc réécrire la condition de sélection de ces clients en SQL pur:

```
(COMPTE >= 0) or (CAT = 'B1')
```

► L'opérateur **ou exclusif**

Cet opérateur stipule que l'une des conditions doit être vérifiée, et une seulement, ainsi que l'exprime la règle r19 :

```
 $P \oplus Q \equiv (Q \text{ and not } P) \text{ or } (P \text{ and not } Q)$ 
```

7. Considérons la loi selon laquelle "*s'il pleut, alors la chaussée est mouillée*" ($P \equiv$ "*il pleut*" et $Q \equiv$ "*la chaussée est mouillée*"). On peut affirmer sans risque que "*il pleut et la chaussée n'est pas mouillée*" décrit une situation qui viole cette loi, d'où, en toute généralité, $(P \wedge \neg Q) = \text{faux}$. On en infère que $\neg (P \wedge \neg Q) = \text{vrai}$. Il vient donc, par application de la règle r12, $P \Rightarrow Q \equiv (\neg P) \vee Q$. D'où la règle r18.

Recherchons par exemple les clients qui n'ont pas de catégorie ou dont le compte est négatif (mais pas les deux) peut s'écrire :

```
where ((CAT is null) and (COMPTE >= 0))
or      ((CAT is not null) and (COMPTE < 0))
```

ou encore, selon la règle r20 :

```
where ((CAT is null) or (COMPTE < 0))
and    not ((CAT is null) and (COMPTE < 0))
```

► L'opérateur d'équivalence

L'équivalence ($P \Leftrightarrow Q$) correspond à une double implication : $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$. Cet opérateur renvoie vrai lorsque P et Q ont même valeur, soit, selon la règle r21 :

$$P \Leftrightarrow Q \equiv (Q \text{ and } P) \text{ or not } (P \text{ or } Q)$$

Recherchons les clients qui, *si leur compte est négatif, ont une catégorie du type 'Bx'*, et inversement, *s'ils ont une telle catégorie, alors leur compte est négatif*. Ceux qui vérifient les deux conditions, ou qui n'en vérifient aucune, sont sélectionnés. En revanche, ceux qui ne vérifient qu'une seule de ces conditions sont écartés. On écrira :

```
where ((COMPTE < 0) and (CAT like 'B_'))
or      not ((COMPTE < 0) or (CAT like 'B_'))
```

ou encore, selon la règle r13 :

```
where ((COMPTE < 0) and (CAT like 'B_'))
or      ((COMPTE >= 0) and (CAT not like 'B_'))
```

5.2.7 Données extraites et données dérivées

Les données extraites jusqu'à présent proviennent directement de la base de données. La clause `select` peut cependant spécifier des données dérivées, ou même des constantes :

```
select 'TVA de ', NPRO, ' = ', 0.21*PRIX*QSTOCK
from   PRODUIT
where  QSTOCK > 500
```

Cette requête produit un tableau des montants TVA des articles en stock dont la quantité restante est supérieure à 500 unités.

TVA de	NPRO	=	0,21*PRIX*QSTOCK
TVA de	CS264	=	67788
TVA de	PA45	=	12789
TVA de	PH222	=	37770,6
TVA de	PS222	=	47397

Lors de l’affichage du résultat, les colonnes reçoivent un nom qui est celui du composant de la clause `select`. Dans le cas de données dérivées, le nom peut apparaître encombrant et peu significatif. On pourra alors définir explicitement le nom qui apparaît en début de colonne, ce qu’on appelle un **alias de colonne**. La formulation ci-dessous illustre ce principe :

```
select NPRO as Produit, 0.21*PRIX*QSTOCK as Valeur_TVA
from   PRODUIT
where  QSTOCK > 500
```

Produit	Valeur_TVA
CS264	67788
PA45	12789
PH222	37770,6
PS222	47397

5.2.8 Les fonctions SQL

SQL offre une panoplie de fonctions permettant de dériver des valeurs à partir, entre autres, des valeurs des colonnes des lignes extraites. Des expressions faisant usage de ces fonctions peuvent apparaître dans la clause `select` et dans la clause `where`⁸, mais aussi, comme nous le verrons plus tard, dans les instructions `update` et `insert`. Nous citerons les principales⁹.

a) Fonctions numériques

- Il s’agit des quatre opérateurs arithmétiques classiques : +, -, * et /, auxquels on ajoutera le changement de signe -. La pauvreté de cette liste est souvent compensée par l’ajout de quelques fonctions mathématiques (exponentielle, logarithme, trigonométriques) dans certains SGBD. C’est ainsi qu’on trouvera souvent l’extraction de la partie entière d’un nombre ou sa valeur absolue¹⁰.

8. Ainsi que, indirectement, dans la clause `from` lorsqu’elle implique des tables dérivées.

9. Rappelons que les SGBD prennent souvent des libertés avec les standards. Ces fonctions ne seront sans doute pas toutes disponibles dans un SGBD particulier ou le seront sous une autre forme.

10. Respectivement `int` et `abs` en Access.

b) Fonctions de chaînes de caractères

Ces fonctions renvoient soit une chaîne, soit un nombre entier.

- `char_length(ch)` : donne le nombre de caractères de la chaîne *ch*;
- `position(ch1 in ch2)` : donne la position de chaîne *ch1* dans la chaîne *ch2*; 1 si *ch1* est vide et 0 si *ch1* n'apparaît pas dans *ch2*; ¹¹
- `ch1 || ch2` : construit une chaîne composée de la concaténation (*mise bout-à-bout*) des chaînes *ch1* et *ch2*; ¹²
- `lower(ch)` : construit une chaîne formée des caractères de *ch* transformés en minuscules;
- `upper(ch)` : construit une chaîne formée des caractères de *ch* transformés en majuscule;
- `substring(ch from I for L)` : construit une chaîne formée des *L* caractères de la chaîne *ch* partant de la position *I*; ¹³.
- `trim(e c from ch)` : supprime les caractères *c* à l'extrémité *e* de la chaîne *ch*; les valeurs de *e* sont *leading*, *trailing* et *both*. Le format simplifié `trim(ch)` correspond à `trim(both ' ' from ch)`.

Exemples

```
trim(both ' ' from ADRESSE) || ' ' || upper(LOCALITE)
position('NEUVE' in upper(ADRESSE)) > 0
upper(ADRESSE) like '% ' || upper(LOCALITE) || '%'
```

c) Fonctions de chaînes de bits

- `bit_length(ch)` : donne le nombre de bits dans la chaîne *ch*;
- `octet_length(ch)` : donne le nombre d'octets de la chaînes de bits *ch*.

d) Fonctions de conversion

- `cast(v as t)` : convertit la valeur *v* selon le type *t*.

Exemples

```
cast(TECOM as char(12))
CLI.COMPTE - cast(QCOM*PRIX as decimal(9,2))
```

e) Fonctions temporelles

- `extract(u from d_t)` : donne, sous forme numérique, le composant *u* de la valeur temporelle *d_t*; les valeurs de *u* sont : *year*, *month*, *day*, *hour*, *minute*, *second*.

11. `Instr(ch1, ch2)` en Access.

12. `&` en Access.

13. `mid(ch, I, L)` en Access.

Exemples

```
extract (year from DATECOM) + 1
extract (hour from current_time) > 18
```

f) Fonctions de sélection

Ces fonctions renvoient une valeur choisie parmi plusieurs.

- `case` *when* *c1* *then* *v1*
 when *c2* *then* *v2*
 ...
 else *vn*

`end`

renvoie la valeur *v1* si la condition *c1* est vraie, sinon, renvoie *v2* si *c2* est vraie, ... , sinon renvoie *vn*;

- `case` *expr*
 when *V1* *then* *v1*
 when *V2* *then* *v2*
 ...
 else *vn*

`end`

variante de la précédente pour des conditions de la forme *expr* = *Vi*;

- `coalesce` (*expr1*, *expr2*, ..., *exprn*) : renvoie la première expression différente de null, soit null s'il n'y en a pas;
- `nullif` (*expr1*, *expr2*) : renvoie *expr1* si celle-ci est non null, *expr2* sinon;

Exemples

```
select NCLI,
       case substring(CAT from 1 for 1)
         when 'A' then 'bon'
         when 'B' then 'moyen'
         when 'C' then 'occasionnel'
         else      'inconnu'
       end, LOCALITE
from CLIENT
```

g) Registres du système

Le SGBD fournit des valeurs courantes relatives à l'environnement de l'utilisateur au moment de l'exécution de la requête :

- `current_user` : l'identification de l'utilisateur courant;
- `current_date` : la date courante
- `current_time` : l'instant courant
- `current_timestamp` : date + instant courant.

La requête suivante ne renverra un résultat que si elle est exécutée le premier du mois :

```
select NPRO, LIBELLE, QSTOCK
from   CLIENT
where  QSTOCK < 0
and extract(day of current_date) = 1
```

5.2.9 Les fonctions agrégatives (ou statistiques)

Il existe également des fonctions prédéfinies qui donnent une valeur agrégée calculée pour les lignes sélectionnées :

- `count (*)` donne le nombre de lignes trouvées,
- `count (nom-colonne)` donne le nombre de valeurs de la colonne,
- `avg (nom-colonne)` donne la moyenne des valeurs de la colonne,
- `sum (nom-colonne)` donne la somme des valeurs de la colonne,
- `min (nom-colonne)` donne le minimum des valeurs de la colonne,
- `max (nom-colonne)` donne le maximum des valeurs de la colonne.

La requête ci-dessous fournit une table d'une ligne décrivant la répartition (moyenne, écart maximum, nombre) des montants des comptes des clients de Namur :

```
select 'Namur', avg (COMPTE) as Moyenne,
      max (COMPTE) - min (COMPTE) as Ecart_max,
      count (*) as Nombre
from   CLIENT
where  LOCALITE = 'Namur'
```

La table résultat serait la suivante :

Namur	Moyenne	Ecart-max	Nombre
Namur	-2520	4580	4

Il est à noter, comme nous allons l'expérimenter, que ces fonctions, à l'exception de la première (`count`), ne considèrent que les valeurs non null de la colonne. En outre, chaque valeur est prise en compte, même si elle apparaît plus d'une fois.

Remarque

Le *nom de colonne* dans les expressions `sum` et `avg` peut être remplacé par toute expression à valeur numérique, comme dans la requête suivante, qui demande la valeur des stocks des produits en sapin :

```
select sum(QSTOCK*PRIX)
from   PRODUIT
where  LIBELLE like '%SAPIN%'
```

a) Attention aux valeurs dupliquées

Ici encore l'existence de lignes dupliquées dans les données avant application de la fonction agrégative peut poser un problème. C'est ainsi que la requête suivante ne

donne pas le *nombre de clients ayant passé au moins une commande*, soit 5, mais bien le *nombre de commandes*, c'est-à-dire 7.

```
select count(NCLI)
from   COMMANDE
```

La requête suivante produirait évidemment¹⁴ le même résultat.

```
select distinct count(NCLI)
from   COMMANDE
```

Pour obtenir le nombre de clients, il faut qualifier l'argument de `count` du modifieur `distinct`; nous voulons en effet compter les *valeurs distinctes de NCLI* :

```
select count(distinct NCLI)
from   COMMANDE
```

L'usage du modifieur `distinct` est généralisable à plusieurs fonctions agrégatives¹⁵. Considérons la requête simple suivante, qui compte le nombre de clients, et dont le résultat est, sans surprise, 16 :

```
select count(*)
from   CLIENT
```

La formulation suivante est, elle aussi, sans surprise, mais nous renseigne sur le traitement des valeurs `null` :

```
select count(NCLI) as Nombre,
       count(NOM) as Noms,
       count(LOCALITE) as Localités,
       count(CAT) as Catégories
from   CLIENT
```

Nombre	Noms	Localités	Catégories
16	16	16	14

La fonction agrégative s'applique à la collection des valeurs non `null`, à raison d'une valeur par ligne satisfaisant le critère de sélection éventuel. On comparera ce résultat à celui de la requête suivante :

```
select count(distinct NCLI) as Nombre,
       count(distinct NOM) as Noms,
       count(distinct LOCALITE) as Localités,
       count(distinct CAT) as Catégories
from   CLIENT
```

14. Pourquoi *évidemment* ?

15. Mais n'est malheureusement pas reconnue par MS Access.

Nombre	Noms	Localités	Catégories
16	15	7	4

b) ... et aux ensembles vides

Toute fonction agrégative produit une unique valeur à partir d'un ensemble d'éléments. Quelle est cette valeur lorsque l'ensemble est vide ?

La réponse est assez intuitive : cette valeur est 0 pour count et inconnu (c'est à dire null) pour toutes les autres fonctions.

```
select count(*) as Somme, sum(COMPTE) as Somme,
       max(CAT) as Max
from   CLIENT
where  LOCALITE = 'Alger'
```

Nombre	Somme	Max
0	<null>	<null>

5.3 SÉLECTION UTILISANT PLUSIEURS TABLES : LES SOUS-REQUÊTES

Les requêtes que nous allons étudier extraient encore des données d'une seule table, mais cette fois, les lignes sources sont sélectionnées en fonction de leur liaison à des lignes déterminées appartenant à d'autres tables. On pourra parler de **condition d'association**. Par exemple, on sélectionnera les *commandes des clients de Namur*, ou encore les *clients qui commandent le produit PA60*.

5.3.1 Les sous-requêtes

Considérons les clients qui habitent dans une localité donnée. Il est possible d'en retrouver les numéros en posant la requête suivante :

```
select NCLI
from   CLIENT
where  LOCALITE = 'Namur'
```

dont l'exécution nous donnerait :

NCLI
B062
C123
L422
S127

Il est alors aisé de retrouver les commandes de ces clients de Namur :

```
select NCOM, DATECOM
from   COMMANDE
where  NCLI in ('C123','S127','B062','L422')
```

Cette procédure n'est évidemment pas très pratique. Il serait plus judicieux de remplacer cette liste de valeurs par l'expression qui a permis de les extraire de la table CLIENT. On peut en effet écrire :

```
select NCOM, DATECOM
from   COMMANDE
where  NCLI in ( select NCLI
                  from   CLIENT
                  where  LOCALITE = 'Namur')
```

Cette **structure emboîtée** correspond à l'association des tables COMMANDE et CLIENT sur la base de valeurs identiques de NCLI. Une structure select-from qui intervient dans une forme where est appelée une **sous-requête**.

Une sous-requête peut elle-même contenir une sous-requête. La requête suivante donne les *produits qui ont été commandés par au moins un client de Namur*.

```
select *
from   PRODUIT
where  NPRO in
      ( select NPRO
        from   DETAIL
        where  NCOM in
              ( select NCOM
                from   COMMANDE
                where  NCLI in
                      ( select NCLI
                        from   CLIENT
                        where  LOCALITE='Namur'))))
```

5.3.2 Sous-requête et clé étrangère multi-composant

On observera que les sous-requêtes exploitent les liens entre tables formés par le couplage d'un identifiant et d'une clé étrangère. Qu'en est-il des *clés étrangères multi-composants* ? Considérons à titre d'exemple le schéma de la figure 3.9, et recherchons les comptes auxquels ont été imputés des achats faits le 12-09-2005. On pourra écrire la requête suivante, qui est basée sur une sous-requête qui définit plus d'une colonne:

```
select *
from   COMPTE
where  (NCLI,NFOURN) in ( select NCLI,NFOURN
                        from   ACHAT
                        where  DATEA='12-09-2005')
```

Tous les SGBD n'acceptent cependant pas cette syntaxe. Nous verrons plus loin d'autres formulations permettant d'exprimer une telle requête (sections 5.3.5 et 5.4.6).

5.3.3 Attention aux conditions d'association négatives

a) Le problème

La requête suivante désigne les commandes (c'est-à-dire les lignes de la table **COMMANDE**) *qui ne spécifient pas le produit PA60*, autrement dit, celles pour lesquelles *il n'existe aucun détail spécifiant PA60*.

```
Q1:  select NCOM, DATECOM, NCLI
      from  COMMANDE
      where NCOM not in ( select NCOM
                          from  DETAIL
                          where  NPRO = 'PA60')
```

La sous-requête (`select NCOM from DETAIL where NPRO = 'PA60'`) désigne ici les numéros des commandes dont au moins un détail spécifie le produit PA60. La commande qui contient un de ces détails est donc à rejeter (`not in`). Contrairement aux requêtes précédentes, on retient les lignes **qui ne sont pas associées** aux éléments d'un ensemble déterminé de lignes.

La requête suivante est parfois proposée, à tort, comme autre solution à ce problème. En fait, cette requête désigne *les commandes qui spécifient au moins un produit différent de PA60* (mais qui par ailleurs peuvent également spécifier le produit PA60 !).

```
Q2:  select NCOM, DATECOM, NCLI
      from  COMMANDE
      where NCOM in ( select NCOM
                      from  DETAIL
                      where  NPRO <> 'PA60')
```

Ces deux requêtes, qui sont basées sur des **conditions de sélection négatives**, doivent être examinées soigneusement car ces dernières sont la source d'erreurs fréquentes. On analysera en particulier le résultat à obtenir sur la base de la partition¹⁶ suivante de l'ensemble des commandes (figure 5.1) :

- C1 : les commandes qui n'ont pas de détails (cas rare !),
- C2 : les commandes dont aucun détail ne spécifie PA60,
- C3 : les commandes ayant à la fois un détail spécifiant PA60 et au moins un détail spécifiant un autre produit,
- C4 : les commandes ne spécifiant que PA60.

16. On rappelle que l'ensemble de sous-ensembles non vides $\{E_1, E_2, \dots, E_n\}$, pour $n > 1$, forme une **partition** de E , non vide, ssi, $(E = E_1 \cup E_2 \cup \dots \cup E_n)$ et $(\forall i, j \in [1..n], i \neq j \Rightarrow E_i \cap E_j = \emptyset)$.

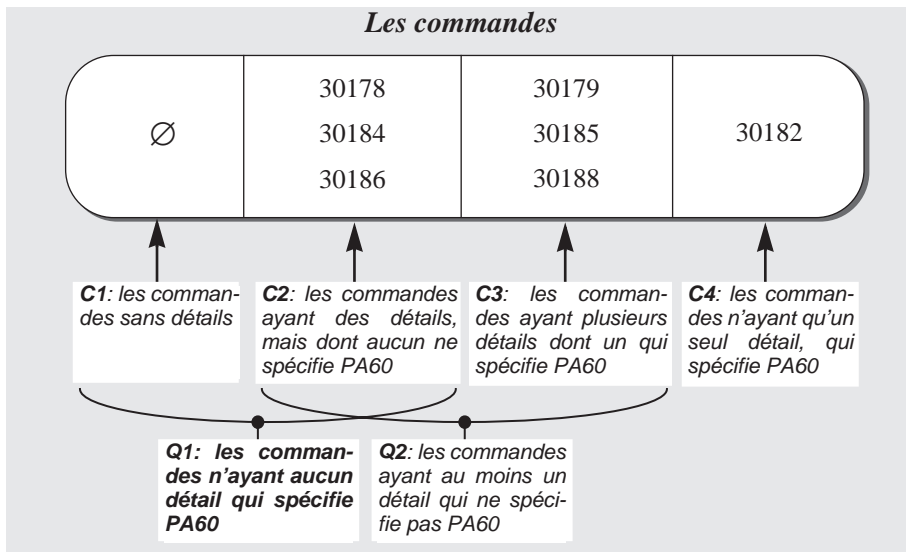


Figure 5.1 - Une partition de l'ensemble des commandes explicitant la portée des requêtes Q1 et Q2. Dans un souci de généralité, on admet qu'une commande puisse n'avoir aucun détail

La première requête correspond aux catégories C1 et C2, tandis que la seconde correspond aux catégories C2 et C3.

b) Méthode générale

D'une manière générale, on traitera ce type de requêtes de la manière suivante.

1. On identifie les deux ensembles de lignes A et B tels que la partie critique de la requête puisse se paraphraser sous la forme *on sélectionne les A qui ne sont pas associés à des B*.
2. On repère les tables et les conditions de sélection qui définissent A et B, soient TA where (CA) et TB where (CB).
3. On repère dans TA et TB les colonnes FA et FB constituant l'identifiant et la clé étrangère dont l'égalité matérialise l'association.
4. On peut alors construire la requête sous la forme :

```
from TA
where (CA)
and   FA not in (select FB
                  from TB
                  where (CB))
```

Appliquons cette procédure à un autre exemple : *sélectionner les clients de Poitiers qui n'ont pas commandé en 2005*.

1. Les deux ensembles sont A = {les clients de Poitiers} et B = {les commandes de 2005}.
2. Ils s'expriment respectivement par "CLIENT where LOCALITE = 'Poitiers'" et "COMMANDE where DATECOM like '%2005'".

3. Les colonnes en correspondance sont respectivement NCLI dans CLIENT et NCLI dans COMMANDE.
4. D'où la requête :

```

from CLIENT
where LOCALITE = 'POITIERS'
and NCLI not in (select NCLI
                  from COMMANDE
                  where DATECOM like '%2005')

```

5.3.4 Références multiples à une même table

a) Premiers exemples

Une sous-requête peut être définie sur la même table que la requête qui la contient. La requête suivante répond à la question : *Quels sont les clients qui habitent dans la même localité¹⁷ que le client n° B512 ?*

```

select *
from CLIENT
where LOCALITE in ( select LOCALITE
                    from CLIENT
                    where NCLI = 'B512')

```

La requête ci-dessous recherche les commandes qui spécifient une quantité du produit PA60 inférieure à celle que spécifie la commande 30182 pour ce même produit.

```

select *
from COMMANDE
where NCOM in (select NCOM
                from DETAIL
                where NPRO = 'PA60'
                and QCOM < (select QCOM
                            from DETAIL
                            where NPRO = 'PA60'
                            and NCOM = '30182'))

```

Il n'y a ici aucune confusion possible quant à savoir à quelle ligne de DETAIL font référence les noms NPRO et NCOM de la deuxième sous-requête. La règle est qu'ils sont relatifs à la requête la plus emboîtée dont la table contient une colonne de ce nom. On notera la double apparition de la condition NPRO = 'PA60'. Pourquoi ?

On peut forcer cette règle de deux manières :

- si deux tables ont deux colonnes de même nom, on pourra préfixer le nom de la colonne du nom de la table. On écrira par exemple COMMANDE.NCLI et CLIENT.NCLI afin de distinguer les deux colonnes de nom NCLI dans le schéma;

17. On notera que le client B512 apparaît dans le résultat. On pourrait l'en exclure aisément. Comment ?

- on peut également attribuer un synonyme, dit **alias de table**, à un nom de table apparaissant dans une clause `from`. Si nécessaire, on qualifiera le nom de colonne de ce synonyme. Le terme `as` est facultatif, et sera omis à l'occasion.

```
select *
from   PRODUIT as P1
where  P1.PRIX > 123
```

b) Les sous-requêtes corrélées

Dans certaines formes, l'alias de table est utilisé pour établir une corrélation entre la requête principale et une sous-requête. A titre d'illustration, exprimons la requête suivante : sélectionner les clients dont le compte est supérieur à la moyenne des comptes des clients de la même commune.

```
select NCLI, NOM, LOCALITE, COMPTE
from   CLIENT as C
where  COMPTE > (select avg (COMPTE)
                  from   CLIENT
                  where  LOCALITE = C.LOCALITE)
order by LOCALITE, COMPTE, NCLI18
```

Ce qui nous donne :

NCLI	NOM	LOCALITE	COMPTE
C123	MERCIER	Namur	-2300
L422	FRANCK	Namur	0
B112	HANSENNE	Poitiers	1250
D063	MERCIER	Toulouse	-2250
C003	AVRON	Toulouse	-1700
F011	PONCELET	Toulouse	0
K729	NEUMAN	Toulouse	0

c) Extension des opérateurs de comparaison

Si la sous-requête renvoie une seule ligne, il est permis d'utiliser les opérateurs de comparaison classiques :

```
select *
from   CLIENT
where  COMPTE > ( select COMPTE
                  from   CLIENT where NCLI = 'C400')
```

De même, l'opérateur "in" pourra s'écrire "=" et "not in" s'écrira aussi "<>".

18. Cette clause, qui peut être ignorée, sera expliquée plus tard.

d) Conditions d'association quantifiées

Il est intéressant de sélectionner les lignes d'une table qui sont associées, non pas à au moins une des lignes d'une autre table qui vérifient une certaine condition, mais bien à un nombre défini de ces lignes. La requête suivante recherche les commandes d'au moins 3 détails.

```
select NCOM, DATECOM, NCLI
from   COMMANDE C
where  (select count(*)
        from DETAIL
        where NCOM = C.NCOM) >= 3
```

e) Null = null ?

C'est l'occasion d'examiner d'un peu plus près les moeurs sociales de la valeur null. De manière assez surprenante, la requête :

```
select NCLI from CLIENT
where  CAT in ( select CAT
                from   CLIENT
                where  NCLI = 'D063')
```

... renvoie une réponse vide, alors que celle-ci devrait au moins contenir la référence D063 ! La raison en est très simple : la valeur de CAT du client D063 est null, or null n'est en principe comparable à rien, même pas à lui-même¹⁹, puisqu'il est généralement interprété comme *inconnu*. Nous rediscuterons de cette question plus tard (section 6.10).

5.3.5 Les quantificateurs ensemblistes

a) Les formes de base

Une condition peut porter sur l'existence (*exists*) ou l'inexistence (*not exists*) d'au moins une ligne dans le résultat d'une sous-requête, ou, en d'autres termes, sur le fait que le résultat d'une sous-requête est *non vide* ou *vide*.

La requête suivante recherche les produits qui ne sont pas commandés pour l'instant, c'est-à-dire ceux pour lesquels il n'existe aucune ligne de DETAIL²⁰. La sous-requête désigne, pour chaque ligne P de PRODUIT, l'ensemble des lignes de DETAIL qui y font référence, c'est-à-dire dont NPRO est égal à NPRO de P.

```
select NPRO, LIBELLE
from   PRODUIT as P
where  not exists ( select *
                    from   DETAIL
                    where  NPRO = P.NPRO)
```

19. Il y a malheureusement des exceptions (*group by* par exemple).

20. Il existe une autre formulation de cette requête qui utilise la relation de comparaison *not in*. Laquelle ?

Les quantificateurs *any* (et son synonyme *some*) et *all* permettent, quant à eux, de comparer une valeur à celles d'un ensemble défini par une sous-requête. *any* signifie qu'*au moins un* élément de l'ensemble satisfait la comparaison et *all* signifie que *tous* les éléments la satisfont.

La requête suivante désigne les détails de commande spécifiant la quantité minimum du produit PA60, c'est-à-dire tels qu'il n'existe pas de détails spécifiant une quantité inférieure :

```
select *
from   DETAIL
where  QCOM <= all (select QCOM
                    from   DETAIL
                    where  NPRO = 'PA60')

and    NPRO = 'PA60'
```

La condition `QCOM <= all (select QCOM from DETAIL where NPRO = 'PA60')` s'interprète ainsi : la valeur de QCOM (de la ligne de DETAIL courante) est inférieure ou égale à **chacun des éléments** de `(select QCOM from DETAIL where NPRO = 'PA60')`.

Quant à la requête ci-dessous, elle désigne les détails de commande de PA60 dont la quantité commandée n'est pas minimale :

```
select *
from   DETAIL
where  QCOM > any ( select QCOM
                    from   DETAIL
                    where  NPRO = 'PA60')

and    NPRO = 'PA60'
```

La condition `QCOM > any (select QCOM from DETAIL where NPRO = 'PA60')` s'interprète comme suit : la valeur de QCOM est supérieure à **au moins un des éléments** de `(select QCOM from DETAIL where NPRO = 'PA60')`.

On notera l'équivalence entre les expressions :

```
in      ≡ = any
not in  ≡ <> all
```

b) Sous-requête et clé étrangère multi-composant

Reportons-nous au schéma de la figure 3.9, et recherchons, comme nous l'avons déjà fait, *les comptes auxquels ont été attachés des achats faits le 23-04-2005*. Les quantificateurs ensemblistes nous permettent d'exprimer cette requête d'une manière élégante :

```
select *
from   COMPTE C
where  exists ( select *
                from   ACHAT
                where  NCLI = C.NCLI
                and    NFOURN = C.NFOURN
                and    DATEA = '23-04-2005' )
```

c) Condition de totalité (pour tout)

Parmi les classes de conditions d'association vues jusqu'ici, il en est une que nous n'avons pas abordée : puisque SQL propose le quantificateur existentiel `exists` (concrétisation du quantificateur \exists), il devrait aussi proposer le quantificateur universel `for all` (analogue à \forall). Ce dernier n'est cependant pas proposé, de sorte qu'il nous faudra tourner ce type de condition d'une autre manière.

Recherchons par exemple *les commandes qui spécifient tous les produits*. Considérons une (ligne de) **COMMANDE** M. Celle-ci est sélectionnée si les **PRODUITS** commandés par M et les **PRODUITS** forment deux ensembles identiques, c'est-à-dire que le second ensemble est inclus dans le premier²¹. Ou encore, M est retenue si, *pour tout PRODUIT P, P est dans l'ensemble des PRODUIT commandés par M*. Le quantificateur *pour tout* n'existant pas, nous allons appliquer l'équivalence suivante, qui permet de se passer du quantificateur \forall :

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

Il vient :

la **COMMANDE** M est retenue *s'il n'existe pas de PRODUIT P, tel que P n'est pas dans l'ensemble des PRODUITS commandés par M*.

La traduction mot-à-mot de cette formule en SQL ne pose pas de problèmes insurmontables :

la COMMANDE M est retenue	→	<code>select NCOM from COMMANDE M</code>
si,	→	<code>where</code>
il n'existe pas	→	<code>not exists</code>
de PRODUIT P,	→	<code>(select * from PRODUIT P</code>
tel que	→	<code>where</code>
P n'est pas dans	→	<code>P.NPRO not in</code>
l'ensemble des PRODUITS	→	<code>(select NPRO from DETAIL</code>
commandés par M.	→	<code>where NCOM = M.NCOM))</code>

En rassemblant ces fragments, on obtient (l'alias P, désormais inutile, pourrait être ignoré) :

```
select NCOM
from   COMMANDE M
where  not exists (select *
                  from   PRODUIT P
                  where   P.NPRO not in (select NPRO
                                          from   DETAIL
                                          where   NCOM = M.NCOM) )
```

La sous-requête interne extrait les produits référencés par la commande courante M; la sous-requête intermédiaire définit l'ensemble des produits que M ne référence pas;

21. Sachant que le premier est forcément inclus dans le second.

la requête principale ne retient que les commandes pour lesquelles cet ensemble est vide.

Examinons deux variantes équivalentes. La première se déduit de l'observation que les valeurs de NCOM qui nous intéressent sont toutes présentes dans la table DETAIL. On peut donc réécrire la requête sous la forme :

```
select NCOM
from   DETAIL M
....
```

La seconde exploite la propriété qui veut que si l'ensemble A est inclus dans B, et que A et B sont de même taille, alors $A = B$. Considérons les lignes de DETAIL relatives à une commande. L'ensemble des valeurs de NPRO de ces lignes représente les produits commandés. S'il contient de valeurs qu'il y a de lignes dans la table PRODUIT, alors cette commande spécifie tous les produits. On peut alors écrire²² :

```
select NCOM
from   DETAIL
group by NCOM
having count(distinct NPRO) = (select count(*) from PRODUIT)
```

5.4 EXTRACTION DE DONNÉES DE PLUSIEURS TABLES (JOINTURE)

Jusqu'ici, nous avons extrait des données, brutes ou dérivées, issues d'une seule table. Nous examinerons dans cette section comment coupler les lignes de deux ou plusieurs tables afin d'en extraire des données corrélées.

5.4.1 La jointure de plusieurs tables

Pour coupler deux tables, il faut d'abord préciser ces tables (clause `from`), ainsi que la règle d'association des lignes de ces deux tables (clause `where`) dont les valeurs sont extraites (clause `select`). Cette règle se présente généralement sous la forme de l'égalité des valeurs de deux colonnes. Une telle opération d'association de tables porte le nom de **jointure** (en anglais *join*).

La requête ci-dessous extrait les informations sur les commandes, complétées pour chacune d'elles de renseignements sur le client qui l'a émise.

```
select NCOM, CLIENT.NCLI, DATECOM, NOM, LOCALITE
from   COMMANDE, CLIENT
where  COMMANDE.NCLI = CLIENT.NCLI
```

Son évaluation donnerait la table suivante.

22. Dans cet exemple, le modificateur `distinct` n'est pas nécessaire. Pourquoi ?

NCOM	NCLI	DATECOM	NOM	LOCALITE
30178	K111	21/12/2005	VANBIST	Lille
30179	C400	22/12/2005	FERARD	Poitiers
30182	S127	23/12/2005	VANDERKA	Namur
30184	C400	23/12/2005	FERARD	Poitiers
30185	F011	2/01/2006	PONCELET	Toulouse
30186	C400	2/01/2006	FERARD	Poitiers
30188	B512	3/01/2006	GILLET	Toulouse

Conceptuellement, le résultat pourrait être obtenu comme suit :

1. On construit une table en couplant chaque ligne de la première table à chaque ligne de la seconde :

```
from COMMANDE, CLIENT.
```

Cette table contient $6 + 3 = 9$ colonnes et $16 \times 7 = 112$ lignes.

2. On sélectionne, parmi les lignes ainsi obtenues, celles qui vérifient la condition d'association (ainsi que les autres conditions éventuelles) :

```
where COMMANDE.NCLI = CLIENT.NCLI,
```

3. On ne retient alors que les colonnes demandées :

```
select NCOM, CLIENT.NCLI, DATECOM, NOM, LOCALITE.
```

En général, le SGBD ne procède pas de la sorte, car il en résulterait le plus souvent une grande inefficacité. Cependant, quelle que soit la manière dont le SGBD retrouve les données demandées, le résultat sera le même.

Par extension, la jointure de trois tables réclamera deux conditions d'association. L'ordre des tables dans la clause `from` est indifférent, de même que celui des conditions dans la clause `where`.

```
select CLIENT.NCLI, NOM, DATECOM, NPRO
from   CLIENT, COMMANDE, DETAIL
where  CLIENT.NCLI = COMMANDE.NCLI
and    COMMANDE.NCOM = DETAIL.NCOM
```

On peut avancer deux interprétations équivalentes de cette requête :

1. comme ci-dessus, on constitue une table dont les lignes sont formées d'une ligne de chacune des trois tables de la clause `from`; on ne retient que celles qui vérifient la clause `where`;
2. on effectue la jointure de deux tables (par exemple `CLIENT` et `COMMANDE`); ensuite, on effectue une deuxième jointure de ce résultat avec la troisième table (`DETAIL`).

5.4.2 Conditions de jointure et conditions de sélection

Les conditions telles que `COMMANDE.NCLI = CLIENT.NCLI` apparaissant dans les requêtes de jointure sont appelée **conditions de jointure**, car elles régissent les asso-

ciations entre les lignes de **COMMANDE** et de **CLIENT**. On observera d'ailleurs que dans cette condition de jointure, **COMMANDE.NCLI** correspond à une clé étrangère et **CLIENT.NCLI** à un identifiant primaire.

Techniquement, cependant, il s'agit d'une condition ordinaire appliquée aux colonnes de chaque couple de lignes, et qui peut apparaître au milieu d'autres conditions de sélection, comme dans la requête suivante, qui limite le résultat aux clients de catégorie C1 et aux commandes antérieures au 23 décembre 2005.

```
select NCOM, CLIENT.NCLI, DATECOM, NOM, ADRESSE
from   COMMANDE, CLIENT
where  COMMANDE.NCLI = CLIENT.NCLI
and    CAT = 'C1'
and    DATECOM < '23-12-2005'
```

5.4.3 Jointures sans conditions : produit relationnel

Une jointure sans condition de jointure telle que la suivante n'est pas interdite :

```
select NCOM, CLIENT.NCLI, DATECOM, NOM, ADRESSE
from   COMMANDE, CLIENT
```

Elle risque cependant d'être extrêmement coûteuse (le résultat contiendrait ici $16 \times 7 = 112$ lignes) et n'offrirait aucun intérêt. Sauf justification, il faudra la considérer comme une erreur. Cette opération porte le nom de *produit relationnel*.

5.4.4 La jointure et les lignes célibataires - Les opérateurs ensemblistes

a) Les lignes célibataires

Assez logiquement, les clients qui n'ont passé aucune commande n'apparaissent pas dans le résultat. Nous qualifierons de *célibataires* les lignes qui leur correspondent. En toute généralité, la jointure de deux tables exclut dans chacune d'elle les *lignes célibataires*, c'est à dire les lignes qui n'ont pas de correspondants dans l'autre table. Dans le cas qui nous occupe, il existe des *clients célibataires*, mais en revanche il n'y a pas de *commandes célibataires* (en effet la clé étrangère est une colonne obligatoire, de sorte qu'il n'y a pas de commande sans client). Il existe une variante de l'opérateur de jointure, dénommée *jointure externe*, qui permet d'inclure les lignes célibataires. Nous en reparlerons plus loin (*outer join*, section 6.3.2).

b) L'union

Si nous désirons réintégrer les *clients sans commandes* dans la jointure, nous pouvons faire appel à l'opérateur *union*, qui permet d'ajouter au résultat d'une requête celui d'une autre requête. La requête évoquée pourrait s'écrire comme suit :

```
select NCOM, CLIENT.NCLI, DATECOM, NOM, LOCALITE
from   COMMANDE, CLIENT
where  COMMANDE.NCLI = CLIENT.NCLI
union
select '--', NCLI, '--', NOM, LOCALITE
```

```

from   CLIENT
where  not exists (select * from COMMANDE
                  where NCLI = CLIENT.NCLI)

```

Il en résulterait la table suivante. On notera, sans surprise, que certains clients apparaissent plus d'une fois, selon le nombre de commandes qu'ils ont passées.

NCOM	NCLI	DATECOM	NOM	LOCALITE
30178	K111	21/12/2005	VANBIST	Lille
30179	C400	22/12/2005	FERARD	Poitiers
30182	S127	23/12/2005	VANDERKA	Namur
30184	C400	23/12/2005	FERARD	Poitiers
30185	F011	2/01/2006	PONCELET	Toulouse
30186	C400	2/01/2006	FERARD	Poitiers
30188	B512	3/01/2006	GILLET	Toulouse
--	B062	--	GOFFIN	Namur
--	B112	--	HANSENNE	Poitiers
--	B332	--	MONTI	Genève
--	C003	--	AVRON	Toulouse
--	C123	--	MERCIER	Namur
--	D063	--	MERCIER	Toulouse
--	F010	--	TOUSSAINT	Poitiers
--	F400	--	JACOB	Bruxelles
--	K729	--	NEUMAN	Toulouse
--	L422	--	FRANCK	Namur
--	S712	--	GUILLAUME	Paris

c) Les opérateurs ensemblistes

Le fonctionnement de l'opérateur `union` est un peu particulier. En effet, il produit un *ensemble* de lignes, c'est à dire une collection de lignes *distinctes*. Si une même ligne apparaît dans le résultat de chacun des membres de l'union, cette ligne n'apparaîtra qu'une seule fois dans l'union. Plus curieux : si le résultat d'un des membres contient une ligne en plusieurs exemplaires, ce qui peut arriver si aucun identifiant n'est repris dans la clause `select`, un seul exemplaire apparaîtra dans l'union. À titre d'illustration, on comparera la requête :

```
select LOCALITE from CLIENT where CAT = 'C1'
```

```

LOCALITE
-----
Poitiers
Namur
Poitiers
Namur
Nalur

```

avec la suivante, qui lui *ajoute* un élément :

```
select LOCALITE from CLIENT where CAT = 'C1'
union
select LOCALITE from CLIENT where COMPTE < 0
```

LOCALITE

Namur
Poitiers
Toulouse

Si on désire empêcher l'élimination des lignes en double, on utilisera l'opérateur **union all**. Une même ligne qui apparaît m fois dans le premier membre et n fois dans le second apparaîtra $m+n$ fois dans le résultat, comme le montre la requête ci-dessous.

```
select LOCALITE from CLIENT where CAT = 'C1'
union all
select LOCALITE from CLIENT where COMPTE < 0
```

LOCALITE

Poitiers
Namur
Poitiers
Namur
Namur
Namur
Toulouse
Toulouse
Namur
Toulouse
Namur

D'autres opérateurs ensemblistes comme l'intersection (*intersect*), qui construit l'ensemble des éléments simultanément présents dans les deux collections, et la différence (*except*), qui construit l'ensemble des éléments appartenant à la première collection mais pas à la seconde, sont également disponibles.

Munis de la clause **all**, ces opérateurs préservent les lignes en double selon un principe similaire à *union all*. Pour une même ligne respectivement en m et n exemplaires,

- *intersect all* produira $\min(m,n)$ exemplaires de cette ligne,
- *except all* produira $\max(m-n,0)$ exemplaires de cette ligne.

d) Produit relationnel : une application pratique

Nous avons vu que le produit relationnel nous permettait de comprendre le mécanisme de la jointure, mais qu'il n'offrait pas grand intérêt en soi. Citons cependant une application intéressante de cet opérateur. Considérons d'abord tous les couples (LOCALITE, NPRO) *possibles* issus de la base de données :

```
select distinct LOCALITE, NPRO
from   CLIENT, PRODUIT
```

Construisons maintenant tous les couples *effectifs*, tels qu'on commande *réellement* le produit dans la localité :

```
select distinct LOCALITE, NPRO
from   CLIENT C, COMMANDE M, DETAIL D
where  C.NCLI=M.NCLI
and    M.NCOM=D.NCOM
```

Le service commercial sera certainement très intéressé par les **produits qu'on ne commande pas** dans chaque commune, et pour lesquels un effort d'information serait utile (les deux clauses *distinct* sont inutiles. Pourquoi ?) :

```
select distinct LOCALITE, NPRO
from   CLIENT, PRODUIT
except
select distinct LOCALITE, NPRO
from   CLIENT C, COMMANDE M, DETAIL D
where  C.NCLI=M.NCLI
and    M.NCOM=D.NCOM
```

e) Remarque sur l'intersection et la différence

Ces opérateurs ne sont pas strictement indispensables (d'ailleurs certains SGBD ne les offrent pas) dans la mesure où ils peuvent être exprimés simplement par les requêtes standards.

L'*intersection* de deux tables est obtenue par leur jointure²³, car celle-ci reprend les éléments qui sont simultanément présents dans ces tables.

La *différence* s'exprimera par le prédicat *not in* dont la sous-requête définit les éléments de la seconde collection. On utilisera aussi la forme *not exists*.

5.4.5 Les requêtes sur des structures de données cycliques

On qualifie de **cyclique** (ou récursive) une structure de données qui fait, directement ou non, référence à elle-même. La table **PERSONNE** illustrée à la figure 5.2 présente une structure cyclique car la colonne **RESPONSABLE** est une clé étrangère vers la table **PERSONNE** elle-même. Le rôle de cette colonne est de désigner le responsable direct de chaque personne, s'il existe.



Figure 5.2 - Un schéma cyclique : la table **PERSONNE** se référence elle-même

23. L'opération évoquée ici, qui retient les lignes d'une table qui sont *joignables* avec celles d'une autre table, est appelée, très logiquement, *semi-jointure*.

Ce responsable étant lui-même une personne peut donc aussi avoir un responsable²⁴, et ainsi de suite. Cette table est déclarée comme suit :

```
create table PERSONNE ( NPERS      char (4) not null,
                        NOM        char(25) not null,
                        RESPONSABLE char (4),
                        primary key (NPERS),
                        foreign key (RESPONSABLE)
                        references PERSONNE)
```

La figure 5.3 représente un exemple de contenu de la table **PERSONNE**. On y lit notamment que les personnes p1 et p2 n'ont pas de responsable, que le responsable de p3 et p4 est p1, et que p4 est responsable de p5 et p6, qui elle-même est responsable de p7. On dessinera les relations définies dans la table afin de maîtriser ces concepts.

Cette table permet, par exemple, de répondre à la question suivante : donner, pour chaque personne (S, pour *subordonné*) ayant un responsable (R), le numéro et le nom de celui-ci. Il vient :

```
select S.NPERS, R.NPERS, R.NOM
from   PERSONNE S, PERSONNE R
where  S.RESPONSABLE = R.NPERS
```

PERSONNE		
NPERS	NOM	(RESPONSABLE)
p1	Mercier	--
p2	Durant	--
p3	Noirons	p1
p4	Dupont	p1
p5	Verger	p4
p6	Dupont	p4
p7	Dermiez	p6
p8	Anciers	p2

Figure 5.3 - Un exemple de contenu de la table **PERSONNE**²⁵

24. On considère souvent implicitement que le domaine d'application décrit par une structure cyclique est soumis à des conditions sur le graphe des objets et de leurs inter-relations. Dans cet exemple, on admettra qu'une personne ne peut être son propre responsable, ni directement ni indirectement. En d'autres termes, si on représente la relation *a-pour-responsable* (correspondant à la clé étrangère RESPONSABLE) par un arc orienté d'une personne vers son responsable, ce graphe ne peut présenter de circuits. En fait, cette contrainte ne peut être exprimée en SQL, ni être vérifiée par le SGBD. Si des données définissant un tel circuit venaient à être introduites, certains programmes d'application pourraient présenter un comportement anormal (bouclage infini).

25. Dans un but de lisibilité, les valeurs <null> ont été représentées par "--".

Cette requête construit des couples de personnes, la première étant la personne subordonnée (S) et la seconde son responsable (R). Elle réalise donc la jointure de la table PERSONNE avec elle-même, ce qu'on appelle une **auto-jointure**.

La requête suivante donne, pour chaque personne de nom Dupont, son numéro, ainsi que le numéro et le nom de son responsable s'il existe.

```
select S.NPERS, R.NPERS, R.NOM
from   PERSONNE S, PERSONNE R
where  S.RESPONSABLE = R.NPERS
and    S.NOM = 'Dupont'
union
select NPERS, '--', '--'
from   PERSONNE
where  RESPONSABLE is null
and    NOM = 'Dupont'
```

Seule la première partie de la requête peut poser un problème (la seconde complète la réponse en indiquant les personnes sans responsable, à la manière d'une jointure externe). Pour l'interpréter, supposons qu'on dispose de deux tables décrivant les personnes (figure 5.4). L'une, de nom PERSONNE, représente toutes les personnes; l'autre, de nom SUPERIEUR, ne représente que les personnes qui sont responsables. Il est évident que les personnes qui sont responsables d'autres personnes sont répertoriées dans les deux tables. En fonction du contenu de la table PERSONNE proposé ci-dessus, ces tables contiendraient les données de la figure 5.4. La première partie de la requête précédente s'écrit alors comme suit.

```
select S.NPERS, R.NPERS, R.NOM
from   PERSONNE S, SUPERIEUR R
where  S.RESPONSABLE = R.NPERS
and    S.NOM = 'Dupont'
```

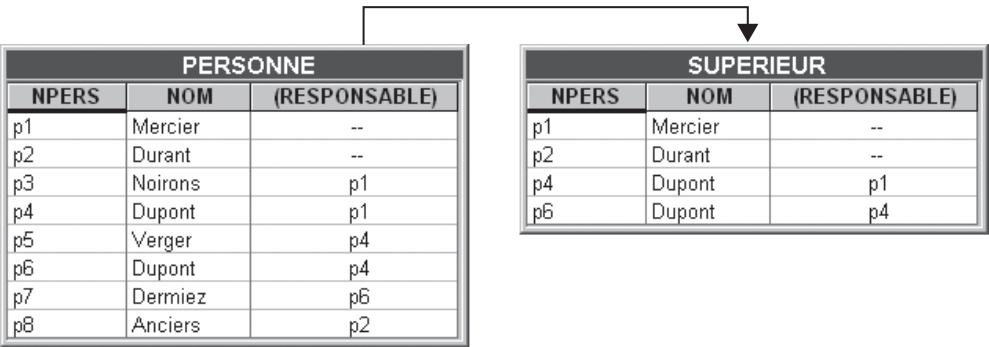


Figure 5.4 - La table SUPERIEUR est un extrait de la table PERSONNE

On observe que la requête donnerait le même résultat si le contenu de SUPERIEUR était le même que celui de PERSONNE. En remplaçant SUPERIEUR par PERSONNE, on obtient alors la requête initiale.

Une table qui, pour la personne de numéro 'p4', contiendrait le numéro et le nom de 'p4', ainsi que le numéro et le nom de ses subordonnés de deuxième niveau

(c'est-à-dire des personnes qui ont pour responsable une personne dont 'p4' est le responsable) s'obtiendrait par une double jointure (on ignore les personnes qui n'ont pas de responsable) :

```
select R.NPERS, R.NOM, SS.NPERS, SS.NOM
from   PERSONNE R, PERSONNE S, PERSONNE SS
where  R.NPERS = 'p4'
and    R.NPERS = S.RESPONSABLE
and    S.NPERS = SS.RESPONSABLE
```

Ces requêtes montrent que SQL ne permet pas d'obtenir facilement tous les responsables, directs et indirects, d'une personne déterminée, ni les personnes qui dépendent, directement ou indirectement, d'une personne déterminée, du moins sans recourir à la programmation procédurale. On dira que SQL, dans ses versions actuelles les plus répandues, ne permet pas d'exprimer des requêtes récursives²⁶.

Un autre exemple classique de structure cyclique décrit une nomenclature de produits. On y indique la composition de chaque produit en sous-produits, ces derniers pouvant également être décomposés en autres sous-produits, et ainsi de suite. Le schéma de la figure 5.5 illustre cette structure. On notera que les relations entre produits ne peuvent plus s'exprimer par une simple clé étrangère ajoutée à la table PRODUIT comme nous l'avions fait pour la table PERSONNE. En effet, alors qu'une personne n'a qu'un seul responsable, un produit peut être constitué de plusieurs composants et entrer dans la composition de plusieurs autres produits. C'est la table COMPOSITION qui représente les relations de composition entre produits. Une ligne $\langle h, b, q \rangle$ indique que le produit b est un composant du produit h , et qu'il faut q unités de b pour fabriquer 1 unité de h . On peut ainsi représenter le fait qu'une unité du produit $p2$ contient 8 unités du produit $p7$ et 2 unités du produit $p8$ (la figure 5.6 représente un exemple de nomenclature).

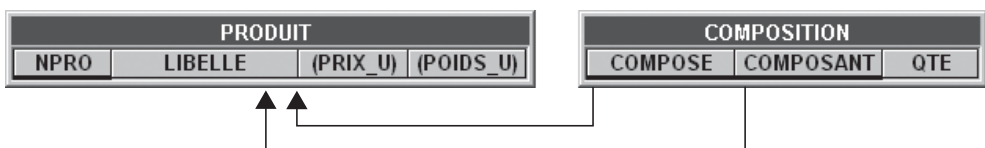


Figure 5.5 - Une structure indirectement cyclique : nomenclature d'une gamme de produits

L'identifiant indique qu'un produit n'est renseigné qu'une seule fois comme composant direct d'un même produit. Les matières premières ont un prix et un poids unitaires qui doivent être fixés; le prix et le poids des autres produits peuvent être calculés à partir des caractéristiques de leurs composants.

26. ORACLE propose une version spécifique de la requête SFW permettant de calculer une jointure cyclique. Nous l'examinerons dans la section 6.3.3. Le lecteur intéressé par les requêtes (récursives) appliquées aux structures de graphes consultera utilement les chapitres 26 et 27 de [Celko,2000].

PRODUIT			
NPRO	LIBELLE	(PRIX_U)	(POIDS_U)
p1	A-200	-	-
p2	A-056	-	-
p3	B-661	3.5	0.70
p4	B-122	8.0	0.25
p5	B-326	5.0	1.15
p6	D-822	-	-
p7	D-507	-	-
p8	G-993	0.5	0.90
p9	F-016	1.7	2.30
p10	J-500	-	-
p11	J-544	-	-
p12	L-009	-	-

COMPOSITION		
COMPOSE	COMPOSANT	QTE
p1	p2	2
p1	p3	1
p1	p4	2
p2	p7	8
p2	p8	2
p3	p8	5
p4	p8	4
p4	p9	5
p4	p10	5
p5	p4	2
p5	p6	7
p9	p11	2
p10	p11	4
p10	p12	3

Figure 5.6 - Un exemple de nomenclature de produits. Les prix et poids unitaires des produits finis et semi-finis, qui sont calculables, n'ont pas été enregistrés explicitement

La requête ci-dessous donne des informations sur le produit 'p4', ainsi que sur sa composition²⁷ :

```
select H.NPRO,H.LIBELLE,C.QTE,B.NPRO,B.LIBELLE
from   PRODUIT H, COMPOSITION C, PRODUIT B
where  C.COMPOSE = H.NPRO
and    C.COMPOSANT = B.NPRO
and    H.NPRO = 'p4'
```

Considérant le contenu des tables représenté à la figure 5.6, on obtient :

H.NPRO	H.LIBELLE	C.QTE	B.NPRO	B.LIBELLE
p4	B-122	4	p8	G-993
p4	B-122	5	p9	F-016
p4	B-122	5	p10	J-500

5.4.6 Sous-requête ou jointure?

On peut faire observer que certaines conditions utilisant une sous-requête (*select* emboîté) peuvent s'exprimer à l'aide d'une jointure. Examinons quelques cas d'application.

27. Dans une relation de composition, les alias H et B désignent respectivement le produit composé (*haut*) et le produit composant (*bas*).

a) Les conditions d'association et de non-association

Il est clair que la requête :

```
select NCOM,DATECOM
from   COMMANDE
where  NCLI in (select NCLI
                from   CLIENT
                where  LOCALITE = 'Poitiers')
```

peut s'écrire également sous la forme d'une jointure :

```
select NCOM,DATECOM
from   COMMANDE, CLIENT
where  COMMANDE.NCLI = CLIENT.NCLI
and    LOCALITE = 'Poitiers'
```

De même, la requête :

```
select *
from   COMMANDE
where  NCOM in (select NCOM
                from   DETAIL
                where  NPRO = 'PA60'
                and    QCOM < (select QCOM
                                from   DETAIL
                                where  NPRO = 'PA60'
                                and    NCOM = '30182'))
```

est-elle équivalente à :

```
select M.NCOM, DATECOM, NCLI
from   COMMANDE M, DETAIL D1, DETAIL D2
where  M.NCOM = D1.NCOM
and    D1.NPRO = 'PA60'
and    D2.NCOM = '30182'
and    D2.NPRO = 'PA60'
and    D1.QCOM < D2.QCOM
```

La condition (M.NCOM = D1.NCOM) associe à chaque commande M chacun de ses détails D1. La condition (D1.NPRO = 'PA60') retient les détails D1 qui spécifient le produit PA60. Les conditions (D2.NCOM = '30182' and D2.NPRO = 'PA60') désignent le détail D2 qui sert de référence. La condition (D1.QCOM < D2.QCOM) établit le critère de sélection des détails D1 par rapport au détail de référence D2. Remarquons enfin que si la jointure entre M et D1 est classique, celle qui concerne D1 et D2 est plutôt spéciale, puisqu'elle ne joue pas sur l'égalité entre un identifiant et une clé étrangère. Nous en reparlerons en 5.4.8.

En revanche, certaines structures de select emboîtés **ne peuvent pas s'exprimer** par une jointure. Tel serait le cas de la recherche des commandes qui ne spécifient pas PA60, vue auparavant et qu'on rappelle :

```

select NCOM, DATECOM, NCLI
from   COMMANDE
where  NCOM not in ( select NCOM
                    from   DETAIL
                    where  NPRO = 'PA60' )

```

Cette requête n'est en effet absolument pas équivalente à :

```

select distinct COMMANDE.NCOM, DATECOM, NCLI
from   COMMANDE, DETAIL
where  COMMANDE.NCOM = DETAIL.NCOM
and    NPRO <> 'PA60'

```

ni d'ailleurs à :

```

select distinct COMMANDE.NCOM, DATECOM, NCLI
from   COMMANDE, DETAIL
where  COMMANDE.NCOM <> DETAIL.NCOM
and    NPRO = 'PA60'

```

Il faut se souvenir qu'une jointure basée sur le couple *identifiant primaire/clé étrangère* permet de matérialiser des associations entre lignes. Elle ne permet pas de spécifier l'*inexistence* d'associations.

Conclusion

La jointure et la sous-requête permettent d'exprimer des **conditions d'association** entre lignes. En revanche, des **conditions de non-association** ne sont généralement exprimables que par des sous-requêtes²⁸, ainsi que, bien sûr, par la forme *not exists* décrite en 5.3.5.

b) Sous-requête et clé étrangère multi-composant

Cette question a déjà été abordée dans la section 5.3.5. Nous pouvons ici lui apporter une réponse supplémentaire. Il s'agit de rechercher (schéma 3.9) *les comptes auxquels ont été attachés des achats faits le 23-04-2005*. Cette requête, que nous avons déjà traduite sous la forme d'une sous-requête et d'une condition existentielle, peut aussi s'exprimer sous la forme d'une jointure :

```

select C.NCLI, C.NFOURN, ...
from   COMPTE C, ACHAT A
where  A.NCLI = C.NCLI
and    A.NFOURN = C.NFOURN
and    DATEA = '23-04-2005'

```

28. Nous verrons cependant, dans la section 6.3.2, comment une jointure externe permet de sélectionner les lignes célibataires.

5.4.7 Valeurs dérivées dans une jointure

La jointure permet également d'effectuer des calculs sur des quantités extraites de plusieurs tables. Le raisonnement est simple : la jointure constitue des lignes fictives dont la clause `select` extrait des valeurs comme elle le ferait d'une ligne réelle issue d'une table. La requête suivante associe à chaque ligne de `DETAIL` le montant à payer.

```
select NCOM, D.NPRO, QCOM*PRIX
from   DETAIL D, PRODUIT P
where  D.NPRO = P.NPRO
```

Quant à la requête suivante, elle établit le montant de la commande 30184.

```
select 'Montant commande 30184 = ', sum(QCOM*PRIX)
from   DETAIL D, PRODUIT P
where  D.NCOM = '30184'
and    D.NPRO = P.NPRO
```

5.4.8 Les jointures généralisées

Les cas de jointure étudiés jusqu'ici étaient basés, sauf exception, sur l'égalité des valeurs d'une *clé étrangère* avec celles d'un *identifiant*. En fait, la forme même de la condition de jointure suggère que toute comparaison peut servir à indiquer comment associer les lignes des tables concernées.

Exemple 1

Considérons par exemple le schéma de la figure 5.7, qui comporte une table `VENTE` dont chaque ligne $\langle c, p, x \rangle$ indique que dans un magasin de la chaîne c , le produit p est vendu au prix x , ainsi qu'une table `LOCALISATION` dont chaque ligne $\langle c, v \rangle$ spécifie qu'un magasin de la chaîne c est implanté dans la ville v . Il est clair que la colonne `CHAINE` de `VENTE` n'est pas une clé étrangère, pas plus qu'elle ne constitue un identifiant de `LOCALISATION`. Tout au plus peut-on espérer que les deux colonnes aient des valeurs en commun. Nous ne sommes donc plus dans le schéma classique représentant des associations explicites.



Figure 5.7 - Deux tables logiquement corrélées via les colonnes `CHAINE`, mais sans clés étrangères

L'expression :

```
select distinct PRODUIT, VILLE, PRIX
from   VENTE V, LOCALISATION L
where  V.CHAINE = L.CHAINE
```

indique, pour chaque ligne $\langle p, v, x \rangle$ du résultat, que le produit p est disponible dans la ville v au prix x , quelles que soient les chaînes qui y proposent p .

Exemple 2

L'exemple suivant est particulièrement intéressant. Il illustre une opération fréquente, qui consiste à condenser une information, telle que les valeurs d'une colonne, de manière à la rendre plus lisible. La table de la figure 5.8 établit des intervalles successifs de valeurs de compte (MIN_CPT et MAX_CPT), et attribue un code à chacun (CODE_CPT).

CLASSE_CPT		
MIN_CPT	MAX_CPT	CODE_CPT
10000	32000	A
5000	10000	B
2000	5000	C
1000	2000	D
500	1000	F
0	500	G
-500	0	U
-1000	-500	V
-2000	-1000	W
-5000	-2000	X
-10000	-5000	Y
-32000	-10000	Z

Figure 5.8 - Table de classification des valeurs des comptes des clients

Nous aimerions associer à chaque client le code de son compte. C'est ce que nous livre la requête suivante²⁹.

```
select NCLI, NOM, CODE_CPT
from   CLIENT, CLASSE_CPT
where  CAT = 'C1'
and    COMPTE >= MIN_CPT and COMPTE < MAX_CPT
```

5.4.9 Interprétation du résultat d'une jointure

La construction d'une requête qui utilise une ou plusieurs jointures peut s'avérer délicate. Il importe donc de bien comprendre ce que représente le résultat d'une jointure. Nous limiterons le propos aux jointures qui sont basées sur l'égalité d'une clé étrangère et d'un identifiant primaire.

a) Quelles entités représente le résultat d'une jointure ?

La question est la suivante. Sachant que toute ligne d'une table représente une entité du domaine d'application (un client, un achat, un détail, etc.), quelles entités les lignes d'une jointure représentent-elles ? Par exemple, chaque ligne produite par l'évaluation de la requête :

29. Compte tenu de la manière dont les intervalles ont été représentés, il n'est pas possible d'utiliser le prédicat *between*. Pourquoi ?


```
select C.NCLI, NOM, LOCALITE
from   CLIENT C, COMMANDE M
where  M.NCLI = C.NCLI
```

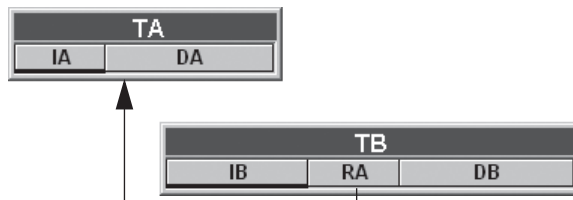
représente-t-elle (1) un client, (2) un client qui a passé une ou plusieurs commandes, (3) une commande ?

Autre formulation : y a-t-il autant de lignes dans le résultat qu'il y a (1) de clients, (2) de clients qui ont passé des commandes, (3) de commandes³⁰ ?

Aussi étrange que cela paraisse, la réponse est *une commande*.

La règle relative à une jointure élémentaire basée sur l'égalité *identifiant/clé étrangère* est simple :

Considérons une table TA, d'identifiant primaire IA, et une table TB, de clé étrangère RA obligatoire, référençant TA (nous ignorons les autres colonnes).



Le résultat de l'évaluation de l'expression

```
select *
from   TA, TB
where  TA.IA = TB.RA
```

contient autant de lignes qu'il y en a dans la table TB. Autrement dit, chaque ligne du résultat de la jointure représente une ligne de TB. Le résultat d'une jointure de TA et TB correspond à la population de TB ou à un de ses sous-ensembles (s'il y a des conditions de sélection additionnelles).

En bref, le résultat d'une jointure représente des *entités de la table contenant la clé étrangère*.

A titre d'exemple supplémentaire, l'expression suivante désigne des lignes de DETAIL, et non pas des lignes de COMMANDE. En effet, il y a autant de lignes dans le résultat qu'il y en a dans la table DETAIL :

```
select COMMANDE.NCOM, DATECOM, NCLI
from   COMMANDE, DETAIL
where  COMMANDE.NCOM = DETAIL.NCOM
```

30. Pour répondre à ces questions, on ne s'arrêtera pas aux éléments de la clause `select`, dont le seul but est d'induire le lecteur en erreur !

L'interprétation d'une jointure multiple se fait de manière itérative : on évalue une première jointure, dont le résultat est joint à la table suivante³¹, et ainsi de suite. Selon cette règle, la requête suivante décrit des lignes de DETAIL.

```
select D.NCOM,D.NPRO,LOCALITE,LIBELLE
from   CLIENT CLI,COMMANDE COM,DETAIL D,PRODUIT P
where  CLI.NCLI = COM.NCLI
and    COM.NCOM = D.NCOM
and    D.NPRO = P.NPRO
```

Dans les situations de jointure multiples, la règle d'interprétation évoquée ne fonctionne que si chaque table citée dans la jointure n'est référencée que par une seule table de cette même jointure. Intuitivement, si les arcs des clés étrangères sont dessinés avec une orientation de bas en haut, le graphe des tables intervenant dans la jointure est un arbre dont la racine est en bas, et qui fournit des informations sur les entités représentées par cette racine. Pour la requête précédente, ce graphe est celui de la figure 3.5. Cette règle est aussi applicable à une jointure `PRODUIT * COMPOSITION * PRODUIT` (figure 5.5) qui associerait ses composants à chaque produit, et qui représenterait des *compositions*.

Intuitivement toujours, la jointure est formée de fragments qui obéissent aux règles de la figure 5.9, où un rectangle représente toute citation d'une table dans la clause `from` et un arc une clé étrangère servant à définir une jointure dans la clause `where` (les fragments du type 2 et 3 peuvent comprendre plus de deux clés étrangères)

En particulier, elle ne fonctionne pas dans le cas 4, où une jointure est définie entre les tables A, B et C, telles que A est référencée par B et C. Cette jointure représente des entités nouvelles, résultant du croisement des entités de B et des entités de C via A³². Bien que de telles jointures soient relativement rares, cette restriction valait d'être notée.

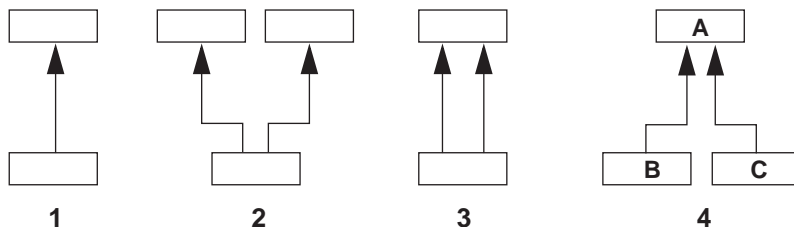


Figure 5.9 - Une jointure fournit des informations sur la racine de son graphe si celui-ci est constitué de fragments des types 1, 2 et 3, mais pas du type 4

31. L'ordre d'évaluation de jointures multiples est indifférent.

32. Un exemple concret : la table ENTREPRISE (A) est référencée par les tables FOURNISSEUR (B) et CLIENT (C). La jointure de ces trois tables représente tous les couples <fournisseur,client> tels que fournisseur et client dépendent de la même entreprise.

b) Identifiant d'une jointure

Ce que nous venons de discuter conduit à une première règle. L'identifiant du résultat de la jointure :

```
select *
from   TA, TB
where  TA.IA = TB.RA
```

est constitué des *colonnes de l'identifiant primaire de TB (soit IB)*. Si certaines colonnes de l'identifiant primaire de TB ne sont pas reprises dans la clause `select` (et qu'aucun autre identifiant de TB n'est repris), le résultat n'a en principe³³ pas d'identifiant, comme on peut l'observer dans la requête suivante, dérivée de celle de la section précédente :

```
select LOCALITE,LIBELLE
from   CLIENT CLI,COMMANDE COM,DETAIL D,PRODUIT P
where  CLI.NCLI = COM.NCLI
and    COM.NCOM = D.NCOM
and    D.NPRO = P.NPRO
```

Cette requête représente les détails de commande, mais pour chacun d'eux on ne retient que la localité du client de la commande, ainsi que le libellé de son produit, ce couple de valeurs n'étant évidemment pas identifiant.

L'ajout du modifieur `distinct` dans la clause `select` permet de reconstituer un identifiant. Cependant, le résultat serait tout autre : chaque ligne $\langle v, l \rangle$ de ce résultat représenterait alors le fait que dans la localité v , on a commandé au moins une fois un produit de libellé l .

5.5 EXTRACTION DE DONNÉES GROUPÉES

Les requêtes examinées dans les sections précédentes³⁴ produisent des lignes qui sont généralement en correspondance *une pour une* avec les lignes d'une table de la clause `from`. Nous allons examiner comment il est possible d'extraire d'une table, ou d'une jointure, des informations sur des concepts *latents* dans ces tables.

5.5.1 Notion de groupe de lignes

Considérons la table CLIENT. Il est permis d'y percevoir, virtuellement du moins, des groupes de clients selon leur localité, ou selon leur catégorie, ou encore selon leur nom³⁵. La requête suivante donne, pour chaque groupe de clients *classés* ou

33. La règle complète est un peu plus complexe. En particulier, si la clé étrangère, même si elle n'est pas (entièrement) reprise dans la clause `select`, forme un identifiant de TB, et si la clause `select` reprend un identifiant de TA, alors cet identifiant est aussi celui du résultat de la jointure.

34. Sauf cas particuliers tels que l'usage de fonctions agrégatives ou du modifieur `distinct`.

regroupés par localité, le nom de celle-ci, le nombre de clients dans le groupe et le montant moyen des comptes des clients du groupe.

```
select LOCALITE,
       count(*) as NOMBRE_CLIENTS,
       avg(COMPTE) as MOYENNE_COMPTE
from   CLIENT
group by LOCALITE
```

Le résultat compte autant de lignes qu'il y a de groupes ainsi constitués, et donc autant qu'il y a de localités distinctes dans CLIENT :

LOCALITE	NOMBRE_CLIENTS	MOYENNE_COMPTE
Poitiers	3	533.33
Namur	4	-2520.00
Geneve	1	0.00
Lille	1	720.00
Toulouse	5	-2530.00
Paris	1	0.00
Bruxelles	1	0.00

Nous avons donc *interrogé* les localités plutôt que les clients. Le mécanisme qui extrait les informations sur les localités à partir des informations sur les clients est invoqué par la clause `group by`.

On ne peut trouver dans la clause `select` que des éléments (colonnes, expression de calcul, fonction agrégatives) définissant une seule valeur par groupe. On n'aurait donc pu y spécifier ni `NOM` ni `CAT`. Intuitivement, on ne peut demander à un groupe que des informations communes à tous ses membres : des composants du critère de groupement, des fonctions agrégatives et bien sûr des constantes³⁵.

5.5.2 Sélection de groupes et sélection de lignes

Des conditions de sélection peuvent être imposées aux groupes à sélectionner. Elles seront exprimées dans une clause `having`, ce qui évite toute confusion avec la clause `where` qui, elle, s'applique aux lignes. Dans la requête ci-dessous, on ne retient par exemple que les groupes d'au moins trois clients :

```
select LOCALITE, count(*), avg(COMPTE)
from   CLIENT
group by LOCALITE
having count (*) >= 3
```

ce qui donne :

35. Les grouper selon leur numéro `NCLI` constituerait des groupes d'une seule personne, ce qui n'aurait guère d'intérêt.

36. Lorsqu'on interroge un groupe, *tous ses membres doivent répondre d'une même voix*.

LOCALITE	count(*)	avg(COMPTE)
Poitiers	3	533.33
Namur	4	-2520.00
Toulouse	5	-2530.00

Dans cette requête, la condition `having` peut porter sur les éléments cités dans la clause `select`, mais aussi sur toute autre fonction d'agrégation calculable sur chaque groupe.

La requête suivante traite les lignes de `COMMANDE` en les regroupant par client :

```
select NCLI, count(*)
from   COMMANDE
group by NCLI
```

On a donc *reconstitué* des clients à partir des commandes. On ne retient ensuite que les groupes d'au moins deux commandes :

```
select NCLI, count(*)
from   COMMANDE
group by NCLI
having count(*) >= 2
```

Dans le résultat on ne considère ensuite, avant groupement, que les commandes spécifiant le produit PA45. En clair, on s'intéresse aux clients qui ont commandé au moins deux fois le produit PA45 (soit ici l'unique client C400).

```
select NCLI, count(*)
from   COMMANDE
where  NCOM in ( select NCOM
                  from   DETAIL
                  where  NPRO = 'PA45')
group by NCLI
having count(*) >= 2
```

5.5.3 Groupes et jointures

Il est possible d'obtenir en outre la quantité totale de ce produit PA45 que chaque client a commandée. Les données à afficher appartenant à plusieurs tables (`NCLI` et somme des `QCOM`), il nous faut opérer un groupement sur le résultat de la jointure de `COMMANDE` et `DETAIL`. On observera que la sous-requête est remplacée par une condition de jointure³⁷.

```
select M.NCLI, count(*) , sum(QCOM)
from   COMMANDE M, DETAIL D
where  M.NCOM = D.NCOM
```

37. Pourquoi ?

```

and      NPRO = 'PA45'
group by M.NCLI
having count(*) >= 2

```

Selon le même principe de groupement sur une jointure, la requête suivante établit pour chaque client de Poitiers le montant total de ses commandes.

```

select 'Montant dû par ',C.NCLI,' = ',sum(QCOM*PRIX)
from   CLIENT C, COMMANDE M, DETAIL D, PRODUIT P
where  LOCALITE = 'Poitiers'
and    M.NCLI = C.NCLI
and    M.NCOM = D.NCOM
and    D.NPRO = P.NPRO
group by M.NCLI

```

La clause `group by NCLI` devrait théoriquement autoriser la présence de `NOM` ou `CAT` dans la clause `select`, puisque par construction à une même valeur de `NCLI` correspondent toujours la même valeur de `NOM` et la même valeur de `CAT`. Il n'en est malheureusement rien, la règle énoncée ci-dessus ne souffrant aucune exception, aussi logique soit-elle. Si nous voulons voir apparaître ces données, alors nous devons écrire `group by M.NCLI, NOM, CAT`. Pas très élégant, mais efficace !

La requête suivante calcule la quantité qui resterait en stock pour chaque produit si on déduisait les quantités commandées. On observera la présence (en apparence inutile) de `QSTOCK` dans le critère de groupement réclamée par son apparition dans la clause `select` en dehors d'une fonction d'agrégation. On observera aussi que seuls les produits effectivement commandés sont repris.

```

select P.NPRO, QSTOCK - sum(D.QCOM) as SOLDE
from   DETAIL D, PRODUIT P
where  D.NPRO = P.NPRO
group by P.NPRO, QSTOCK

```

NPRO	SOLDE
CS262	-15
CS464	-135
PA45	535
PA60	-1
PH222	690
PS222	620

5.5.4 Composition du critère de groupement

Nous venons de le voir, le critère de groupement peut inclure plusieurs noms de colonne. L'exemple supplémentaire ci-dessous calcule pour chaque groupe de mêmes valeurs de `LOCALITE` et `NPRO` le montant total commandé.

```

select LOCALITE,P.NPRO,sum(QCOM*PRIX) as Montant
from   CLIENT C, COMMANDE M, DETAIL D, PRODUIT P
where  M.NCLI = C.NCLI

```

```
and      M.NCOM = D.NCOM
and      D.NPRO = P.NPRO
group by LOCALITE, P.NPRO
```

L'ordre des composants est indifférent : group by P.NPRO, LOCALITE donnerait le même résultat.

Le critère de groupement peut aussi inclure une expression de calcul quelconque. La requête suivante constitue des groupes de clients selon la première lettre de leur valeur de CAT.

```
select substring(CAT from 1 for 1) as CAT, count(*) as N
from   CLIENT
group by substring(CAT from 1 for 1)
```

CAT	N
	2
B	8
C	6

Il est évident que certains critères de groupement sont plus pertinents que d'autres. Par exemple, un groupement sur les valeurs COMPTE serait sans intérêt. En revanche, il serait très intéressant de regrouper les clients selon leurs valeurs de COMPTE par intervalles de 1.000, comme dans la requête ci-dessous.

```
select "de ", int(COMPTE/1000)*1000 as Min,
       " à ", int(COMPTE/1000)*1000 + 999 as Max,
       count(*) as N
from   CLIENT C
group by int(COMPTE/1000)
```

de	Min	à	Max	N
de	-9000	à	-8001	1
de	-5000	à	-4001	1
de	-4000	à	-3001	1
de	-3000	à	-2001	2
de	-2000	à	-1001	1
de	0	à	999	9
de	1000	à	1999	1

5.5.5 Attention aux groupements multi-niveaux

L'extraction de données groupées est à définir avec précaution en présence de jointures. Cherchons par exemple à déterminer, *pour chaque localité, la somme des comptes des clients et le nombre de commandes*. On serait tenté d'écrire :

```
select LOCALITE, sum(COMPTE), count(*)
from   CLIENT C, COMMANDE M
where  C.NCLI = M.NCLI
group by LOCALITE
```

ce qui nous donnerait :

LOCALITE	sum(COMPTE)	count(*)
Lille	720	1
Namur	-4580.00	1
Poitiers	1050.00	3
Toulouse	-8700.00	2

Ce résultat, en apparence correct, **est pourtant erroné** (indépendamment du fait que les clients sans commandes ne sont pas repris). En effet, le résultat de la jointure représente des commandes et non des clients. En particulier, le compte du client C400 est compté trois fois, pour un total de 1050 au lieu de 350. Le calcul de la somme des comptes se fait donc sur des ensembles de commandes et non de clients. Tout client qui possède plus d'une commande verra son compte intervenir plus d'une fois dans la somme. En revanche, le comptage des commandes par localité est correct. Il faudra, pour répondre à la question posée, procéder en deux étapes indépendantes.

Dans certains cas cependant, il est possible de citer des fonctions agrégatives à plusieurs niveaux, pour autant que *toutes les fonctions sommatives (sum et avg) s'adressent au niveau le plus bas des jointures*, les fonctions count, max et min pouvant s'appliquer à tous les niveaux. La requête ci-dessous illustre cette structure. Elle recherche, *pour chaque client, le nombre de commandes et le montant total de ces commandes*. La première information est relative au niveau COMMANDE tandis que la seconde dérive du niveau DETAIL. Pour obtenir le nombre exact de commandes, on utilisera le modifieur distinct dans la fonction count.

```
select M.NCLI, count(distinct M.NCOM), sum(QCOM*PRIX)
from   COMMANDE M, DETAIL D, PRODUIT P
where  M.NCOM = D.NCOM
and    D.NPRO = P.NPRO
group by M.NCLI
```

5.5.6 Peut-on éviter l'utilisation de données groupées ?

Il est possible d'éviter la clause group by lorsque le concept latent dans une table est explicitement représenté par une autre table, et que le regroupement ne sert qu'à la sélection. On recherche par exemple les produits dont on a commandé plus de 500 unités en 2005. La forme qui semble s'imposer est la suivante, qui extrait les produits comme concept latent de la table DETAIL, via la colonne NPRO :

```
select D.NPRO
from   DETAIL D, COMMANDE M
where  D.NCOM = M.NCOM
and    DATECOM like '%2005'
group by D.NPRO
having sum(QCOM) > 500
```


Cependant, en consultant directement la table `PRODUIT`, on peut exprimer la condition de sélection de manière plus naturelle :

```
select NPRO
from   PRODUIT P
where  (select sum(QCOM)
        from DETAIL
        where NPRO = P.NPRO
        and NCOM in (select NCOM from COMMANDE
                     where DATECOM like '%2005')) > 500
```

Toujours dans les situations où le concept interrogé est aussi décrit par une table, on peut éviter la structure du `group by` également lorsque des valeurs agrégées sont demandées dans la clause `select`, comme on le verra dans la section 6.3.1.

5.6 ORDRE DES LIGNES D'UN RÉSULTAT

Par construction, l'ordre des lignes d'une table est arbitraire. On ne peut donc supposer que les lignes sont stockées dans un ordre déterminé (celui de leur instant de création par exemple). Si elles semblent l'être un jour, rien ne garantit qu'elles le seront encore le lendemain. En principe, l'ordre des lignes du résultat d'une requête est aussi arbitraire. Il est cependant possible d'imposer un ordre de présentation spécifique par la clause `order by`.

Les lignes de la table résultant de la requête suivante vont apparaître classées (on dira *triées*) par valeurs croissantes de `LOCALITE`.

```
select NCLI, NOM, LOCALITE
from   CLIENT
where  CAT in ('C1','C2')
order by LOCALITE
```

On peut indiquer plusieurs critères de tri :

```
select *
from   CLIENT
order by LOCALITE, CAT
```

Les clients vont apparaître classés par localité, puis dans chaque localité, classés par catégorie. Par défaut, le classement se fait par ordre ascendant des valeurs. On peut également spécifier explicitement un ordre ascendant (`asc`) ou descendant (`desc`) :

```
select *
from   PRODUIT
where  LIBELLE like '%SAPIN%'
order by QSTOCK desc
```

Le critère de tri est constitué d'une ou plusieurs expressions, apparaissant ou non dans la clause `select`. Si une expression apparaît dans la cause `select`, elle sera spécifiée par son nom s'il s'agit d'une colonne, comme dans les exemples précé-

dents, ou par son alias en cas d'expression calculée, comme dans l'exemple ci-dessous, qui présente les localités par valeurs décroissantes de population de clients :

```
select LOCALITE, count(*) as POPULATION, sum(COMPTE)
from   CLIENT
group by LOCALITE
order by POPULATION desc
```

Certains SGBD n'admettent pas l'usage d'un alias de colonne dans la clause `order by`; on utilisera alors le numéro d'ordre de la colonne : `order by 2 desc`.

L'exemple suivant définit un ordre selon une expression qui n'apparaît pas dans la clause `select` :

```
select NCOM, NPRO, QCOM
from   DETAIL D, PRODUIT P
where  D.NPRO = P.NPRO
order by NCOM, QCOM*PRIX desc
```

L'ordre des composants du critère d'ordre n'est pas indifférent : `order by LOCALITE, CAT` donnera une autre séquence que `order by CAT, LOCALITE`.

5.7 INTERPRÉTATION D'UNE REQUÊTE

Certaines requêtes étant relativement complexes, on pourra, pour interpréter leur signification, considérer utilement la procédure d'évaluation suivante. Il s'agit d'une évaluation fictive, le SGBD utilisant généralement d'autres procédés plus efficaces pour construire le résultat.

Pour une requête monotable :

- on considère la table spécifiée dans la clause `from`;
- on sélectionne les lignes sur base de la clause `where`;
- on classe ces lignes en groupes comme spécifié dans la clause `group by`;
- on ne retient que les groupes qui vérifient la cause `having`;
- les lignes des groupes sont ordonnées selon la clause `order by` éventuelle;
- de chacune des lignes, on extrait les valeurs demandées dans la clause `select`.

Pour une requête multitable :

- on considère les tables spécifiées dans la clause `from`;
- on effectue la jointure de ces tables selon le critère de jointure de la clause `where`;
- on sélectionne les lignes de la jointure sur la base des autres conditions de la clause `where`;
- on classe ces lignes en groupes comme spécifié dans la clause `groupe by`;
- on ne retient que les groupes qui vérifient la cause `having`;
- les lignes des groupes sont ordonnées selon la clause `order by` éventuelle;
- de chacune des lignes, on extrait les valeurs demandées dans la clause `select`.

À titre d'exemple, on indique, pour la requête ci-dessous, l'ordre des actions d'interprétation qui conduisent à l'élaboration de la réponse.

```
7 : select NCLI, count(*), sum(QCOM)
1 : from   COMMANDE M, DETAIL D
2 : where  M.NCOM = D.NCOM
3 : and    NPRO = 'PA60'
4 : group by NCLI
5 : having count(*) >= 2
6 : order by NCLI
```

5.8 MODIFICATION DES DONNÉES

On présente les opérateurs qui permettent l'introduction (*insert*), la suppression (*delete*) et la modification (*update*) des données d'une base de données. On réexamine ensuite le comportement du SGBD lors de mises à jour en présence de contraintes référentielles.

5.8.1 Ajout de lignes

Si on désire ajouter une ligne constituée de valeurs nouvelles, on utilisera l'instruction *insert values*:

```
insert into DETAIL values ('30185','PA45',12)
```

L'ordre des valeurs est celui de la déclaration des colonnes lors de la création de la table. On peut aussi, en particulier lorsque certaines valeurs seulement sont introduites, préciser le nom et l'ordre des colonnes concernées :

```
insert into CLIENT (NCLI,NOM,ADRESSE,COMPTE,LOCALITE)
values ('C402','BERNIER','avenue de France, 28',
       -2500,'Lausanne')
```

Toute colonne non spécifiée, telle que CAT, prend la valeur *null*, ou la valeur par défaut si celle-ci a été déclarée comme propriété de la colonne. Toute colonne obligatoire (*not null*) doit recevoir une valeur, sauf si on lui a assigné une valeur par défaut (*default*) lors de sa déclaration.

Note

Chaque valeur peut être exprimée sous la forme d'une constante, comme illustré ci-dessus, mais plus généralement de toute expression dont l'évaluation donne une valeur : expression arithmétique ou requête SFW.

Il est possible d'insérer dans une table existante des données extraites d'une ou plusieurs tables. Ces données sont obtenues par une expression SFW attachée à l'instruction *insert*.

Dans la commande suivante, on ajoute à la table `CLIENT_TOULOUSE` (`NUMERO`, `NOM`, `ADRESSE`) les données extraites de `CLIENT` :

```
insert into CLIENT_TOULOUSE
select NCLI, NOM, ADRESSE
from   CLIENT
where  LOCALITE = 'Toulouse'
```

Cette table peut ensuite faire l'objet de manipulations comme toute autre table. On notera qu'à partir du moment où les données ont été insérées, la table `CLIENT_TOULOUSE` devient indépendante de la table `CLIENT`. En particulier, aucune mise à jour des lignes de `CLIENT` n'aura d'effet sur le contenu de la table `CLIENT_TOULOUSE`, ni inversement³⁸.

Dans tous les cas, les données insérées doivent respecter les contraintes d'intégrité (identifiants, intégrité référentielle, colonnes obligatoires) attachées à la table à laquelle les nouvelles lignes sont destinées. Notons cependant qu'il est fréquent que ces tables ne soient accompagnées d'aucune contrainte, en particulier lorsqu'elles ne font l'objet que de consultations après leur remplissage.

5.8.2 Suppression de lignes

L'ordre de suppression porte sur un sous-ensemble des lignes d'une table. Ces lignes sont désignées par une clause `where` dont le format est le même que celui de l'instruction `SFW` :

```
delete from CLIENT
where NCLI = 'K111'
```

Cet ordre supprime de la table `CLIENT` la ligne qui décrit le client numéro `K111`.

```
delete from DETAIL
where NPRO in (select NPRO
               from   PRODUIT
               where  QSTOCK <= 0)
```

Cette instruction supprime de la table `DETAIL` les lignes, quel qu'en soit le nombre, qui spécifient un produit en rupture de stock.

Après l'opération, la base de données doit être dans un état qui respecte toutes les contraintes d'intégrité auxquelles elle est soumise, et en particulier les contraintes référentielles. Nous y reviendrons.

38. On appelle parfois *instantané* (snapshot) ce type de table dérivée. Ces instantanés seront notamment utilisés pour distribuer l'information à partir d'une base de données centrale et pour constituer des systèmes d'aide à la décision (les *entrepôts de données* par exemple). Signalons que certains SGBD (Oracle par exemple), contrairement à la règle de base, peuvent sur demande gérer les instantanés, et propager les modifications des tables sources vers l'instantané.

5.8.3 Modification de lignes

Ici encore, la modification sera effectuée sur toutes les lignes qui vérifient une condition de sélection :

```
update CLIENT
set   ADRESSE = '29, av. de la Magne',
      LOCALITE = 'Niort'
where NCLI = 'F011'
```

Les nouvelles valeurs peuvent être obtenues par une expression arithmétique. La commande suivante enregistre une augmentation de prix de 5% pour les produits en sapin :

```
update PRODUIT
set   PRIX = PRIX * 1.05
where LIBELLE like '%SAPIN%'
```

Elles peuvent également provenir de la base de données elle-même. La requête ci-dessous déduit de la quantité en stock de chaque produit la somme des quantités actuellement en commande.

```
update PRODUIT P
set   QSTOCK = QSTOCK - (select sum(QCOM)
                        from   DETAIL
                        where  NPRO = P.NPRO)
where exists (select * from DETAIL where NPRO = P.NPRO)
```

On notera l'usage de l'alias de table *P* dans la clause *update*, qui permet de relier des détails au produit en cours de modification. La condition de sélection "*exists (select NPRO from DETAIL)*" est nécessaire pour éviter une destruction des données. En effet, pour un produit *P* qui n'a pas de détails :

- l'ensemble *DETAIL where NPRO = P.NPRO* est vide;
- l'expression *sum(QCOM)* appliquée à un ensemble vide renvoie *null*;
- l'expression *QSTOCK - null* vaut *null*;
- et donc la valeur de la colonne *QSTOCK* de *P* est détruite !

5.8.4 Mise à jour et contraintes référentielles

a) Principes

Une requête de modification du contenu de la base de données (*insert*, *delete*, *update*) ne sera exécutée que si le résultat respecte toutes les contraintes d'intégrité définies sur cette base. L'impact de ces contraintes sur le comportement des données en cas de mise à jour a été décrit dans la section 3.8, et est garanti par tous les SGBD relationnels. Le principe de base est qu'une opération dont l'exécution va laisser les données dans un état invalide est refusée. Cependant, comme nous l'avons déjà

précisé, le comportement du SGBD en cas de suppression ou de modification en présence d'une contrainte référentielle est plus nuancé.

Considérons par exemple la sous-structure constituée des tables CLIENT et COMMANDE.

```
create table CLIENT ( NCLI char(10) not null,
                    ...,
                    primary key (NCLI) )

create table COMMANDE ( NCOM char(12) not null,
                       NCLI char(10) not null,
                       ...,
                       primary key (NCOM),
                       foreign key (NCLI) references CLIENT)
```

Lors de la suppression d'une ligne de CLIENT, trois types de comportement sont possibles :

- **Blocage** : la suppression est refusée s'il existe une ou plusieurs lignes de COMMANDE dépendantes (c'est-à-dire dont COMMANDE.NCLI = CLIENT.NCLI). On définit ce comportement comme suit³⁹ :

```
create table COMMANDE ( NCOM char(12) not null,
                       NCLI char(10) not null,
                       ...,
                       primary key (NCOM),
                       foreign key (NCLI) references CLIENT
                                   on delete no action)
```

- **Propagation** : la suppression est acceptée, mais elle entraîne la suppression conjointe des lignes de COMMANDE dépendantes. On écrira alors :

```
create table COMMANDE ( NCOM char(12) not null,
                       NCLI char(10) not null,
                       ...,
                       primary key (NCOM),
                       foreign key (NCLI) references CLIENT
                                   on delete cascade)
```

- **Découplage** : la suppression est acceptée, mais les lignes dépendantes sont rendues indépendantes par l'attribution de la valeur null à leur colonne NCLI⁴⁰. Ce comportement est spécifié comme suit :

```
create table COMMANDE ( NCOM char(12) not null,
                       NCLI char(10),
                       ...,
                       primary key (NCOM),
```

39. Une variante est disponible dans certains SGBD : le mode **restrict**.

40. Il existe une variante **set default**, par laquelle la valeur par défaut se substitue à la valeur null.

```
foreign key (NCLI) references CLIENT
on delete set null)
```

La modification de la valeur de l'identifiant NCLI de CLIENT est aussi régie par des règles similaires (on update) : refus s'il existe des commandes dépendantes (no action), propagation par modification des clés étrangères pour les commandes dépendantes (cascade) et mise à null des clés étrangères (set null) :

```
create table COMMANDE ( NCOM char(12) not null,
                        NCLI char(10) not null,
                        ...,
primary key (NCOM),
foreign key (NCLI) references CLIENT
on delete no action
on update cascade)
```

b) Les hiérarchies de clés étrangères

La définition de ces modes comportementaux demande un soin particulier. En effet, le mode d'une clé étrangère peut influencer celui d'une autre clé étrangère. Considérons par exemple le schéma simplifié suivant

```
create table CLIENT ( NCLI char(10) not null,
primary key (NCLI) )

create table COMMANDE ( NCOM char(12) not null,
                        NCLI char(10) not null,
primary key (NCOM),
foreign key (NCLI) references CLIENT
on delete cascade)

create table DETAIL ( NCOM char(12) not null,
                     NPRO char(15) not null,
foreign key (NCOM) references COMMANDE
on delete no action)
```

Lors de la suppression d'une ligne de CLIENT, le SGBD tente de supprimer en cascade les lignes de COMMANDE qui en dépendent. Cependant, si une de ces lignes de COMMANDE est associée à une ligne de DETAIL, sa suppression est refusée (on delete no action), et donc la suppression de la ligne de CLIENT l'est également. De ce schéma, on déduit qu'on ne peut supprimer un client que s'il n'a aucune commande qui possède des détails.

Les modes comportementaux liés aux suppressions et aux mises à jour posent des problèmes complexes qui amènent généralement les SGBD à travailler en deux phases : dans un premier temps, les lignes à supprimer sont simplement marquées, puis les modifications sont effectivement réalisées.

5.8.5 Modification des structures de données

La modification du schéma d'une base de données implique le plus souvent des modifications de données. Par exemple, l'ajout d'une colonne à une table contenant des lignes est suivi de la modification de cette colonne pour chacune des lignes (mise à `null` ou à la valeur par défaut). Ou encore, la suppression d'une table est précédée de la suppression de chacune de ses lignes. Il est évident que ces opérations de modification doivent respecter les contraintes d'intégrité au moment de leur exécution. Nous donnerons quelques exemples de règles induites par la modification des données.

a) *Ajout d'une colonne*

Si la colonne est facultative, l'opération s'effectue sans contrainte. Si elle est obligatoire (`not null`), alors la table doit être vide ou la colonne doit être accompagnée d'une valeur par défaut (`default`).

b) *Suppression d'une colonne*

Cette colonne ne peut intervenir dans la composition d'un identifiant ou d'une clé étrangère. Si nécessaire, ces derniers doivent d'abord être supprimés.

c) *Suppression d'une table*

Si la table est référencée par une clé étrangère, alors une opération `delete` est appliquée à chacune de ses lignes avant sa suppression. Cette suppression peut donc être refusée si le `delete` mode de cette clé est `no action` et si la table référençante contient au moins ligne dont la clé étrangère a une valeur non `null`.

d) *Ajout d'un identifiant*

Si la table n'est pas vide, les lignes doivent respecter la contrainte d'unicité.

e) *Suppression d'un identifiant*

Cette suppression n'est pas soumise à des conditions sur les données. Cependant, cet identifiant ne doit pas être référencé par une clé étrangère.

f) *Ajout d'une clé étrangère*

Si la table n'est pas vide, les lignes doivent respecter la contrainte référentielle.

g) *Autres opérations*

En revanche, l'ajout d'une table, l'ajout et la suppression d'un index non identifiant ainsi que la suppression d'une clé étrangère ne sont pas soumis à conditions.

5.9 EXERCICES

Exprimer les requêtes ci-dessous en SQL (sauf mention contraire). Les énoncés sont classés par degré de difficulté croissante, numéroté de 1 à 7.

5.9.1 Énoncés de type 1

- 5.1 Afficher les caractéristiques des produits (c'est-à-dire, pour chaque produit, afficher ses caractéristiques).
- 5.2 Afficher la liste des localités dans lesquelles il existe au moins un client.
- 5.3 Afficher le numéro, le nom et la localité des clients de catégorie C1 n'habitant pas à Toulouse.
- 5.4 Afficher les caractéristiques des produits en acier.
- 5.5 Donner le numéro, le nom et le compte des clients de Poitiers et de Bruxelles dont le compte est positif.
- 5.6 Dessiner l'organigramme des personnes selon le contenu de la table 5.3. De même, dessiner le graphe de composition de la nomenclature de la figure 5.6.

5.9.2 Énoncés de type 2

- 5.7 Quelles catégories de clients trouve-t-on à Toulouse ?
- 5.8 Afficher le numéro, le nom et la localité des clients dont le nom précède alphabétiquement la localité où ils résident.
- 5.9 Afficher le total, le minimum, la moyenne et le maximum des comptes des clients (compte non tenu des commandes actuelles).
- 5.10 Afficher les numéros des clients qui commandent le produit de numéro 'CS464'.
- 5.11 Afficher les localités des clients qui commandent le produit de numéro 'CS464'.
- 5.12 Donner le numéro et le nom des clients de Namur qui n'ont pas passé de commandes.
- 5.13 Quels sont les produits en sapin qui font l'objet d'une commande ?

- 5.14 Rechercher les clients qui, s'ils ont un compte négatif, ont passé au moins une commande. Quelle est la différence avec la requête suivante : *rechercher les clients qui ont un compte négatif et ont passé au moins une commande* ?
Suggestion. Il s'agit d'une relation d'implication.
- 5.15 On considère comme ci-dessus les clients qui, s'ils ont un compte négatif, ont passé au moins une commande. Rechercher les autres clients.
Suggestion. Il s'agit de la *négation* d'une relation d'implication.
- 5.16 Rechercher les commandes qui, si elles référencent des produits en sapin, référencent aussi des pointes en acier.
- 5.17 Ecrire les requêtes SQL qui recherchent les clients (on simplifiera si nécessaire) :
- habitant à Lille ou à Namur.
 - qui à la fois habitent à Lille et n'habitent pas à Namur.
 - qui habitent à Lille ou n'habitent pas à Namur.
 - qui n'habitent ni à Lille ni à Namur.
 - qui n'habitent pas à Lille ou qui n'habitent pas à Namur.
 - de catégorie C1 habitant à Namur.
 - de catégorie C1 ou habitant à Namur.
 - de catégorie C1 n'habitant pas à Namur.
 - qui n'ont pas été sélectionnés dans la question précédente.
 - qui soit sont de catégorie B1 ou C1, soit habitent à Lille ou à Namur (ou les deux conditions).
 - qui soit sont de catégorie B1 ou C1, soit habitent à Lille ou à Namur (mais pas les deux conditions).
 - qui sont de catégorie B1 ou C1, et qui habitent à Lille ou à Namur.
 - qui n'ont pas été sélectionnés dans la question précédente.

5.9.3 Énoncés de type 3

- 5.18 Calculer le montant de chaque détail de commande.
- 5.19 Afficher la valeur totale des stocks (compte non tenu des commandes actuelles).
- 5.20 Calculer le montant commandé des produits en sapin.
- 5.21 Afficher le total et la moyenne des comptes des clients, ainsi que le nombre de clients, selon chacune des classifications suivantes :
- par catégorie,

- par localité,
- par catégorie dans chaque localité.

5.22 Afficher le numéro et le libellé des produits en sapin :

- qui ne sont pas commandés,
- qui sont commandés à Toulouse,
- qui ne sont pas commandés à Toulouse
- qui ne sont commandés qu'à Toulouse,
- qui ne sont pas commandés qu'à Toulouse,
- qui sont commandés à Toulouse, mais aussi ailleurs.

5.23 Combien y a-t-il de commandes spécifiant un (ou plusieurs) produit(s) en acier ?

5.24 Dans combien de localités trouve-t-on des clients de catégorie C1 ?

5.25 Créer une table et y ranger les données suivantes relatives aux détails de commande : numéro et date de la commande, quantité commandée, numéro et prix du produit, montant du détail.

5.26 Annuler les comptes négatifs des clients de catégorie C1.

5.27 Compléter le fragment suivant de manière à former une requête valide

```
select CAT, NPRO, sum(QCOM*PRIX)
from ...
```

5.28 Rechercher les clients qui ont commandé le produit PA60 ou le produit PA45, mais pas les deux.

*Suggestion. Appliquer un *ou exclusif*.*

5.29 En se basant sur le schéma 5.5, écrire les requêtes SQL qui donnent :

- les matières premières (produits qui n'ont pas de composants);
- les produits finis (qui n'entrent dans la composition d'aucun autre);
- les produits semi-finis (tous les autres);
- le prix et poids unitaires d'un produit fini ou semi-fini dont tous les composants ont un poids et un prix unitaires.

5.9.4 Énoncés de type 4

5.30 Afficher le numéro et le nom des clients qui n'ont pas commandé de produits en sapin.

- 5.31 Rechercher les clients qui, s'ils ont commandé le produit PA60, en ont commandé plus de 500 unités au total.

Suggestion. Il s'agit d'une relation d'implication.

- 5.32 A la question : "*rechercher les localités dans lesquelles on n'a pas commandé de produit PA60*", quatre utilisateurs proposent les requêtes suivantes. Indiquer la (ou les) requêtes correctes, et interprétez les autres.

```
select distinct LOCALITE
from   CLIENT
where  NCLI in
        (select NCLI
         from   COMMANDE
         where  NCOM in
                (select NCOM
                 from   DETAIL
                 where  NPRO <> 'PA60'))
```

```
select distinct LOCALITE
from   CLIENT
where  NCLI in
        (select NCLI
         from   COMMANDE
         where  NCOM not in
                (select NCOM
                 from   DETAIL
                 where  NPRO = 'PA60'))
```

```
select distinct LOCALITE
from   CLIENT
where  NCLI not in
        (select NCLI
         from   COMMANDE
         where  NCOM in
                (select NCOM
                 from   DETAIL
                 where  NPRO = 'PA60'))
```

```
select distinct LOCALITE
from   CLIENT
where  LOCALITE not in
        (select LOCALITE
         from   CLIENT
         where  NCLI in
                (select NCLI
                 from   COMMANDE
                 where  NCOM in
                        (select NCOM
                         from   DETAIL
                         where  NPRO = 'PA60'))))
```

5.33 Que signifie la requête suivante ?

```
select *
from   COMMANDE
where  NCOM not in (select NCOM
                    from   DETAIL
                    where  NPRO <> 'PA60')
```

5.34 Dans quelles localités a-t-on commandé en décembre 2005 ?

5.35 Calculer le montant de chaque commande.

5.36 Calculer le montant dû par chaque client. Dans ce calcul, on ne prend en compte que le montant des commandes. Attention aux clients qui n'ont pas passé de commandes.

5.37 Calculer le montant dû par les clients de chaque localité. Dans ce calcul, on ne prend en compte que le montant des commandes. Attention aux localités dans lesquelles aucun client n'a passé de commandes.

5.38 Calculer, par jour, le total des montants des commandes.

5.39 On suppose qu'on n'a pas trouvé utile d'imposer un identifiant primaire sur la table PRODUIT. Il se peut donc que plusieurs lignes aient même valeur de la colonne NPRO, ce qui viole le principe d'unicité des valeurs de cette colonne.

- Écrire une requête qui recherche les valeurs de NPRO présentes en plus d'un exemplaire.
- Écrire une requête qui indique combien de valeurs de NPRO sont présentes en plus d'un exemplaire.
- Écrire une requête qui indique combien de lignes comportent une erreur d'unicité de NPRO.
- Écrire une requête qui, pour chaque valeur de NPRO présente dans la table, indique dans combien de lignes cette valeur est présente.
- Écrire une requête qui, pour chaque valeur de NPRO qui n'est pas unique, indique dans combien de lignes cette valeur est présente.
- Écrire une suite de requêtes qui crée une table contenant les numéros de NPRO qui ne sont pas uniques.

5.40 On suppose qu'on n'a pas trouvé utile de déclarer NCOM clé étrangère dans la table DETAIL. Il est donc possible que certaines lignes de DETAIL violent la contrainte d'intégrité référentielle portant sur cette colonne. Écrire une requête qui recherche les anomalies éventuelles.

5.41 Normalement, à toute commande doit être associé au moins un détail. Écrire une requête qui vérifie qu'il en est bien ainsi dans la base de données.

- 5.42 Afficher pour chaque localité, les libellés des produits qui y sont commandés.
- 5.43 Afficher par localité, et pour chaque catégorie dans celle-ci, le total des montants des commandes.
- 5.44 Quels sont les produits (numéro et libellé) qui n'ont pas été commandés en 2005 ?
- 5.45 Indiquer, pour chaque localité, les catégories de clients qui n'y sont pas représentées
Suggestion. Construire l'ensemble de tous les couples (LOCALITE, CAT) possibles et en retirer ceux qui existent dans la base. Attention aux valeurs null, qui ne doivent pas être prises en compte.
- 5.46 Produire (à l'écran) une table de couples $\langle X, Y \rangle$ de clients tels que X et Y habitent dans la même localité. On évitera de renseigner $\langle X, X \rangle$, mais aussi $\langle Y, X \rangle$ si $\langle X, Y \rangle$ est déjà repris.
Suggestion. Auto-jointure de CLIENT. On évitera les couples inverse en imposant un ordre sur les valeurs de NPRO (p.ex. $X < Y$).
- 5.47 En se basant sur le schéma 5.5, écrire une requête de mise à jour qui complète les prix et poids unitaires des produits finis ou semi-finis. Pour simplifier la procédure, on admet que cette requête soit exécutée autant de fois que nécessaire pour que tous les produits soient complétés.
- 5.48 En considérant le schéma de la figure 5.7, calculer pour chaque ville, le prix moyen de chaque produit.
- 5.49 Afficher pour chaque client, le nombre de commandes, le nombre de produits commandés et le nombre de détails. On se limite aux clients qui ont passé au moins une commande.
Suggestion : il s'agit d'une requête basée sur des groupements multi-niveaux.
- 5.50 Afficher, pour chaque localité et pour chaque catégorie, (1) le nombre de commandes passées par les clients de cette localité et de cette catégorie, (2) le montant total de ces commandes.

5.9.5 Énoncés de type 5

- 5.51 Calculer le nombre moyen de produits par commande. De même : le nombre moyen de commandes par client, par localité ou par catégorie.
Remarque. Il n'est pas possible de demander directement la moyenne d'une somme. *Suggestion :* construction et interrogation d'une vue ou calcul de la moyenne dans le `from`.

- 5.52 Quel est, pour chaque localité, le nombre moyen de commandes par client.
- 5.53 Ecrire une requête SQL qui donne, pour chaque catégorie de produit, le nombre de produits qui ont été commandés le 25-5-2005.
- 5.54 Donner pour chaque localité dans laquelle se trouve au moins un client de catégorie 'C2' la liste des produits en sapin qu'on y a commandés.
- 5.55 Donner pour chaque produit la liste des localités dans lesquelles ce produit est commandé en plus de 500 unités (= au total pour la localité).
- 5.56 Afficher, pour chaque localité, les produits qu'on y commande et qui sont aussi commandés dans au moins une autre localité.
Suggestion. Un produit est intéressant si le nombre de localités dans lesquelles il est commandé est supérieur et égal à 2.
- 5.57 Rechercher les clients qui ont commandé tous les produits.
Suggestion. Application du quantificateur *pour tout*. On recherche les clients tels qu'il n'existe pas de produits qui n'apparaissent pas dans les détails de leurs commandes.
- 5.58 Dans quelles localités a-t-on commandé tous les produits en acier (tous clients confondus) ?
- 5.59 Rechercher les produits qui ont été commandés par tous les clients.
- 5.60 Rechercher les localités dont aucun client n'a passé de commande.
- 5.61 Rechercher les localités dont tous les clients ont passé au moins une commande.
- 5.62 Rechercher les produits qui sont commandés dans toutes les localités.
- 5.63 Dans quelles localités peut-on trouver au moins un client qui a commandé tous les produits en sapin, et ce, pour un montant total, pour ces produits, dépassant 10.000 ?
- 5.64 Calculer, pour chaque localité, le nombre de catégories distinctes.
- 5.65 La section 5.8.1 suggère une comparaison entre un instantané et une vue de même définition. En effet, ces deux techniques pourraient être considérées comme équivalentes pour la formulation de requêtes. Établir une liste de critères de comparaison et l'appliquer aux deux techniques.
- 5.66 Mettre à jour les comptes des clients en en déduisant le montant des commandes en cours.

Suggestion. cfr mise à jour des quantités en stock des produits; attention aux clients qui n'ont pas commandé.

- 5.67 Mettre à jour les quantités en stock des produits en en déduisant les quantités commandées par les clients de catégorie B1 et C1.
- 5.68 On suppose qu'on dispose de deux tables `PRODUIT1` et `PRODUIT2` décrivant des produits de deux filiales distinctes, et de même structure que `PRODUIT`. Il est possible qu'un produit soit présent simultanément dans les deux filiales. Le même numéro de produit est repris alors dans les deux tables. On désire intégrer les deux dépôts. En conséquence, on fusionne les deux tables initiales en une troisième selon les règles suivantes :
- si un produit n'est présent que dans une seule filiale, sa ligne est reprise telle quelle,
 - si un produit est présent dans les deux filiales, on ne le décrit qu'une seule fois. Son libellé est celui de `PRODUIT1`, son prix est le minimum des deux valeurs et sa quantité en stock est la somme des deux valeurs.

Ecrire les requêtes d'intégration.

- 5.69 Soit une jointure de 2 ou plusieurs tables, dont la clause `select` spécifie certaines colonnes. On désire que le résultat ne comporte pas de lignes en double. Indiquer dans quelles conditions le modificateur `distinct` est inutile.
- 5.70 On considère une base de données constituée des deux tables suivantes : `R(A,B)` et `S(B,C)`. Evaluer l'ordre de grandeur du nombre de requêtes différentes qu'il est possible de formuler sur cette base de données.

5.9.6 Énoncés de type 6

- 5.71 La figure 9.24 (Voyages en train) représente un schéma qui est accompagné de suggestions de questions auxquelles une base de données traduisant ce schéma permettrait de répondre. On suppose que ce schéma est traduit en structure de tables (cf. exercice 11.3). Exprimer chacune de ces questions sous la forme de requêtes SQL.
- 5.72 On désire réaliser le couplage deux à deux de certaines localités où résident des clients. À cette fin, on désire construire des listes de couples candidats. On se limite cependant aux couples de localités dans lesquelles on achète au moins un même produit.
- 5.73 Même question que 5.72 ci-dessus, mais on se limite aux couples de localités dans lesquelles on achète exactement les mêmes produits.
- 5.74 Pour chaque produit, indiquer la ville dans laquelle la quantité totale commandée est minimale, et celle dans laquelle cette quantité est maximale.

5.9.7 Énoncé de type 7

5.75 Rédiger un script (suite de requêtes) SQL qui réalise en fin de mois les opérations de gestion régies par les règles suivantes :

- pour chaque client, on calcule le total des montants des commandes du mois écoulé (= les commandes d'un mois et d'une année déterminés);
- si ce montant est supérieur à celui du mois précédent, on applique une réduction de 5%;
- ensuite, si le client est le seul de sa localité, on applique une réduction supplémentaire de 5%;
- on range dans une table les informations nécessaires à l'édition des factures du mois;
- on met à jour le compte des clients;
- on met à jour le niveau de stock (QSTOCK) des produits commandés;
- pour chaque produit dont le niveau de stock a été mis à jour, et dont le niveau est inférieur ou égal à 0, on prépare dans une table adéquate les informations de réapprovisionnement.

Chapitre 6

SQL avancé

Ce chapitre est destiné au lecteur désireux d'aller plus avant dans la maîtrise des fonctions des bases de données et du langage SQL. On y décrit les fonctions de contrôle d'accès et les vues. La section suivante est consacrée à des formes plus puissantes de la requête SFW. On examine des mécanismes avancés tels que les prédicats, les procédures SQL et les déclencheurs. On donne ensuite une brève introduction aux concepts de catalogue et d'interface SQL avec les programmes d'application. La question de l'information incomplète clôture le chapitre.

6.1 LE CONTRÔLE D'ACCÈS

Tous les utilisateurs ne sont pas habilités à consulter et modifier toutes les données d'une base. SQL offre des fonctions de réglementation des droits d'accès aux données sous la forme de *privileges*.

a) *Principe*

Un privilège est l'autorisation qui est accordée à un utilisateur d'effectuer une opération sur un objet. Un privilège concerne donc, et doit spécifier, une opération, un objet (ou ressource) de la base de données ou de son environnement, l'utilisateur qui accorde le privilège (c'est celui qui exécute la requête de création ou retrait du privilège) et celui qui le reçoit¹. On étudie ici les commandes SQL qui permettent de créer et supprimer des privilèges.

b) Privilèges sur une table ou une vue

Les principales opérations admises sur le contenu des tables sont les suivantes :

- select : extraction de données,
- insert : ajout de lignes,
- delete : suppression de lignes,
- update : modification des valeurs de certaines colonnes,

Les objets concernés sont des tables ou des vues. Rappelons que certaines opérations de modification ne sont pas valides pour certains formats de vues.

Les utilisateurs sont désignés chacun par un nom. La commande suivante autorise les utilisateurs de nom P_MERCIER et S_FINANCIERS à consulter le contenu de la table PRODUIT et à modifier les valeurs de QSTOCK et PRIX :

```
grant select, update(QSTOCK, PRIX)
on    PRODUIT
to    P_MERCIER, S_FINANCIERS
```

Il est aussi possible de permettre au destinataire d'un privilège de transmettre celui-ci à d'autres utilisateurs (par une commande `grant`). Tel est le cas de la commande suivante, qui en outre permet toutes les opérations possibles sur la table CLIENT :

```
grant all privileges
on    CLIENT
to    P_MERCIER, S_FINANCIERS
with grant option
```

Un privilège peut être accordé à tous les utilisateurs :

```
grant select
on    COM_COMPLETE
to    public
```

c) Privilèges sur un programme

Ce privilège permet à un utilisateur d'exécuter un programme d'application ou une procédure qui manipule le contenu d'une base de données. Le propriétaire doit être autorisé à transmettre certains de ses privilèges, qui lui ont donc été accordés avec `grant option`. Quelques exemples :

```
grant run
on    SUP_DETAIL
to    public

grant run
```

1. En toute généralité, il faudrait ajouter à cette description le contexte spatio-temporel, qui permettrait, par exemple, de préciser que le privilège est actif pendant les jours ouvrables, jusqu'à une date limite, et à partir d'un terminal spécifique. Ceci n'est en principe pas pris en charge par SQL pour l'instant. Voir cependant l'exercice 6.1.

```
on      COMPTA01
to      S_FINANCIERS
with    grant option
```

d) Autres privilèges

Il existe d'autres privilèges liés notamment à l'administration de la base de données, en particulier à la définition et à la modification de structures de données : `create table`, `alter table`, `drop index`.

e) Suppression des privilèges

L'ordre `revoke` permet de retirer un privilège préalablement accordé :

```
revoke update (PRIX)
on      PRODUIT
to      P_MERCIER

revoke run
on      COMPTA01
from    P_MERCIER
```

L'effet de cette opération peut être délicat à interpréter. Par exemple, lorsqu'un même privilège a été accordé à un utilisateur U par plusieurs utilisateurs indépendants, alors U dispose de ce privilège tant que chacun de ces utilisateurs ne le lui aura pas retiré.

Il est possible de retirer la faculté de transmettre un privilège par une commande telle que :

```
revoke grant option for update (COMPTE)
on CLIENT
from P_MERCIER
```

Que se passe-t-il si P_MERCIER avait déjà transmis ce privilège ? Deux stratégies sont proposées : ou bien P_MERCIER conserve cette faculté de transmission (mode `restrict`) ou bien les privilèges qu'il a transmis sont retirés (mode `cascade`).

On trouvera dans [Melton, 1999] une présentation intuitive des principales règles de propagation des privilèges et de leur retrait.

6.2 LES VUES SQL

Les données sont en principe perçues selon leur organisation en tables et colonnes telles qu'elles ont été définies jusqu'ici. Il est cependant possible d'offrir aux utilisateurs une présentation différente de ces mêmes données *via* la notion de **vue** relationnelle.

6.2.1 Principe et objectif des vues

Une vue correspond à une table *virtuelle* dont seule la définition, sous la forme d'une requête SFW, est stockée et non le contenu. Une requête SQL peut extraire des données de tables réelles (stockées dans la base) ou virtuelles (vues). Il est même possible sous certaines conditions de modifier les données d'une vue (c'est-à-dire d'ajouter, supprimer, modifier des lignes). Dans ce cas le SGBD répercute sur les données réelles les modifications demandées. D'une manière générale on considérera qu'une vue peut être utilisée comme une table réelle.

6.2.2 Définition et utilisation d'une vue

Une vue est définie par une requête SQL qui spécifie son nom, celui de ses colonnes et la requête SFW qui calcule ses lignes² :

```
create view COM_COMPLETE (NCOM, NCLI, NOMCLI, LOC, DATECOM)
as select NCOM, COM.NCLI, NOM, LOCALITE, DATECOM
   from CLIENT CLI, COMMANDE COM
   where COM.NCLI = CLI.NCLI
```

La vue COM_COMPLETE peut être utilisée comme une table ordinaire. Les requêtes suivantes sont dès lors tout à fait valides :

```
select NOMCLI, NCOM, DATECOM
   from COM_COMPLETE
  where LOC = 'Toulouse'

select NPRO
   from DETAIL
  where NCOM in ( select NCOM
                  from COM_COMPLETE
                  where LOC = 'Toulouse' )

select LOC, count(*)
   from COM_COMPLETE CC, DETAIL D
  where CC.NCOM = D.NCOM
 group by LOC
```

Lorsqu'une requête utilise une vue, le SGBD reformule cette requête en y remplaçant les éléments de la vue par leur définition, puis l'exécute³. C'est ainsi que la première requête est d'abord reformulée comme suit, avant d'être exécutée :

```
select NOM, NCOM, DATECOM
   from CLIENT CLI, COMMANDE COM
  where COM.NCLI = CLI.NCLI
 and    LOC = 'Toulouse'
```

2. La notion de vue n'existe pas en MS Access, mais le nom d'un objet requête de type SFW peut apparaître dans la clause *from* d'une autre requête.

3. La réalité est un peu plus complexe, mais ce modèle est suffisant pour nos besoins.

Une vue peut être supprimée par l'instruction DROP :

```
drop view COM_COMPLETE
```

Cette opération ne supprime pas les données décrites par COM_COMPLETE. Elle interdit simplement d'y accéder selon cette vue. Seules les instructions `drop table` et `delete` permettent de supprimer des données.

6.2.3 Les vues comme interface pour des besoins particuliers

Le but d'une vue est avant tout d'offrir à un utilisateur de la base de données une présentation des données qui soit adaptée à ses besoins, et qui lui évite, d'une part, la complexité d'une base de données dont seules quelques données lui sont utiles, et d'autre part, de devoir rédiger les requêtes complexes correspondant à ses besoins.

La vue ci-dessous pourrait convenir à un analyste marketing qui étudie la répartition géographique des commandes de produits :

```
create view HABITUDE_ACHAT (LOCALITE, NPRO, VOLUME)
as  select  LOCALITE, P.NPRO, sum(QCOM*PRIX)
    from    CLIENT CLI, COMMANDE COM, DETAIL D, PRODUIT P
    where   COM.NCLI = CLI.NCLI
    and     D.NCOM = COM.NCOM
    and     P.NPRO = D.NPRO
    group by LOCALITE, P.NPRO
```

Note

Il serait intéressant de comparer les domaines d'application d'un *instantané* tel que CLIENT_TOULOUSE (section 5.8.1) avec une vue de même définition.

6.2.4 Les vues comme mécanisme de contrôle d'accès

Dans une entreprise, il n'est pas envisageable que tout le monde puisse faire n'importe quoi sur n'importe quelle donnée. C'est le rôle du contrôle d'accès que de régenter les opérations sur les données (section 6.1). Etant donné une classe d'utilisateurs, on détermine à quelles données ceux-ci ont accès, et on définit les vues qui ne reprennent que ces données. Ensuite, on accorde à ces utilisateurs l'autorisation d'utiliser ces vues, mais on leur interdit l'accès aux tables de base. Un employé chargé de faire des études sur les habitudes d'achat des clients selon différents critères, ne doit pas avoir accès aux données personnelles de ces clients. On lui donnera donc l'autorisation d'accéder à la vue suivante, mais pas à la table CLIENT elle-même :

```
create view ANALYSE (LOCALITE, CAT, DATE, NPRO, QCOM) as
select  LOCALITE, CAT, DATECOM, NPRO, QCOM
    from  CLIENT C, COMMANDE M, DETAIL D
    where C.NCLI = M.NCLI and M.NCOM = D.NCOM
```

6.2.5 Les vues comme mécanisme d'évolution de la base de données

Les vues permettent de protéger un utilisateur de l'effet de modifications de la structure de la base de données. Dans ce dernier cas, une modification de la requête définissant la vue permet d'offrir à l'utilisateur une perception inchangée des données.

Imaginons que la table `CLIENT` soit désormais décomposée en deux nouvelles tables, l'une qui reprend les données signalétiques `SIG_CLIENT(NCLI, NOM, ADRESSE, LOCALITE)` et la seconde partie qui ne reprend que les données commerciales `COM_CLIENT(NCLI, CAT, COMPTE)`. Par une vue définie comme la jointure de ces deux tables, on reconstitue la table `CLIENT` d'origine, certes désormais virtuelle, mais tout-à-fait opérationnelle.

6.2.6 Les vues comme aide à l'expression de requêtes complexes

Les vues permettent également l'expression de requêtes complexes difficiles ou impossibles à rédiger directement sous la forme SFW en SQL2. Par exemple, le calcul de la *valeur globale des stocks, déduction faite des quantités commandées*, s'obtiendra comme suit :

```
create view VAL_STOCK(STOCK, VALEUR) as
select P.NPRO, (QSTOCK - sum(D.QCOM)) * PRIX
from   DETAIL D, PRODUIT P
where  D.NPRO = P.NPRO
group by P.NPRO, QSTOCK, PRIX

select sum(VALEUR)
from   VAL_STOCK
```

6.2.7 Mise à jour des données via une vue

On sait qu'une vue est considérée comme une table de base pour ce qui concerne les requêtes d'extraction. Il en ira de même en ce qui concerne les requêtes de modification, pour autant que le SGBD soit capable de propager les modifications vers les tables de base. En effet, certaines vues ne reprennent pas les informations qui permettraient d'identifier les lignes des tables de base à modifier : que signifierait par exemple la modification de `LOCALITE` d'une ligne de la vue `HABITUDE_ACHAT` construite à la section 6.2.2 ? On conçoit aisément qu'on ne puisse supprimer une ligne d'une table dont l'identifiant primaire n'est pas repris dans la vue, ou insérer une ligne via une vue qui ne reprendrait pas certaines colonnes obligatoires⁴.

Les principales restrictions qui rendent une vue *modifiable* (c'est-à-dire pouvant faire l'objet d'insert, delete, update) sont les suivantes. Considérons d'abord une vue définie sur une seule table (ou vue). Les conditions de modifiabilité sont les suivantes :

4. A moins que celles-ci ne disposent d'une valeur par défaut, ou qu'un *trigger before* (Chapitre 5) ne garnisse ces colonnes.

- la vue ne contient pas les clauses `distinct` ni `group by`;
- la vue ne contient pas de fonctions agrégatives;
- la vue ne contient pas de sous-requête qui cite la table de la vue;
- la vue contient l'identifiant primaire de la table de base;
- si la vue est définie sur une autre vue, celle-ci doit être elle-même modifiable.

Si la vue est définie sur plusieurs tables, on observe en outre les contraintes suivantes :

- la vue n'est pas construite à partir des opérateurs ensemblistes `union`, `intersect` et `except`;
- si la vue est définie sur une jointure de deux ou plusieurs tables, seules celles dont l'identifiant primaire est également identifiant de la vue peuvent faire l'objet de modifications (voir section 5.4.9). Une vue définie sur la jointure de `CLIENT` et `COMMANDE` permet de supprimer et modifier une ligne de `COMMANDE` mais pas une ligne de `CLIENT`.

Une contrainte d'intégrité peut être associée à une vue grâce à la clause additionnelle `with check option`. Celle-ci rend active la condition définissant la vue (clause `where`), non seulement lors de l'extraction des données via cette vue, mais également en cas de modification. Si la clause `with check option` est spécifiée, alors les modifications sont soumises à la condition de la vue, de telle sorte que l'état final des données ne puisse violer celle-ci. Considérons la vue `CLI`, définie comme suit :

```
create view CLI (NCLI, NOM, ADRESSE, LOCALITE, CAT, COMPTE) as
select *
from   CLIENT
where  CAT is null or CAT in ('B1', 'B2', 'C1', 'C2')
with check option
```

La requête suivante sera rejetée car elle viole la condition, rendue active en modification :

```
insert into CLI
values ('B313', 'DURAND', 'place Monge', 'Mons', 'D7', 0)
```

6.3 EXTENSION DE LA STRUCTURE DES REQUÊTES SFW

6.3.1 Extension de la clause *select*

La clause `select` permet de spécifier les grandeurs à extraire de chaque ligne de la table fictive (ou de l'ensemble des lignes pour les fonctions agrégatives) définie par les clauses `from/group-by/having`. Toute expression qui renvoie une valeur pour chaque ligne est valide : nom de colonne, constante, expression arithmétique, de caractères ou de date. Il est aussi possible d'y inclure une requête SFW pourvu qu'elle renvoie une seule valeur :

```
select NCOM, (select sum(QCOM*PRIX)
              from   DETAIL D, PRODUIT P
              where  D.NPRO = P.NPRO
```

```

                        and      D.NCOM = M.NCOM) as MONTANT
from      COMMANDE M
where     MONTANT > 1000

```

6.3.2 Extension de la clause *from*

D'une manière générale, la clause *from* permet de mentionner les tables desquelles des données élémentaires ou calculées sont extraites (clause *select*). Jusqu'ici, on n'y mentionnait que le nom de tables de base ou de vues. Il est cependant possible d'y spécifier des tables dérivées, résultant d'opérateurs produisant des tables. Nous en examinerons trois : les opérateurs ensemblistes, la requête SFW elle-même et les opérateurs de jointure.

a) Les expressions ensemblistes

Toute expression consistant en une *union*, une *intersection* ou une *différence* de tables, qu'elles soient de base ou dérivées, peut être utilisée dans la clause *from*. Les règles de formation de telles expressions sont celles qui ont été brièvement décrites en 5.4.4. Considérant les tables *BON_CLIENT* (décrivant un sous-ensemble des clients) et *PROSPECT* (reprenant les clients potentiels), de même schéma que *CLIENT*, on pourra écrire⁵ :

```

select NCLI, NOM
from ((select NCLI, NOM, LOCALITE from CLIENT)
      except
      (select NCLI, NOM, LOCALITE from BON_CLIENT)
      union
      (select NCLI, NOM, LOCALITE from PROSPECT))
where LOCALITE = 'Poitiers'

```

b) Les requêtes SFW

Rappelons que l'évaluation d'une requête SFW a pour résultat une table d'une ou plusieurs colonnes, éventuellement vide. Il est permis d'utiliser cette table comme source de lignes pour une autre requête SFW sans qu'il soit nécessaire de stocker la table intermédiaire par un "insert into"⁶. La requête suivante calcule la moyenne des montants des commandes⁷ :

```

select avg(MONTANT)
from (select NCOM, sum(QCOM*PRIX) as MONTANT
      from   DETAIL D, PRODUIT P
      where  D.NPRO = P.NPRO
      group by NCOM)

```

5. La forme intuitive *from CLIENT except BON_CLIENT ...* n'est pas valide. Cependant la forme équivalente *from table CLIENT except table BON_CLIENT ...* est autorisée.

6. La section 6.2.6 propose une autre technique, basée sur la définition d'une vue intermédiaire.

7. On ne peut pas écrire : *avg(sum(QCOM*PRIX))*.

Combinant cette possibilité avec les opérateurs ensemblistes, on peut aussi écrire, pour afficher la quantité totale commandée de chaque produit :

```
select NPRO, TOTAL_QTE
from ( (select NPRO, sum(QCOM)
      from   PRODUIT P, DETAIL D
      where  P.NPRO=D.NPRO
      group by NPRO)
      union
      (select NPRO, 0
      from   PRODUIT
      where  NPRO not in (select NPRO from DETAIL))
      )
      as DP(NPRO,TOTAL_QTE)
where TOTAL_QTE < 1000
```

On notera la définition d'un alias de table (DP) accompagnée de celle des colonnes résultantes, ce qui permet de les utiliser dans les clauses `select` et `where`.

c) Les opérateurs de jointure

SQL propose des opérateurs explicites pour réaliser la jointure de deux ou plusieurs tables. Leur usage peut simplifier certaines requêtes, mais aussi rendre proprement illisibles certaines expressions, auquel cas on lui préférera celui de la formulation étudiée en 5.4.1, plus régulière. Le langage offre cinq formes d'opérateurs, largement redondantes, et dont l'utilité peut laisser perplexe.

► Cross join

La forme :

```
select *
from   CLIENT cross join COMMANDE
where  ...
```

est équivalente à l'expression classique du *produit relationnel* :

```
select *
from   CLIENT, COMMANDE
where  ...
```

► Natural join

Si les colonnes servant à la jointure portent le même nom, et que toutes les autres colonnes portent des noms différents dans les deux tables, on peut écrire :

```
select *
from   CLIENT natural join COMMANDE
where  ...
```

dont la formulation classique est :

```
select *
from   CLIENT C, COMMANDE M
where  C.NCLI = M.NCLI
and    ...
```

► Join on

La même requête pourrait s'écrire :

```
select *
from   CLIENT C join COMMANDE M
       on (C.NCLI = M.NCLI)
where  ...
```

Il est possible de construire des jointures multiples. Dans ce cas, elles sont évaluées de gauche à droite. Ainsi, l'expression :

```
select *
from   CLIENT C join COMMANDE M on (C.NCLI = M.NCLI)
       join DETAIL D on (M.NCOM = D.NCOM)
       join PRODUIT P on (D.NPRO = P.NPRO)
where  ...
```

est-elle équivalente à l'expression classique, plus régulière et plus lisible :

```
select *
from   CLIENT C, COMMANDE M, DETAIL D, PRODUIT P
where  C.NCLI = M.NCLI
and    M.NCOM = D.NCOM
and    D.NPRO = P.NPRO
```

Attention. Les jointures étant effectuées de gauche à droite, une condition ne peut citer que les composants (tables, alias, colonnes) des arguments de l'opérateur auquel elle est associée, et qui sont mentionnés à sa gauche. Ainsi, dans la formulation ci-dessus, la condition de la première jointure ne peut citer que les tables CLIENT et COMMANDE alors que la dernière peut citer les quatre tables. L'ordre de ces opérateurs est donc important, contrairement à la formulation classique.

► Join using

Le comité de normalisation propose une version plus compacte du join on :

```
select *
from   CLIENT join COMMANDE using (NCLI)
where  ...
```

En revanche, les colonnes ayant servi à définir cette jointure ne peuvent être préfixées dans les clauses select et where : on peut y citer NCLI, mais pas CLIENT.NCLI⁸. Il est permis d'utiliser le qualifieur inner, facultatif :

```
from CLIENT inner join COMMANDE using (NCLI)
```

► Outer join

La section 5.4.4 introduisait la possibilité d'ajouter au résultat d'une jointure les lignes *célibataires* d'une des tables, c'est-à-dire les lignes qui n'ont pas de correspondant dans l'autre table. Cette extension de la jointure s'appelle *jointure externe*, par opposition à la jointure classique, dite *interne*. Ainsi, la requête de la section 5.4.4, qu'on rappelle :

```
select NCOM, C.NCLI, DATECOM, NOM, LOCALITE
from COMMANDE M, CLIENT C
where M.NCLI = C.NCLI
union
select null, NCLI, null, NOM, ADRESSE
from CLIENT
where NCLI not in (select NCLI from COMMANDE)
```

peut être écrite de manière plus compacte et plus claire sous la forme :

```
select NCOM, C.NCLI, DATECOM, NOM, LOCALITE
from COMMANDE M right outer join CLIENT C
on (M.NCLI = C.NCLI)
```

Le terme **right** outer join indique qu'on inclut les lignes célibataires de la table de droite (CLIENT) dans le résultat. Il existe une version **left** outer join qui préserve les lignes célibataires de la table de gauche et **full** outer join qui conserve les lignes célibataires des deux tables.

Cette syntaxe n'est pas adoptée par tous les SGBD. C'est ainsi qu'Oracle utilise une variante de la jointure standard, illustrée ci-dessous par la reprise de la dernière requête. Le côté célibataire y est indiqué par le symbole "(+)" :

```
select NCOM, C.NCLI, DATECOM, NOM, LOCALITE
from COMMANDE M, CLIENT C
where M.NCLI = (+) C.NCLI
```

Conclusion. Il semble que seules les formes d'outer join apportent réellement plus de facilité dans l'écriture des requêtes SFW. En outre, la déclaration explicite de la jointure externe permet au SGBD d'exécuter cet opérateur, par ailleurs très coûteux, de manière efficace. Les autres formes peuvent entraîner une complexité inutile, et donc un risque d'erreurs. On leur préférera autant que possible les expressions plus régulières proposées en 5.4.1⁹. Le lecteur apprendra avec ravissement

8. Cette remarque n'a en soi aucun intérêt, mais illustre la complexité induite par ces formes de jointure.

9. Inélegamment qualifiées de *old style* par certains auteurs [Melton,1999].

que la norme SQL3 ajoute encore quelques formes supplémentaires de jointures relatives notamment aux clés étrangères, chacune plus indispensable que les autres.

d) Retour sur la notion de condition de non association

À la section 5.4.6, nous avons observé qu'une condition d'association pouvait s'exprimer indifféremment par une sous-requête ou par une jointure (sans parler de la forme *exists*). En revanche, une condition de *non-association* ne pouvait être traduite qu'en une sous-requête ou une forme *not exists*. Il est en fait possible de l'exprimer en passant pas le mécanisme de jointure externe. Les clients *qui n'ont pas passé de commandes* peuvent s'obtenir par les trois requêtes suivantes :

```
select NCLI,NOM
from   CLIENT
where  NCLI not in (select NCLI from COMMANDE)

select NCLI,NOM
from   CLIENT C
where  not exists (select * from COMMANDE
                  where NCLI = C.NCLI)

select NCLI,NOM
from   CLIENT C left outer join COMMANDE M
      on (C.NCLI = M.NCLI)
where  M.NCOM is null
```

e) Synthèse sur la forme des requêtes SFW

Ces extensions, ajoutées aux formes de base pourraient donner du langage SQL une image de complexité et d'incohérence. Bien que cette critique soit en partie méritée, les nouvelles constructions augmentent fortement la régularité du langage et simplifient ce dernier. Pour dire les choses plus concrètement, la probabilité qu'une requête construite par un rédacteur naïf soit correcte¹⁰ et s'interprète comme prévu, est plus grande qu'auparavant !

Une requête SFW comporte principalement deux parties (pour simplifier, on ignorera les clauses *group by*, *having*, *order by* et les fonctions agrégatives) :

```
select liste-valeurs
from   expression-de-table
```

où *liste-valeurs* désigne une liste d'expressions définissant chacune une valeur pour chaque ligne de la table *expression-de-table*;
et *expression-de-table* est une expression définissant une table réelle ou virtuelle (vue).

10. Un exemple concret : là où il est permis de mentionner une table, il est désormais possible d'y introduire une expression dont l'évaluation produit une table.

Un élément de *liste-valeurs* est toute expression qui peut renvoyer une valeur : le nom d'une colonne, une constante, une expression de calcul (arithmétique, chaîne de caractère, temporel), ou une expression SFW qui renvoie une seule valeur.

Il existe différents moyens de définir une table selon *expression-de-table*, qui tous ont comme propriété de renvoyer une suite de lignes : une table réelle de base ou une vue (le cas le plus fréquent), l'union (intersection, différence) de deux tables (elles-mêmes réelles ou vues), la jointure de deux tables (réelles ou vues) ou le résultat d'une requête SFW, qui, comme on le sait, renvoie une suite de lignes.

L'élégance de cette définition vient de sa récursivité : on peut rédiger une requête SFW dont la clause `from` contient la jointure de tables définies par des requêtes SFW, et ainsi de suite. Il faut reconnaître que les développeurs de SGBD n'apprécient que très modérément cette puissance, et que, dans la réalité, ces possibilités sont souvent fortement bridées, voire ignorées (l'interpréteur, hagard, déclarant forfait, parfois sans avertissement¹¹). Au lecteur de vérifier les possibilités offertes par son SGBD.

6.3.3 Les requêtes récursives

Comme nous l'avons constaté lors de l'étude des structures cycliques (section 5.4.5), SQL2 ne permet pas d'effectuer une auto-jointure de manière itérative le nombre de fois nécessaire pour épuiser une structure hiérarchique. Des extensions sont proposées dans SQL3, dont une variante est disponible dans Oracle. Nous illustrerons cette dernière en construisant la requête qui donne l'organigramme des personnes qui dépendent, directement ou non, de la personne p4.

```
select LEVEL, NPERS, NOM, RESPONSABLE
from   PERSONNE
start with NPERS = 'p4'
connect by prior NPERS = RESPONSABLE
```

Les clauses `start with` et `connect to` indiquent que cette requête est à appliquer de manière récursive :

- `start with` : spécifie le point de départ de la recherche (PERSONNE dont NPERS = 'p4');
- `connect` : indique comment on passe d'un niveau au suivant; RESPONSABLE est la colonne de la ligne inférieure qui doit être égale (=) à la colonne NPERS de la ligne supérieure (`prior`); à partir d'une ligne PERSONNE P déjà sélectionnée, la requête recherche toutes les lignes dont la valeur de RESPONSABLE est égale à celle de NPERS de P;
- la clause `where` (absente dans l'exemple) filtre les lignes sélectionnées;

11. Oracle avertit très honnêtement que l'exécution d'une requête peut produire des résultats erronés lorsque celle-ci est *trop complexe*, mais sans préciser ce que ce terme veut dire.

- la colonne fictive LEVEL est disponible dans les clauses `select` et `where`; elle représente le numéro de niveau dans l'arbre de parcours, la racine (ici `NPERS = 'p4'`) étant au niveau 1;
- la requête échoue lorsqu'une ligne déjà extraite est sélectionnée à nouveau, témoignant de l'existence d'un circuit.

Appliquée au contenu de la table PERSONNE de la figure 5.3, cette requête fournira le résultat suivant :

LEVEL	NPERS	NOM	RESPONSABLE
1	p4	Dupont	p1
2	p5	Verger	p4
3	p6	Dupont	p4
3	p7	Dermiez	p6

6.4 LES PRÉDICATS (*check*)

Un prédicat est une condition associée à un schéma, à une table ou à une colonne. Un prédicat lié à un schéma, le plus souvent dénommé *assertion*, définit une propriété que la base de données doit respecter après chaque opération de modification. Ce type de prédicat étant rarement implémenté dans les SGBD, nous limiterons notre discussion aux autres types.

Un prédicat de table ou de colonne définit une propriété que les lignes de la table doivent respecter à tout instant. Un prédicat est évalué lors de toute modification d'une ligne de la table. En cas de violation, la modification est rejetée.

Considérons par exemple que les valeurs de la colonne CAT sont limitées à une liste prédéfinie. On écrira alors :

```
create table CLIENT ( NCLI ...,
                    ...,
                    CAT char(2),
                    primary key (NCLI),
                    check (CAT is null or
                          CAT in ('B1','B2','C1','C2'))
```

Cette contrainte peut être ajoutée a posteriori :

```
alter table CLIENT
add check (CAT is null or CAT in ('B1','B2','C1','C2'))
```

La forme suivante permet de nommer la contrainte¹² :

12. Aussi valable dans la déclaration de la table :

```
constraint CHK_CAT check (...)
```



```
alter table CLIENT
add constraint CHK_CAT
check (CAT is null or CAT in ('B1','B2','C1','C2'))
```

Le prédicat ci-dessous garantit que les dates de commande (lorsque l'ensemble des commandes est non vide) sont correctement attribuées selon l'ordre chronologique¹³ :

```
alter table COMMANDE
add check ((DATECOM >= (select max(DATECOM)
                        from COMMANDE)
and DATECOM <= CURRENT_DATE) is not false)
```

Un prédicat peut contribuer au respect d'une contrainte référentielle :

```
alter table COMMANDE
add check (NCLI in (select NCLI from CLIENT))
```

Ce dernier protège la table COMMANDE contre l'introduction de valeurs de NCLI erronées, mais n'offre aucune garantie quant aux modifications du contenu de la table CLIENT. Il est donc préférable de s'en tenir à la déclaration explicite des clés étrangères ou, si on désire mettre en place une gestion personnalisée, faire appel à un déclencheur (voir ci-après).

Une contrainte nommée peut être supprimée :

```
alter table CLIENT
drop constraint CHK_CAT
```

Une clause check qui concerne une seule colonne peut être écrite dans la déclaration de cette dernière :

```
CAT char(2) check(CAT is null or
                  CAT in ('B1','B2','C1','C2')),
```

6.5 LES PROCÉDURES SQL (stored procedures)

Une procédure SQL est une séquence d'instructions SQL précompilées dont l'exécution peut être demandée par un utilisateur, un programme d'application, un *trigger* ou une autre procédure SQL. Une procédure autorise l'utilisation de paramètres. Ceux-ci consistent en des valeurs qui sont communiquées à la procédure ou, au contraire, que la procédure a calculées et qu'elle communique en retour.

13. Certains SGBD (tels Oracle et DB2) n'admettent qu'une forme réduite de prédicats, dans lesquels seules les valeurs de la ligne courante peuvent être citées. Pour ces SGBD, les deux prédicats suivants sont donc interdits. Ils sont en revanche autorisés en InterBase.

L'avantage d'une procédure SQL par rapport à une procédure intégrée à un programme d'application est qu'elle est stockée dans la base de données et qu'elle peut donc être considérée comme une ressource unique et commune pour toutes les applications, évitant à celles-ci une duplication de code intempestive. Elle permet aussi de définir des comportements complexes, en particulier en ce qui concerne l'intégrité des données.

A titre d'exemple, la procédure SUP_DETAIL supprime la ligne de détail dont elle reçoit (in) le numéro de commande (COM) et le numéro de produit (PRO). Si cette ligne est la dernière de sa commande, cette dernière est également supprimée¹⁴ :

```
create procedure SUP_DETAIL (in COM char(12),
                           in PRO char(15))
begin
  delete from DETAIL
  where NCOM = :COM and NPRO = :PRO;
  if (select count(*) from DETAIL
      where NCOM=:COM) = 0
  then delete from COMMANDE
      where NCOM = :COM
  end if;
end;
```

On pourra l'invoquer par une commande telle que la suivante :

```
call SUP_DETAIL('30182', 'PA60');
```

6.6 LES DÉCLENCHEURS (*triggers*)

Un *trigger* est un mécanisme constitué d'une section de code accompagnée des conditions qui entraînent son exécution. Sa forme générale est la suivante :

```
before/after E
when C
begin
  A
end
```

et s'interprète de la manière suivante : *dès qu'un événement **E** survient, si la condition **C** est satisfaite, alors exécuter l'action **A** soit avant (before) soit après (after) **E**.*

De cette forme découle le nom de **mécanisme E-C-A** (événement-condition-action). On considère quatre types d'événements : insertion d'une ligne, suppression d'une ligne, modification d'une ligne (toutes colonnes confondues) et modification d'une colonne d'une ligne.

14. Cette procédure, ainsi que les déclencheurs de la section suivante sont rédigés dans une syntaxe fictive proche de celle de PL/SQL d'Oracle ou du langage d'InterBase. On notera que l'usage d'un argument est préfixé de ":" afin d'éviter toute ambiguïté avec un nom de colonne.

L'exemple ci-dessous définit la modification automatique de la valeur de CAT lorsque l'état du compte d'un client descend en dessous d'un certain seuil.

```
create trigger MAJ_CLI
before update of COMPTE on CLIENT
for each row
when (new.COMPTE < -10000)
begin
    if old.CAT = 'B2'
        then new.CAT := 'B1';
    end if;
    if old.CAT = 'C2'
        then new.CAT := 'C1';
    end if;
end;
```

Le *trigger* porte le nom de MAJ_CLI. L'événement déclencheur est la modification de la colonne COMPTE de CLIENT (update of COMPTE on CLIENT). Pour chaque ligne ainsi modifiée (for each row), et si la nouvelle valeur de COMPTE est inférieure à -10.000 (when new.COMPTE < -10000), alors, avant la modification (before update), les valeurs de CAT sont rétrogradées de B2 en B1 et de C2 en C1. Dans cette interprétation, old et new désignent respectivement les états de la ligne en cours de modification *avant* et *après* la modification (update). En cas de création (insert), seule la version new existe, tandis que pour une suppression (delete), seule la version old est accessible. L'expression new.COMPTE désigne donc la nouvelle valeur de COMPTE de la ligne courante. L'assignation new.CAT := 'B1' permet de modifier les valeurs de CAT avant l'enregistrement.

Remarque

La forme générale E-C-A peut être simplifiée, sans perte de généralité, comme suit :

```
before/after E
begin
    if C then
        A
    end if
end
```

C'est d'ailleurs la seule forme admise par certains SGBD, tels que Firebird.

6.7 LE CATALOGUE

L'environnement des bases de données SQL comporte, outre les tables créées par ses utilisateurs, une collection de tables dont le contenu décrit les structures de cette base. Ces tables constituent le *dictionnaire des données* ou encore le *catalogue du système*.

a) Principes d'un dictionnaire de données

Tout SGBD relationnel gère, outre les tables définies par les utilisateurs, un ensemble de tables standard dont le contenu décrit tous les objets qui ont été déclarés par les utilisateurs et l'administrateur¹⁵ de la base de données. En particulier, ces tables permettent de savoir quelles tables ont été définies et renseignent sur la définition de leurs colonnes, des index, des privilèges (y compris les mots de passe), les utilisateurs eux-mêmes ainsi que les programmes qui utilisent la base de données. Ces informations étant rangées dans des tables SQL ordinaires, il est possible à l'utilisateur qui en a l'autorisation de consulter leur contenu.

Ces tables constituent ce qu'on appelle un *dictionnaire de données*. Elles sont utilisées par le SGBD SQL, qui peut ainsi vérifier la validité des requêtes, les traduire en algorithmes efficaces et les exécuter. Ces tables seront également utilisées lors de l'exécution d'un programme, afin de vérifier que la structure de la base de données n'a pas été modifiée depuis la compilation du programme (et, le cas échéant, recompiler automatiquement le programme) et d'effectuer un contrôle d'accès.

b) Les tables du catalogue

Le dictionnaire de données que nous venons de décrire, ou *catalogue du système*, contient plusieurs dizaines de tables, dont le nombre, le nom, la composition et l'interprétation sont susceptibles de varier selon les SGBD et selon les versions.

Les deux tables les plus importantes sont certainement la *table des tables* et la *table des colonnes*. La première (que nous appellerons `SYS_TABLE`) nous renseigne sur les tables qui constituent la base de données tandis que la seconde (`SYS_COLUMN`) décrit leurs colonnes (figure 6.1). Nous décrirons brièvement la composition de ces deux tables¹⁶.

Chaque ligne de **SYS_TABLE** décrit une des tables de la base de données. Elle en spécifie le nom (`TNAME`), le créateur et propriétaire (`CREATOR`) et le type (`TTYPE`), qui est *réel* (R) ou *vue* (V).

La table **SYS_COLUMN** indique pour chaque colonne de la base de données le nom de sa table (`TNAME`), son propre nom (`CNAME`), le type de ses valeurs (`CTYPE`), leur longueur (`LEN1`), le nombre de décimales (`LEN2`) ainsi que son caractère obligatoire (`NULLS = N`) ou facultatif (`NULLS = Y`). Ces tables comportent d'autres colonnes qui ne présentent pas d'intérêt dans une présentation introductive.

On observera que ces deux tables décrivent non seulement les tables et colonnes des utilisateurs, mais aussi celles du catalogue (y compris `SYS_TABLE` et

15. Un administrateur d'une base de données destinée à desservir une communauté d'utilisateurs est le responsable de la création, du bon fonctionnement de la base de données et de la qualité des données. C'est lui qui est chargé, entre autres choses, de la gestion des privilèges et de l'évolution de la base de données.

16. Le script de création des quatre tables du catalogue en InterBase est disponible sur le site de l'ouvrage. Ce catalogue est évidemment fictif.

SYS_COLUMN elles-mêmes ainsi que leurs colonnes), dont le créateur est par convention SYSTEM.

SYS_TABLE			SYS_COLUMN					
TNAME	CREATOR	TTYPE	TNAME	CNAME	CTYPE	LEN1	LEN2	NULLS
SYS_TABLE	SYSTEM	R	SYS_TABLE	TNAME	varchar	18		N
SYS_COLUMN	SYSTEM	R	SYS_TABLE	CREATOR	char	8		N
SYS_KEY	SYSTEM	R
SYS_KEY_COMP	SYSTEM	R	CLIENT	ADRESSE	char	60		N
....	CLIENT	CAT	char	2		Y
CLIENT	AGF	R	CLIENT	COMPTE	decimal	9	2	N
COMMANDE	AGF	R	CLIENT	LOCALITE	char	30		N
DETAIL	AGF	R	CLIENT	NCLI	char	10		N
PRODUIT	PDE	R	CLIENT	NOM	char	32		N
VAL_STOCK	AFB	V	COMMANDE	DATECOM	date			N
....	COMMANDE	NCLI	char	10		N
			COMMANDE	NCOM	char	12		N
			DETAIL	NCOM	char	12		N
			DETAIL	NPRO	char	15		N
			DETAIL	QCOM	decimal	8	0	N
			PRODUIT	LIBELLE	char	60		N
			PRODUIT	NPRO	char	15		N
			PRODUIT	PRIX	decimal	6	0	N
			PRODUIT	QSTOCK	decimal	8	0	N
			VAL_STOCK	STOCK	date			N
			VAL_STOCK	VALEUR	decimal	12	0	N
		

Figure 6.1 - Les deux tables principales du catalogue : la table des tables et la table des colonnes

La représentation des identifiants et clés étrangères est un peu plus complexe¹⁷. La table SYS_KEY représente les *clés* et leurs relations, tandis que la table SYS_KEY_COMP décrit leurs composants. Chaque ligne de SYS_KEY décrit un identifiant primaire (KTYPE = P) ou une clé étrangère (KTYPE = F). Elle donne pour chacune son code interne qui permet de la distinguer (KEYID), sa table (TNAME), son type (KTYPE) et, pour les clés étrangères, le code de l'identifiant correspondant dans la table cible (KTARG).

Chaque ligne de la table SYS_KEY_COMP représente un composant d'une *clé*. Elle reprend le code de la *clé*, le nom de la colonne et son numéro d'ordre.

D'autres tables précisent la définition de chaque vue, décrivent les contraintes, les triggers et les procédures, répertorient les utilisateurs, définissent les privilèges, décrivent les index et espaces de stockage ainsi que leur mode et paramètres d'implémentation, décrivent les programmes d'application, et contiennent des statistiques sur les données. Il existe enfin des tables qui contiennent des données propres au SGBD, et qui sont sans signification pour l'utilisateur.

17. D'autant plus que, pour des raisons de performances, les représentations effectivement adoptées dans les SGBD sont fortement optimisées, et donc différentes de celle qui est discutée ici. L'exercice 6.3 aborde cette question.

SYS_KEY			
KEYID	TNAME	KTYPE	KTARG
K1	CLIENT	P	
K2	COMMANDE	P	
K3	COMMANDE	F	K1
K4	DETAIL	P	
K5	DETAIL	F	K2
K6	DETAIL	F	K7
K7	PRODUIT	P	
....

SYS_KEY_COMP		
KEYID	CNAME	KSEQ
K1	NCLI	1
K2	NCOM	1
K3	NCLI	1
K4	NCOM	1
K4	NPRO	2
K5	NCOM	1
K6	NPRO	1
K7	NPRO	1
....

Figure 6.2 - Deux tables du catalogue représentant les identifiants primaires, les clés étrangères, leurs composants et leurs relations

c) Utilisation du catalogue

Il apparaît immédiatement que le contenu des tables du catalogue ne peut être modifié comme pourrait l'être celui des tables ordinaires. En effet, toute modification intempestive de ces données entraînerait de graves conséquences sur le fonctionnement de la base de données. Par exemple, supprimer une ligne de la table SYS_TABLE rend inaccessible la table que cette ligne décrit. En principe, le contenu des tables du catalogue ne peut être modifié que par le système de gestion lui-même lorsqu'il exécute une requête telle que create table, create index, drop table, alter table, create view, grant, revoke, etc. En revanche, ces données peuvent être consultées par tout utilisateur qui en a reçu l'autorisation. Nous donnerons ci-après quelques exemples de requêtes représentatives. D'autres traitements, plus complexes, seront décrits en 7.5.4.

- Quelles sont les colonnes de la table DETAIL ?

```
select  CNAME, CTYPE, LEN1, NULLS
from    SYS_COLUMN
where   TNAME = 'DETAIL'
```

CNAME	CTYPE	LEN1	NULLS
NCOM	CHAR	12	N
NPRO	CHAR	15	N
QCOM	DECIMAL	8	N

- Dans quelles tables de base existe-t-il des colonnes dont le nom commence par 'NCOM' ?

```
select  TNAME
from    SYS_TABLE
where   TNAME in (select TNAME
                  from  SYS_COLUMN
                  where  CNAME like 'NCOM%')
and     TTYPE = 'R'
```

TNAME

COMMANDE
DETAIL

- Quelles sont les propriétaires des tables référencées par la table DETAIL ?

```
select distinct CREATOR
from   SYS_TABLE
where  TNAME in
        (select TNAME
         from   SYS_KEY
         where  KEYID in
              (select KTARG
               from   SYS_KEY
               where  TNAME = 'DETAIL'))
```

CREATOR

AGF
PDE

6.8 LES EXTENSIONS PROPOSÉES PAR SQL3

La norme SQL3, ou SQL:1999, comprend un nombre considérable d'extensions par rapport à SQL2. Son élaboration ayant pris plus de huit ans, il n'est pas étonnant que certaines de celles-ci aient déjà été intégrées aux versions récentes des SGBD avant même la stabilisation du standard. Tel est le cas des domaines définis par l'utilisateur, des déclencheurs, des procédures SQL et de certaines extensions objets. La présentation et la discussion de cette norme occuperaient plusieurs volumes (voir par exemple [Melton, 2002] et [Melton, 2003] et dépasseraient les objectifs méthodologiques de cet ouvrage. Nous nous contenterons donc de citer quelques-unes des principales nouveautés concernant les structures de données.

- Une colonne peut être constituée d'un agrégat de valeurs élémentaires.
- Une colonne peut être constituée d'un tableau de valeurs.
- Une valeur élémentaire peut être la référence d'une ligne d'une table; cette valeur est alors considérée comme la ligne référencée elle-même, par opposition à la clé étrangère.
- Un type de données peut être défini par sa structure, ses fonctions et ses propriétés d'héritage. Il est possible de définir une table à partir d'un tel type; elle se comporte alors comme une classe d'objets de ce type. Le couplage des bases de données avec les langages orientés-objet en est facilité [Bouzeghoub, 1997].

6.9 LES INTERFACES ENTRE BD ET PROGRAMMES D'APPLICATION

En principe, toute commande SQL peut être transmise au SGBD SQL par un programme d'application en vue de son exécution immédiate. Dans ce cas, les instructions SQL auront été incorporées dans le programme¹⁸ selon des règles que nous allons décrire brièvement. Etant donné la nature séquentielle des langages de programmation classiques (COBOL, C, Java, Basic, etc.), un protocole de communication spécifique a été établi entre le programme d'application et le SGBD. Il concerne d'une part les échanges entre ces derniers, et d'autre part le mode d'interaction entre un résultat ensembliste (le SGBD fournit une table en réponse à une requête d'extraction) et un programme itératif.

a) Extraction des données

En ce qui concerne l'extraction des données, le programme doit pouvoir traiter chaque ligne du résultat en séquence. D'autre part il doit pouvoir passer des valeurs de paramètres en argument d'une commande, et recevoir les valeurs d'une ligne de résultat dans des variables qui lui sont propres. Il existe donc des structures de contrôle et des conventions de communication liées à l'utilisation d'une base de données SQL par un programme d'application.

Du point de vue syntaxique, il existe quelques conventions d'écriture. Elles sont liées au mode de traduction d'un programme : celui-ci est généralement traité par un préprocesseur SQL qui transforme les ordres SQL en appels à des procédures système. Le programme peut alors être compilé et relié d'une manière traditionnelle. En outre, ce préprocesseur effectue des vérifications de conformité des ordres par rapport aux définitions qui sont enregistrées dans le dictionnaire de données (en particulier les privilèges); il met également ce dernier à jour.

Nous préciserons les conventions qui sont d'application pour les langages tels que COBOL. Elles sont très semblables pour C ou PASCAL. Le cas de Java est un peu particulier, dans la mesure où ce langage dispose de ses propres protocoles d'interaction avec les SGBD SQL (JDBC ou SQLJ).

1. Tout ordre SQL doit être précédé de `exec SQL` et clôturé par `end-exec`.
2. Les variables qui interviennent dans une commande SQL, soit comme arguments, soit comme résultats, doivent être déclarées dans un bloc délimité par les instructions :

```
exec SQL  begin declare section  end-exec.  
et  
exec SQL  end declare section  end-exec.
```

3. Les communications entre le programme et le SGBD SQL utilisent certaines variables système (dont `SQLCODE`) qu'il faut déclarer, ou que l'on peut faire in-

18. D'où l'appellation *ESQL*, ou *embedded SQL*, par opposition à *ISQL*, ou *interactive SQL*, que nous avons considéré jusqu'ici.

introduire par le pré-processeur par la commande :

```
exec SQL include SQLCA end-exec
```

Une requête dont le résultat comporte une seule ligne est en accord avec les principes des langages de programmation classiques, qui traitent un enregistrement à la fois. L'exemple ci-dessous, exprimé dans un langage procédural fictif, illustre ces principes. On observe que la requête SQL comporte une clause supplémentaire (into) qui indique la destination des données extraites. Les noms des variables externes apparaissant dans la requête sont préfixés du symbole ":".

```
exec SQL begin declare section end-exec;
  NOM   char(32);
  ADR   char(60);
  LOC   char(30);
  NUM   char(10);
exec SQL end declare section end-exec;

read NUM;

exec SQL
  select NOM,ADRESSE,LOCALITE into :NOM,:ADR,:LOC
  from CLIENT
  where NCLI = :NUM
end-exec;

display NOM, ADR, LOC;
```

En revanche, un résultat comportant plusieurs lignes doit faire l'objet de précautions particulières. Le protocole proposé s'inspire du schéma classique d'accès séquentiel à un fichier, qui correspond généralement à ce qui suit :

```
déclarer le fichier F
...
ouvrir F
lire un enregistrement
tant que la lecture a réussi faire
  traiter l'enregistrement
  lire un enregistrement
fin
fermer F
```

En SQL, le fichier F correspond au résultat d'une requête d'extraction, auquel est associé un *curseur*. Ce dernier doit être vu comme un fichier logique qu'il faut ouvrir, qu'on peut lire ligne par ligne, puis qu'il faut fermer lorsque son traitement est terminé.

L'accès aux lignes successives d'une table résultat est rendu possible grâce aux principes suivants.

1. La requête SQL est transmise au SGBD par une commande de définition de

curseur (`declare cursor`). Le résultat de l'exécution de cette requête peut être considéré comme un fichier. Pour accéder à ses enregistrements (les lignes), il faut *ouvrir* ce fichier, *demandeur* chaque ligne jusqu'à la dernière, puis *fermer* le fichier. Le curseur, qui porte un nom qui l'identifie, est associé à ce fichier résultat, et plus particulièrement à une ligne de celui-ci. Ce fichier n'a pas nécessairement d'existence matérielle. La commande de définition `declare cursor` est essentiellement déclarative; elle n'entraîne pas l'exécution de la requête.

2. L'ouverture du fichier résultat (`open`) provoque l'évaluation de la requête à partir des valeurs courantes des paramètres et le positionnement du curseur sur la première ligne, si elle existe.
3. Si le curseur référence une ligne, la demande d'accès (`fetch`) fournit cette ligne au programme tandis que le curseur est avancé d'une position.
4. Un indicateur de diagnostic (`SQLCODE`) précise à l'issue de chaque exécution d'une commande la manière dont celle-ci s'est déroulée. A titre d'exemple, `SQLCODE = 0` indique que l'opération s'est déroulée correctement, et `SQLCODE = 100` indique notamment qu'aucune ligne n'a été trouvée (*fin de fichier*).
5. La fermeture du fichier résultat (`close`) entraîne l'abandon du fichier résultat. Toute ouverture ultérieure provoquera à nouveau l'évaluation de la requête. Entre une fermeture et une ouverture, on peut donc modifier des valeurs de paramètres de la requête.
6. Dans une commande d'accès il faut préciser les variables dans lesquelles le SGBD rangera les valeurs de la ligne obtenue.
7. La définition d'une requête SQL peut contenir des noms de variables en lieu et place des constantes dans la condition de sélection.

L'exemple suivant extrait et traite les informations relatives aux clients de Toulouse.

```
...
exec SQL declare CURCLI cursor for
  select NCLI, NOM, ADRESSE
  from   CLIENT
  where  LOCALITE = :LOC
end-exec;
...
LOC := 'Toulouse';
exec SQL open CURCLI end-exec;
exec SQL fetch CURCLI into :NUM, :NOM, :ADR end-exec;
while SQLCODE = 0 do
  <traiter le client>
  exec SQL fetch CURCLI into :NUM, :NOM, :ADR end-exec;
endwhile;
exec SQL close CURCLI end-exec;
...
```

b) Modification des données

Les commandes `insert`, `delete`, et `update` peuvent, elles-aussi, apparaître dans un programme, selon les conventions syntaxiques spécifiées ci-dessus. Les valeurs

spécifiées dans ces commandes peuvent l'être sous forme de constantes (comme dans les exemples SQL présentés jusqu'ici), ou de variables du programme.

Il existe une variante particulière des requêtes `update` et `delete`, qualifiée de `positioned`. Elle consiste à traiter la ligne courante d'un curseur, comme dans l'exemple suivant :

```
...
exec SQL declare CURCLI cursor for
  select ... from CLIENT where LOCALITE = :LOC
end-exec;
...
LOC := 'Toulouse';
exec SQL open CURCLI end-exec;
exec SQL fetch CURCLI into :NUM,:NOM,:ADR end-exec;
while SQLCODE = 0 do
  exec SQL
    update CLIENT
    set CAT = 'A1'
    where current of CURCLI
  end-exec;
  exec SQL fetch CURCLI into :NUM,:NOM,:ADR end-exec;
endwhile;
exec SQL close CURCLI end-exec;
...
```

c) Définition et modification des structures de données

Les commandes SQL `create table`, `create index`, `alter table`, `drop table`, `drop index` peuvent être exécutées par un programme. Elles apparaissent dans le texte de celui-ci entre `exec SQL` et `end-exec`.

d) Définition dynamique de commandes SQL

Il existe une possibilité de définir dynamiquement une commande SQL. Ceci permet de ne définir une commande qu'au moment où le programme s'exécute, par exemple sur base de directives obtenues au terminal. Le programme construit la commande sous la forme d'une chaîne de caractères. Cette chaîne est analysée par SQL (par opposition à la définition statique, qui est analysée par le préprocesseur), puis peut être exécutée. Ceci permet la réalisation de programmes exploitant la base de données d'une manière non prédéfinie.

Les interfaces ODBC de Microsoft et JDBC (Java) correspondent également à une exécution dynamique. En effet, elles consistent à transmettre au SGBD les requêtes sous la forme de chaînes de caractères non validées. Ces questions dépassant le cadre de l'ouvrage, le lecteur est renvoyé à la littérature plus spécialisée [Delmal, 2001], [Reese, 1998], [Geiger, 1995].

6.10 SQL ET L'INFORMATION INCOMPLÈTE

6.10.1 Introduction

La question de l'information incomplète ou manquante est l'une des plus complexes du domaine des bases de données. Elle couvre plusieurs situations distinctes telles que les suivantes, qu'on peut illustrer sur la table **CLIENT**, qui contient des informations sur un client qui *n'a pas de valeur de CAT* :

- *information pertinente mais absente* : le client pourrait avoir une catégorie, mais on ne lui en a pas encore attribué;
- *information non pertinente* : de par son statut le client n'a pas de catégorie,
- *information inconnue* : le client a une catégorie, mais celle-ci n'est pas connue.

6.10.2 La valeur `null` de SQL

Malheureusement, SQL ne propose qu'un seul mécanisme pour représenter ces différents cas de figure, la valeur `null`. Dans la plupart des cas, cette valeur représente le fait que l'information n'est pas connue. En théorie, `null` n'est pas une valeur, mais un **marqueur** indiquant que la *case est vide*. Nous verrons cependant qu'à l'occasion, SQL la considère comme une valeur réelle.

6.10.3 La logique ternaire de SQL

L'information incomplète présente aussi un autre visage sous la forme de la valeur logique *inconnu*, qui se note `unknown` en SQL. En effet, l'évaluation d'un prédicat, dans une clause `where` par exemple, sur une ligne particulière peut renvoyer la valeur `vrai`, la valeur `faux`, mais aussi la valeur *inconnu*, signifiant par là qu'il est impossible de décider si la valeur est `vrai` ou `faux`. L'un des problèmes est que dans certaines circonstances, *inconnu* signifie *soit vrai soit faux*, alors qu'à d'autres moments, il signifie *ni vrai ni faux*.

Techniquement, le comportement de la valeur *inconnu* est défini par les tables de vérité des opérateurs \neg (not), \wedge (and), \vee (or), \oplus , \Rightarrow , et \Leftrightarrow dont le contenu est donné ci-dessous¹⁹, `inc` étant mis pour *inconnu* :

<i>P</i>	<i>Q</i>	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \oplus Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
vrai	vrai	faux	vrai	vrai	faux	vrai	vrai
vrai	faux	faux	faux	vrai	vrai	faux	faux
vrai	inc	faux	inc	vrai	inc	inc	inc
faux	vrai	vrai	faux	vrai	vrai	vrai	faux
faux	faux	vrai	faux	faux	faux	vrai	vrai

19. On notera que \neg *inconnu* = *inconnu*, ce qui est *logique* mais pas *intuitif*.

faux	inc	vrai	faux	inc	inc	vrai	inc
inc	vrai	inc	inc	vrai	inc	vrai	inc
inc	faux	inc	faux	inc	inc	inc	inc
inc	inc	inc	inc	inc	inc	inc	inc

L'élaboration de cette table est plus simple qu'il n'y paraît. La valeur de vérité de $P \text{ op } Q$, pour tout opérateur op , se détermine comme suit, lorsque P (ou Q) est inconnu : on remplace P successivement par *vrai*, puis par *faux*, et on observe les valeurs résultantes selon la table de la section 5.2.6. Si le résultat de ces deux expressions est le même, on inscrit celui-ci dans le tableau, sinon, on inscrit *inconnu*.

Considérons l'expression $P \Rightarrow Q$, lorsque $P = \text{vrai}$ et $Q = \text{inconnu}$ (3e ligne du tableau). Si $Q = \text{vrai}$, alors l'expression vaut *vrai*, alors qu'elle vaut *faux* lorsque $Q = \text{faux}$. Ces deux résultats étant différents, la valeur de $\text{vrai} \Rightarrow \text{inconnu}$ est donc *inconnu*.

Dans ce qui suit, nous utiliserons les dénominations *true*, *false* et *unknown* propres à SQL.

Découlant de cette logique, SQL propose un jeu de prédicats binaires permettant de référencer explicitement ces trois valeurs (C est un prédicat binaire ou ternaire quelconque) :

- **$C \text{ is true}$** vaut *true* si C vaut *true*, et *false* si C vaut *false* ou *unknown*;
- **$C \text{ is false}$** vaut *true* si C vaut *false*, et *false* si C vaut *true* ou *unknown*;
- **$C \text{ is unknown}$** vaut *true* si C vaut *unknown*, et *false* si C vaut *true* ou *false*.

Ces prédicats absorbent la valeur *unknown*. Les formes inverses *is not true*, *is not false* et *is not unknown* sont aussi disponibles.

Par exemple, la condition " $(\text{CAT} = 'C2') \text{ is true}$ " vaudra *true* pour le client F400, et *false* pour tous les autres, y compris D063 et K729.

6.10.4 La propagation de null en SQL

En général, SQL suit des règles conformes à la *logique* (au sens mathématique du terme) mais s'en écarte parfois lorsque celles-ci ne correspondent pas à l'intuition. Cela donne comme on le verra un comportement peu homogène difficile à maîtriser.

a) Les expressions de calcul

Une expression dont l'évaluation renvoie une valeur numérique, caractères ou temporelle est évaluée à *null* si l'un de ses arguments est *null*. Cette règle ne concerne pas les fonctions qui *absorbent* les valeurs *null*, telles que *case*, *coalesce* ou *nullif*. Quelques exemples (on admet pour les besoins de la cause que *CAT*, mais aussi *COMPTE* puissent prendre la valeur *null*) :

- $A + B + C$ vaut null *si A ou B ou C est null*;
- `COMPTE - (select sum(QCOM*PRIX) from .. where ..)` vaut null *si la sous-requête from-where correspond à un ensemble vide*;
- `cast(CAT as varchar(10))` vaut null *si CAT = null*;
- `ADRESSE || ' - ' || CAT` vaut null *si CAT = null*;
- `COMPTE - COMPTE` vaut null *si COMPTE = null*.

b) Les fonctions agrégatives

Ces fonctions travaillent sur des ensembles dont les éventuelles valeurs null sont ignorées. Si l'ensemble est vide, les fonctions sum, avg, min, et max renvoient null, tandis que count renvoie 0.

On notera que cette règle ne suit pas celle des expressions de calcul. En effet, la fonction sum devrait, comme toute somme, renvoyer null dès que l'un au moins de ses éléments est null, ce qui n'est pas le cas. L'intuition l'emporte ici sur la rigueur.

6.10.5 La propagation de unknown en SQL

La toute première règle est la suivante : la clause where P d'une instruction SFW ne retient que les lignes pour lesquelles P est évalué à true. Elle écarte donc les lignes pour lesquelles P vaut false ou unknown. Nous précisons ensuite les principales règles d'évaluation des prédicats, qui s'ajoutent à la table de vérité décrite ci-dessus.

a) Les expressions de comparaison

Une comparaison entre deux valeurs dont une au moins est null renvoie unknown. C'est ainsi que le prédicat `CAT = CAT` renvoie unknown pour les lignes de CLIENT dont `CAT = null`.

b) Les prédicats absorbants is true, is false, is unknown et leurs inverses

Sont évalués à true ou false, jamais unknown. Malgré leur lourdeur ces expressions sont plus explicites que les autres formes, et donc moins sujettes à erreur d'interprétation.

On notera que `P is not true` n'est pas équivalent à `not P`. En effet, si P est évalué à unknown, la première forme a pour valeur true tandis que `not P` vaut unknown.

c) Les prédicats is null et is not null

Sont évalués à true ou false, jamais unknown.

d) Les prédicats in et not in

Leur effet se déduit de leur développement : soit $E \equiv \{a, b, c\}$ l'ensemble de référence, chacune de ces trois valeurs pouvant être null (ici, null est donc une valeur)

$A \text{ in } E$ est équivalent à $(A = a) \text{ or } (A = b) \text{ or } (A = c)$. On en déduit les règles suivantes.

- Le prédicat prend la valeur **true** si A est l'un des éléments non null de E .
- Il prend la valeur **unknown** si (1) $A = \text{null}$ ou si (2) A est non null, n'est égal à aucun élément non null de E et E contient null.
- Dans les autres cas, le prédicat prend la valeur **false**.

En particulier, si A est null et E est vide, la valeur est unknown²⁰.

Le prédicat `not in` s'étudie de manière similaire.

e) Les prédicats `exists` et `not exists`

Sont évalués à true ou false, jamais unknown. Attention, en présence de valeurs null dans le corps de l'argument de ces prédicats, l'interprétation peut être surprenante (voir exemple plus loin).

f) Les quantificateurs `all` et `any`

Les formes " $A \text{ op any } (E)$ " et " $A \text{ op all } (E)$ ", où `op` est un opérateur de comparaison et E une expression dont la valeur est une table, sont évaluées à unknown si A vaut null ou E désigne un ensemble vide. Cependant certains SGBD interprètent différemment la situation où A vaut null et E désigne un ensemble vide [Date, 1992].

6.10.6 Les problèmes de l'information incomplète en SQL

La nature même de l'information incomplète, et plus encore la manière dont SQL gère ses représentations induisent des problèmes d'interprétation et de cohérence qui doivent rendre l'utilisateur particulièrement circonspect.

Montrons d'abord que SQL ne gère pas toujours la valeur null de manière cohérente.

a) En principe, nul n'est pas égal à null

Exécutons la requête suivante, qui donne un résultat sans surprise :

```
select NCLI from CLIENT where CAT is null
```

NCLI
K729
D063

20. Assez maladroitement, la norme indique qu'un ensemble vide peut être considéré comme un ensemble contenant la valeur null. Cependant ce cas particulier n'est pas traité de la même manière par tous les SGBD [Date, 1992]. La requête `select CAT from CLIENT where NCLI = 'D063'` renvoie un résultat formé d'une ligne constituée d'une valeur null. Cependant, la requête `select count (CAT) from CLIENT where NCLI = 'D063'` renvoie 0. Les deux résultats sont conformes à la norme mais sont cependant mathématiquement incohérents.

Il n'en va pas de même de la requête ci-dessous qui renvoie un résultat vide !

```
select NCLI
from CLIENT
where CAT = (select CAT
              from CLIENT
              where NCLI = 'K729')
```

Comme nous l'avons déjà vu dans la section 5.3.4, la raison en est que la comparaison de deux expressions dont la valeur est `null` renvoie `unknown` (voir règle ci-dessus). On pourrait estimer que SQL aurait pu se montrer plus conciliant (plus *intelligent*), et renvoyer au moins `K729`, qui, de toute évidence, devrait avoir la même valeur de `CAT` que lui-même ! C'est donc la règle "*X = Y vaut unknown si X ou Y vaut null*" qui prévaut.

D'ailleurs, la condition `CAT = CAT` induira un comportement analogue : elle écartera les deux lignes litigieuses.

On peut encore ajouter à la liste des interprétations correctes, bien que contraires à l'intuition, le fait qu'en présence de valeurs `null`, l'expression `X < X` (ou `X <> X`) bien que fausse ne vaut pas `false` et `X = X` bien que vraie ne vaut pas `true` !

b) ... sauf, hélas, dans certains cas

Malheureusement, les choses ne sont pas toujours aussi simples. Constituons par exemple des groupes de clients par catégories :

```
select CAT, count(*)
from CLIENT
group by CAT
```

Le résultat semble naturel à première vue :

CAT	count(*)
B1	4
B2	4
C1	5
C2	1
<null>	2

Or, pour constituer le dernier groupe, il faut admettre que toutes les lignes dont `CAT = null` ont *même valeur* de `CAT`, et donc qu'ici `null = null`. Certains SGBD préféreront faire de chaque ligne dont `CAT = null` un groupe autonome, ce qui est cette fois cohérent avec la règle de comparaison, mais peu intuitif.

Citons un autre exemple, celui d'un identifiant formé d'une colonne facultative. Supposons que les produits d'exportation reçoivent un numéro d'agrégation unique, ce que nous traduisons par une nouvelle colonne associée à la table `PRODUIT`. Puisque ce numéro ne concerne que certains produits, la colonne est déclarée facultative; ses valeurs étant uniques, on y ajoute une clause `unique` :


```
NUMAGREATION char(12) unique;
```

Dans certains SGBD, cette contrainte d'unicité porte également sur les valeurs `null`. Il est donc possible d'insérer **une** ligne sans valeur pour cette nouvelle colonne, mais **pas deux** ! La définition d'un `unique index` est sans espoir : le comportement sera identique²¹. Conclusion, ici encore, `null = null`.

c) Null est-il plus grand que 'C2' ?

La valeur `null` n'est pas comparable aux autres valeurs et est donc exclue de toute relation d'ordre. Cependant, la requête suivante donne une liste ordonnée dans laquelle `null` apparaît en bonne place (sa position dépend du SGBD) :

```
select distinct CAT from CLIENT order by CAT;
```

CAT
B1
B2
C1
C2
<null>

On pourrait donc légitimement penser que le prédicat (`CAT < X`) est évalué à `true` au lieu de `unknown` lorsque `X = null` et que `CAT` n'est pas `null`. Il n'en est rien.

Dans le même (dés)ordre d'idées, le prédicat suivant vaut `unknown` et non `true` lorsque l'expression `CAT` vaut `null` :

```
(CAT < 'C1') or (CAT = 'C1') or (CAT > 'C1')
```

d) Sommes et valeurs null

En présence de valeurs `null`, la somme n'est pas distributive. Autrement dit, `sum(A + B)` n'est pas nécessairement égal à `sum(A) + sum(B)`.

Considérons la table `ACTIVITE` comportant les deux colonnes facultatives `H_COURS` et `H_TP`, représentant le nombre d'heures de cours et de travaux pratiques consacrées à chaque activité. On convient de représenter l'absence de cours par `H_COURS = null`, et de même pour l'absence de TP (figure 6.3).

21. A ces interprétations subtilement divergentes, certains SGBD ajoutent leur grain de sel. Par exemple, Oracle 7 assimile une chaîne de caractères vide à l'absence de valeur (`null`). Cette erreur a été corrigée dans les versions ultérieures.

Au lecteur perplexe quant à la subtile différence entre ces deux situations, on fera observer que, par exemple, la fonction `len`, qui donne le nombre de caractères d'une valeur, donnera 0 dans le cas d'une chaîne vide et `null` dans le cas d'une valeur absente.

ACTIVITE				
CODE_ACTIV	INTITULE	TITULAIRE	(H_COURS)	(H_TP)
INFO 1232	Java	PHE	30	15
INFO 1241	Labo prog.	PHE		45
INFO 2101	BD	JLH	30	
INFO 2111	Projet qualité	NHA		30
INFO 2120	Modélisation	JLH	20	10
INFO 2213	Conception	VEN	45	
INFO 2214	Mise en œuvre	VEN	60	
INFO 2231	Labo gestion	NHA		45

Figure 6.3 - Table d'activités

On désire calculer la charge de chaque titulaire. Deux expressions sont possibles :

```
select TITULAIRE, sum(H_COURS) + sum(H_TP) as CHARGE
from   ACTIVITE
group by TITULAIRE
```

et

```
select TITULAIRE, sum(H_COURS + H_TP) as CHARGE
from   ACTIVITE
group by TITULAIRE
```

Ces deux expressions sont en principe équivalentes. Or, non seulement elles produisent des résultats différents, mais de plus ceux-ci sont inexacts, du moins selon l'idée qu'on se fait de la charge d'une personne :

TITULAIRE	CHARGE	TITULAIRE	CHARGE
JLH	60	JLH	30
NHA		NHA	
PHE	90	PHE	45
VEN		VEN	

Il est inutile de soupçonner une quelconque erreur de calcul, SQL applique fidèlement les règles d'évaluation décrites ci-dessus.

De même, pour un prédicat P quelconque, les deux expressions ci-dessous ne donnent pas nécessairement le même résultat, même si P est purement binaire (ne peut être évalué qu'à true ou false) :

```
(select sum(H_COURS) from ACTIVITE)

(select sum(H_COURS) from ACTIVITE where P)
+ (select sum(H_COURS) from ACTIVITE where not P)
```

e) Fonction agrégative sur un ensemble vide

La règle qui stipule que la fonction `sum` renvoie `null` pour un ensemble vide est parfois contraire à l'intuition, notamment dans le domaine numérique. Ainsi, la *somme des montants des commandes des clients qui n'ont pas de commandes* vaudra `null` et non 0. Si cette somme est retirée de `COMPTE`, la valeur de cette dernière colonne sera détruite chez les clients qui n'ont pas commandé, ce qui exige de traiter séparément ce cas particulier, comme on l'a montré à la section 5.8.3.

f) Les valeurs null sont timides : elles disparaissent dès qu'on s'y intéresse

Il n'est pas nécessaire d'étudier des requêtes très complexes pour rencontrer des problèmes d'interprétation. La requête suivante donne un résultat sans surprise :

```
select CAT from CLIENT where LOCALITE = 'Toulouse'
```

CAT
B1
B1
<null>
B2
<null>

Si nous voulions éliminer les valeurs de `CAT` se terminant par 1, nous serions tentés d'écrire :

```
select CAT from CLIENT where LOCALITE = 'Toulouse'
and CAT not like '_1';
```

CAT
B2

... faisant ainsi disparaître des lignes qui ne semblaient pas concernées par la condition supplémentaire. Cette disparition des valeurs `null` est aisée à expliquer : lors de l'évaluation de la condition `CAT not like '_1'` elles rendent celle-ci `unknown` par leur simple présence et donc disparaissent.

g) En revanche, on en trouve là où on ne les attend pas

Nous donnerons encore un exemple d'interprétation particulièrement problématique du prédicat `exists` en présence de valeurs `null`. Recherchons les clients dont la catégorie est la plus élevée. On décide de formuler la requête comme suit : on recherche les clients pour lesquels *il n'existe pas de clients dont la catégorie soit supérieure*. Il vient :

```
select NCLI, CAT
from CLIENT C1
where not exists (select *
```

```

from    CLIENT C2
where   C2.CAT > C1.CAT);

```

NCLI	CAT
K729	<null>
D063	<null>
F400	C2

Ce résultat en apparence absurde est rigoureusement conforme aux règles d'interprétation de SQL. Considérons la ligne K729. Sa valeur de CAT étant null, la condition `C2.CAT > C1.CAT` vaut unknown et l'ensemble entre parenthèses est vide. Le prédicat `not exists` est évalué à true, et la ligne est retenue !

6.10.7 Deux recommandations

En raison des multiples difficultés qu'entraîne l'usage de la valeur null, certains auteurs recommandent de les éviter purement et simplement²². Citons deux techniques qui permettent de les contourner si on l'estime nécessaire :

1. détachement des colonnes facultatives

La table est décomposée de manière à extraire toute colonne facultative (ou groupe de colonnes simultanément facultatives) sous la forme d'une table autonome. C'est ainsi que la table CLIENT pourrait être remplacée par les deux tables suivantes qui ne comportent plus que des colonnes not null :

```

create table CLIENT (NCLI .. not null primary key,
                    NOM .. not null,
                    ADRESSE .. not null,
                    LOCALITE .. not null,
                    COMPTE .. not null);

create table CLICAT( NCLI .. not null primary key,
                    CAT char(2) not null,
                    foreign key (NCLI) reference CLIENT);

```

2. utilisation de valeurs par défaut

La colonne facultative est déclarée not null, mais est accompagnée d'une valeur par défaut. La prise en compte de cette valeur est à charge de l'utilisateur.

```

create table CLIENT (NCLI .. not null primary key,
                    NOM .. not null,
                    ADRESSE .. not null,
                    LOCALITE .. not null,
                    CAT char(2) default 'A0' not null,
                    COMPTE .. not null);

```

22. Citons par exemple les références [Codd, 1989], [Date, 1992], et [Date, 1997], auxquelles nous avons emprunté certains exemples de cette section.

Dans l'exemple de la table *ACTIVITE*, on rendra obligatoires les colonnes des heures, et on conviendra d'utiliser la valeur 0 pour indiquer l'absence d'une partie d'une activité.

Ces techniques compliquent la tâche des utilisateurs, mais peuvent rendre la base de données plus déterministe et donc son usage plus fiable. Ajoutons encore qu'en présence de valeurs *null*, il est prudent de distinguer dans les requêtes SQL les cas sans et avec valeurs *null*. Même si la norme définit l'interprétation d'une requête, il n'est pas certain que le SGBD l'adoptera dans tous les cas limites.

Notons enfin qu'éviter les colonnes facultatives n'empêchent malheureusement l'apparition de valeurs *null* dans le résultat de certaines requêtes, ni celle de la valeur de vérité *unknown* dans les conditions. Nous en avons rencontré plusieurs exemples.

Face à ces critiques, les auteurs du langage SQL avancent des arguments de bon sens qu'on retrouvera par exemple dans [Chamberlin, 1996].

6.11 EXERCICES

6.11.1 Contrôle d'accès

- 6.1 Développer, à l'aide des concepts étudiés dans cet ouvrage, un mécanisme qui permet de limiter l'accès à une table par un utilisateur déterminé à des moments bien déterminés, par exemple du lundi au vendredi, de 9h à 17h.

6.11.2 Le catalogue

- 6.2 Ecrire une requête qui donne, pour chaque table, le nombre de colonnes et la longueur maximum des lignes.
- 6.3 Le catalogue que nous avons décrit (figures 6.1 et 6.2) diffère quelque peu des catalogues proposés par les SGBD. En particulier, les catalogues réels ont souvent une structure complexe destinée à améliorer les performances. Ecrire les requêtes qui garnissent les tables de la figure 6.2 à partir du contenu des tables du catalogue de votre SGBD. Pour fixer les idées, on pourra s'exercer sur la structure ci-dessous, limitée à la représentation des *clés* (identifiants et clés étrangères) :

Une ligne de cette table représente un composant d'une *clé*. Elle indique successivement le nom de la table de la *clé*, le type de la clé (*Primary*, *Foreign*), son identifiant interne et le nom de la colonne; s'il s'agit d'une clé étrangère, elle indique en outre le nom de la table et de la colonne cibles.

SYS_KEY_COL					
TNAME	KTYPE	KEYID	CNAME	TARGETAB	TARGETCOL
CLIENT	P	K1	NCLI		
COMMANDE	P	K2	NCOM		
COMMANDE	F	K3	NCLI	CLIENT	NCLI
PRODUIT	P	K7	NPRO		
DETAIL	P	K4	NCOM		
DETAIL	P	K4	NPRO		
DETAIL	F	K5	NCOM	COMMANDE	NCOM
DETAIL	F	K6	NPRO	PRODUIT	NPRO

- 6.4 Le lecteur attentif aura observé que la table SYS_KEY_COL de l'exercice 6.3 n'est pas normalisée. On l'invite à repérer les dépendances fonctionnelles anormales puis à décomposer la table de manière à obtenir des fragments normalisés.

Chapitre 7

Applications avancées en SQL

Il existe des domaines de problèmes auxquels les bases de données peuvent apporter des solutions simples, originales et efficaces. Ce chapitre introduit le lecteur à quatre de ces problématiques : les structures d'ordre, les bases de données actives, les données temporelles et la production automatisée de code.

Les requêtes SQL que nous avons développées dans les derniers chapitres ne couvrent pas, loin s'en faut, le champ d'application de ce langage. C'est ce que nous illustrerons en présentant très brièvement (chaque thème mériterait un ouvrage complet) quelques éléments de quatre domaines d'actualité auxquels le lecteur motivé risque d'être confronté tôt ou tard : les structures d'ordre, les bases de données actives, les données temporelles et la production automatisée de code.

7.1 LES STRUCTURES D'ORDRE

Par construction, il n'existe aucun ordre prédéfini parmi les lignes d'une table. De même, le résultat de l'évaluation d'une requête se présente sans ordre significatif, à moins qu'un ordre n'ait été explicitement demandé par la clause `order by`. Il est cependant possible de rendre explicite, sous la forme d'une table à deux composants, une relation d'ordre dont une table, réelle ou résultant de l'exécution d'une requête, est le siège. Bien que l'exemple support de l'exposé ne s'y prête pas idéa-

lement, nous pouvons l'utiliser pour illustrer ce principe. Nous donnerons ensuite un autre exemple, sans doute plus convaincant.

Construisons une table qui à chaque numéro de client X associe le numéro de client Y qui le suit directement. On représente donc la *relation d'ordre strict* siégeant parmi les valeurs de NCLI. Il existe plusieurs manières d'exprimer cette relation. Selon la première, le client Y a le numéro le plus petit de ceux qui sont supérieurs à celui de X, ce qui se traduit comme suit (X est représenté par l'alias *prec* et Y par *suiv*) :

```
select prec.NCLI,suiv.NCLI
from   CLIENT prec, CLIENT suiv
where  suiv.NCLI = (select min(NCLI)
                    from   CLIENT
                    where  NCLI > prec.NCLI)

order by prec.NCLI
```

Une deuxième formulation consiste à rechercher les couples (X,Y) tels qu'il n'existe pas de clients dont le numéro soit strictement compris entre ceux de X et Y¹.

```
select prec.NCLI,suiv.NCLI
from   CLIENT prec, CLIENT suiv
where  not exists (select *
                  from   CLIENT
                  where  NCLI > prec.NCLI
                  and    NCLI < suiv.NCLI)

order by prec.NCLI
```

Ces requêtes donnent le résultat suivant :

prec.NCLI	suiv.NCLI
B062	B112
B112	B332
B332	B512
B512	C003
.
K729	L422
L422	S127
S127	S712

Dans cette même catégorie de problèmes on trouvera les questions relatives aux éléments dont le rang est compris entre I et J selon un ordre défini. Par exemple, recherchons les 3 derniers clients, par ordre de NCLI croissant, c'est-à-dire les

1. Optimisation possible : on impose en outre dans la clause *where* : *prec < suiv*, afin d'éviter l'examen de configurations stériles.

clients pour lesquels il y a moins de trois clients ayant un NCLI supérieur. Il s'agit des clients L422, S127 et S712.

```
select NCLI
from   CLIENT prec
where  (select count(*)
        from   CLIENT
        where  NCLI > prec.NCLI) < 3
order by NCLI
```

Proposons un exemple plus représentatif. On dispose d'une table BOURSE enregistrant l'évolution journalière d'un titre en bourse (figure 7.1).

BOURSE	
DATEV	VALEUR
12-09-2005	1250
13-09-2005	1245
14-09-2005	1242
15-09-2005	1244
16-09-2005	1251
17-09-2005	1254
18-09-2005	1259
19-09-2005	1255

Figure 7.1 - Table représentant l'évolution d'une valeur boursière

Nous voudrions extraire de cette table, pour chaque date (sauf bien sûr la première), la différence de valeur du titre par rapport à celle de la veille.

Ces différences pourraient s'obtenir comme suit.

```
select courant.DATEV as Date,
       courant.VALEUR - hier.VALEUR as Ecart
from   BOURSE hier, BOURSE courant
where  courant.DATEV = (select min DATEV
                        from   BOURSE
                        where  DATEV > hier.DATEV)
order by courant.DATE
```

Ce qui donne :

Date	Ecart
13-09-2005	-5
14-09-2005	-3
15-09-2005	2
16-09-2005	7
17-09-2005	3
18-09-2005	5
19-09-2005	-4

Le lecteur pourra trouver dans [Celko, 2000] un traitement plus détaillé des structures d'ordre.

7.2 LES BASES DE DONNÉES ACTIVES

Les mécanismes des prédicats, des procédures SQL et des déclencheurs permettent d'incorporer dans la base de données elle-même des composants actifs qui d'ordinaire sont inclus dans les programmes d'application. De telles bases de données sont souvent appelées *actives*, *réactives* ou *intelligentes*.

Ces composants permettent par exemple de coder des contraintes d'intégrité complexes, de définir des comportements particuliers en cas de violation de contraintes, de contrôler la redondance ou de traduire des lois de comportement du domaine d'application en comportement des données. Examinons quelques exemples d'applications.

7.2.1 Les contraintes d'intégrité statiques

Une contrainte d'intégrité statique précise une propriété que les données doivent vérifier à tout instant. Les contraintes d'unicité, référentielles et de colonnes obligatoires en sont les exemples les plus représentatifs, et d'ailleurs gérés automatiquement par la plupart des SGBD. Supposons qu'une commande ne puisse comporter plus de 5 détails. Nous pourrions vérifier cette contrainte par le déclencheur suivant.

```
create trigger MAX-5-DET
before insert or update NCOM on DETAIL
for each row
begin
    if (select COUNT(*) from DETAIL where NCOM = new.NCOM) = 5;
    then abort(); end if;
end;
```

L'opération `abort` a pour effet d'annuler l'action qui provoque l'événement, ici un `delete`.

7.2.2 Les contraintes d'intégrité dynamiques

En revanche, une contrainte dynamique indique quels changements d'états sont valides. On ne peut détecter une violation lors d'une modification qu'en connaissant les deux états avant et après l'événement. Ces deux états peuvent respecter toutes les contraintes statiques, alors que le passage de l'un à l'autre est illégal. Nous en reparlerons en 9.5.2.

Admettons qu'on n'autorise la suppression d'un client que s'il n'a plus envoyé de commandes depuis le 1er janvier 2005. Cette règle est prise en charge par le déclencheur ci-dessous.

```

create trigger SUP_CLIENT
before delete on CLIENT
for each row
begin
    if (select count(*) from COMMANDE
        where NCLI = old.NCLI and DATECOM > '1-1-2005') > 0
    then abort(); end if;
end;

```

Dans l'exemple suivant, le déclencheur protège les produits contre toute augmentation de prix qui dépasserait 5%.

```

create trigger MAJOR
before update of PRIX on PRODUIT
for each row
begin
    if new.PRIX > (old.PRIX * 1.05)
    then abort(); end if;
end;

```

7.2.3 Le contrôle de la redondance

En principe, une base de données ne devrait pas contenir de données redondantes, c'est-à-dire dont la valeur est calculable à partir d'autres données existantes. Si de telles données devaient malgré ce principe être introduites, alors il serait nécessaire de prévoir les mécanismes de gestion de cette redondance de manière à garantir leur intégrité. Admettons par exemple que nous ayons ajouté à la table DETAIL une nouvelle colonne de nom MONTANT, dont la valeur s'obtient par définition en multipliant la quantité commandée (QCOM) par le prix unitaire (PRIX) du produit correspondant. Les valeurs de cette colonne sont manifestement des données redondantes, qu'on peut tolérer à la condition qu'elles soient gérées de manière automatique. Tentons une analyse sommaire afin de déterminer les déclencheurs nécessaires.

Soit D une ligne de DETAIL, P la ligne de PRODUIT telle que P.NPRO = D.NPRO, D.QCOM la quantité commandée et P.PRIX le prix unitaire du produit. La redondance s'exprime par la relation suivante, qui doit être vérifiée à tout instant

$$D.MONTANT = D.QCOM * P.PRIX$$

Quels sont les événements (opérations de modification) qui sont susceptibles d'entraîner une violation de cette contrainte, et quelle est la réaction adéquate ? Il en existe quatre :

- | | |
|----------------------------|--------------------------------------|
| • insert into DETAIL; | <i>réaction</i> : calculer MONTANT |
| • update DETAIL set QCOM; | <i>réaction</i> : recalculer MONTANT |
| • update DETAIL set NPRO; | <i>réaction</i> : recalculer MONTANT |
| • update PRODUIT set PRIX; | <i>réaction</i> : recalculer MONTANT |

Toutes les autres opérations sont sans effet sur la relation exprimant la redondance. Nous devons pour chacun de ces événements écrire un déclencheur qui actualise la valeur de MONTANT. Celui qui correspond au premier se présenterait comme suit :

```
create trigger MONTANT_INS_DET
after insert on DETAIL
for each row
declare P decimal(6)
begin
    select PRIX into :P from PRODUIT
    where  NPRO = new.NPRO;
    update DETAIL
    set     MONTANT = QCOM * :P
    where  NCOM = new.NCOM and NPRO = new.NPRO;
end;
```

ou encore, plus simplement, sous la forme suivante :

```
create trigger MONTANT_INS_DET
before insert on DETAIL
for each row
declare P decimal(6)
begin
    select PRIX into :P from PRODUIT
    where  NPRO = new.NPRO;
    new.MONTANT = new.QCOM * :P;
end;
```

7.2.4 Les alerteurs

Un alerteur est un mécanisme qui envoie automatiquement un message dès qu'une situation déterminée se présente dans le domaine d'application. Il s'agit de situations requérant une intervention de la part d'un agent extérieur, humain ou programmé. Tel serait le cas d'un comportement dangereux dans une centrale électrique, de mouvements bancaires douteux de la part d'un client ou d'une demande d'un produit qui dépasserait la normale. Si la situation, ou l'événement qui entraîne celle-ci, peuvent se détecter par un état particulier des données, ou par une transition d'états, alors il est possible de développer des alerteurs sous la forme de déclencheurs.

Dans l'exemple ci-dessous, le passage sous zéro du stock d'un produit constitue un événement dont le gestionnaire des approvisionnements doit être averti le plus rapidement possible. Il existe plusieurs réactions possibles dans ce type de problèmes. Si le langage procédural des déclencheurs permet l'invocation de programmes externes, alors on pourra envoyer un courriel au gestionnaire l'avertissant du problème. Dans le cas contraire, on pourra utiliser la technique suivante. Le déclencheur écrit dans une table ALERTE un enregistrement décrivant la situation. A intervalle régulier, toutes les 10 secondes par exemple, une petite procédure consulte cette table, y repère les enregistrements non encore traités, pour chacun

d'eux, envoie le courriel et marque l'enregistrement comme étant traité. La table ALERTE comporte trois colonnes : TRAITE, qui indique par un caractère '*' ou un espace si l'enregistrement a ou non été traité, INSTANT donnant l'instant de la survenance de l'événement et NPRO reprenant le produit en rupture de stock. Le déclencheur doit s'activer lorsque la valeur de QSTOCK passe sous la barre de zéro, et non quand elle est inférieure à zéro. On pourrait donc écrire :

```
create trigger RUPT_STOCK
after update of QSTOCK on PRODUIT
for each row
when (old.QSTOCK >= 0 and new.QSTOCK < 0)
begin
    insert into ALERTE values (' ',current_timestamp,old.NPRO);
end;
```

7.2.5 Personnalisation des comportements standard

Le mécanisme des déclencheurs permet aussi de définir de manière personnalisée la réaction du SGBD face à une tentative de violation d'une contrainte.

Considérons la clé étrangère NCLI de COMMANDE qui référence CLIENT. On décide de ne pas déclarer cette clé étrangère par une clause *foreign key*, mais de la gérer explicitement via des déclencheurs attachés aux tables CLIENT et COMMANDE. La requête suivante définit la réaction à adopter lors de l'*insertion d'une ligne de COMMANDE à laquelle ne correspond aucune ligne de CLIENT*. Il s'agit d'un comportement du type **correctif**, qui ne s'oppose pas à l'opération litigieuse, mais qui corrige les données de manière à les rendre conformes à la contrainte. En cas de problème, le déclencheur ajoute une ligne décrivant (très sommairement !) le client manquant.

```
create trigger MAJ_CLIENT
after insert on COMMANDE
for each row
begin
    if new.NCLI not in (select NCLI from CLIENT)
    then insert into CLIENT (NCLI,NOM,ADRESSE,LOCALITE,COMPTE)
        values (new.NCLI,'?', '?', '?', 0)
    end if;
end;
```

7.2.6 Intégration d'une règle de gestion dans la base de données

L'exemple de la section 6.6 traduisait une règle de gestion de l'entreprise. Il est effectivement tentant d'introduire dans la base de données des opérations de gestion qui traditionnellement sont réalisées par les programmes d'application eux-mêmes.

Le déclencheur suivant provoque la mise à jour du compte du client dès qu'un nouveau détail est introduit. Il calcule le montant de ce détail et soustrait cette valeur du compte du client concerné. D'autres déclencheurs devraient prendre en compte

les autres modifications susceptibles d'affecter la valeur de la colonne COMPTE. Leur rédaction est laissée à l'initiative du lecteur.

```
create trigger MAJC
after insert on DETAIL
for each row
begin
    update CLIENT
    set COMPTE = COMPTE -
        (new.QCOM*(select PRIX
                    from   PRODUIT
                    where  NPRO = new.NPRO))
    where CLIENT.NCLI in
        (select NCLI
         from   COMMANDE
         where  COMMANDE.NCOM = new.NCOM) ;
end
```

Pour en savoir plus

Le lecteur intéressé trouvera dans [Patton, 1999] et [Ceri, 1997] des matériaux plus détaillés en matière de bases de données actives.

7.3 LES DONNÉES TEMPORELLES

Le temps est une dimension fondamentale dans de nombreux domaines d'application. Aussi la plupart des bases de données comprendront-elles des structures de données représentant le temps. Au delà des colonnes décrivant la survenance d'événements tels que naissances, livraisons, engagements ou dépenses (par exemple DATECOM dans COMMANDE), nous dirons quelques mots sur les données historiques, qui constituent le cas de figure le plus fréquent de données temporelles.

7.3.1 Représentation des données temporelles

L'un des modes de stockage habituels est illustré à la figure 7.2.²

On y représente les états successifs de quelques clients, pour un nombre limité de colonnes. Pour chaque état, on indique les valeurs des colonnes ainsi que la période durant laquelle ces valeurs étaient courantes. On notera que les périodes [début, fin] sont ouvertes à droite, la borne supérieure étant exclue³. L'état courant actuel d'un client est caractérisé par *fin = futur infini*, ici conventionnellement représenté par une date raisonnablement inaccessible⁴ (1-01-3000). Les clients encore actifs sont

2. Le script de création et d'exploitation de cette base de données est disponible sur le site de l'ouvrage.

3. Cette convention est choisie pour des raisons techniques : simplicité et efficacité de certaines requêtes et indépendance de l'unité de temps.

donc ceux qui ont un état courant à l'heure actuelle, c'est-à-dire tel que `fin = '1-01-3000'`. L'identifiant primaire de cette table est (NCLI, debut).

H_CLIENT					
NCLI	debut	fin	NOM	LOCALITE	CAT
C400	03-11-1998	18-05-2000	FERARD	Poitiers	B1
C400	18-05-2000	26-12-2000	FERARD	Poitiers	B2
C400	26-12-2000	07-04-2001	FERARD	Paris	B2
C400	07-04-2001	01-01-3000	FERARD	Paris	C1
F011	08-10-1997	12-03-2000	PONCELET	Lille	B1
F011	12-03-2000	15-03-2001	PONCELET	Paris	B2
F011	15-03-2001	22-07-2001	PONCELET	Toulouse	B1

Figure 7.2 - Table décrivant l'historique de deux clients

7.3.2 Interrogation de données temporelles

Certaines requêtes s'expriment très simplement. Ainsi, la recherche de la localité du client C400 en date du 18-05-2000 s'obtient par⁵ :

```
select NCLI, LOCALITE
from   H_CLIENT
where  NCLI = 'C400'
and    debut <= '18-05-2000' and '18-05-2000' < fin
```

L'état courant des clients est aussi facile à extraire :

```
select NCLI, NOM, LOCALITE, CAT
from   H_CLIENT
where  fin = '1-01-3000'
```

... de même que l'état courant de la table à la date 15-12-1998 :

```
select NCLI, NOM, LOCALITE, CAT
from   H_CLIENT
where  debut <= '15-12-1998' and '15-12-1998' < fin
```

Malheureusement, certaines requêtes, pourtant très fréquentes, exigeront une formulation beaucoup plus complexe. Nous en citerons deux : la projection et la jointure.

4. Technique sans doute peu élégante, mais qui permet de traiter le *futur infini* comme une date valide, ce qui simplifie considérablement certaines requêtes courantes. La théorie suggère plutôt l'usage d'une valeur spéciale *until_changed*, qui n'est cependant pas disponible dans les SGBD actuels. Quoi qu'il en soit, on peut supposer que cet ouvrage sera périmé à cette date.

5. Pourquoi n'a-t-on pas utilisé la condition *between* ?

7.3.3 La projection⁶ temporelle

Supposons que nous ne soyons intéressés que par l'évolution de la catégorie des clients. Une première formulation peut être proposée :

```
select NCLI, debut, fin, CAT
from   H_CLIENT
```

<incorrect>

... mais le résultat est inadéquat (les lignes 2 et 3, représentant une seule période, devraient être fusionnées) :

NCLI	debut	fin	CAT
C400	03-11-1998	18-05-2000	B1
C400	18-05-2000	26-12-2000	B2
C400	26-12-2000	07-04-2001	B2
C400	07-04-2001	01-01-3000	C1
F011	08-10-1997	12-03-2000	B1
F011	12-03-2000	15-03-2001	B2
F011	15-03-2001	22-07-2001	B1

Pour fusionner les périodes consécutives de même valeur de CAT pour le même client⁷, on pourrait écrire ceci :

```
select NCLI, min(debut), max(fin), CAT
from   H_CLIENT
group by NCLI, CAT
```

<incorrect>

NCLI	debut	fin	CAT
C400	03-11-1998	18-05-2000	B1
C400	18-05-2000	07-04-2001	B2
C400	07-04-2001	01-01-3000	C1
F011	08-10-1997	22-07-2001	B1
F011	12-03-2000	15-03-2001	B2

On voit clairement que si on résout le problème ci-dessus (ligne 2), on en introduit un autre : l'état <F011, 12-03-2000, 15-03-2001, B2> apparaît comme étant *inclus* dans <F011, 8-10-1997, 22-07-2001, B1>, ce qui est manifestement incorrect, les états d'un client devant être consécutifs.

Le concept clé qui nous permettra de résoudre le problème est celui de *suite maximale, pour un même client, d'états consécutifs de même valeur de CAT*. Cette suite

6. On appelle projection d'une table l'opérateur par lequel seules certaines colonnes d'une table sont conservées. Considérant la table T(A, B, C, D), la formulation SQL de la projection de T sur A et B est la suivante : `select distinct A, B from T`.

7. Cet opérateur de fusion de périodes porte le nom anglais de *coalescing*.

est fusionnée pour former une ligne du résultat. La figure 7.3 nous permet de mieux comprendre ce concept. Une telle suite est délimitée par les deux états C1 et C2 du client concerné tels que :

- 1. C1 et C2 sont identiques (même valeur de CAT),
- 2. C1 est le premier de la suite, c'est-à-dire qu'il n'existe pas d'état C0 identique qui précéderait directement C1,
- 3. C2 est le dernier de la suite, c'est-à-dire qu'il n'existe pas d'état C3 identique qui suivrait directement C2,
- 4. les états du client entre C1 et C2 sont identiques à C1, c'est-à-dire qu'il n'existe pas d'état C12 entre C1 et C2 qui leur serait différent.

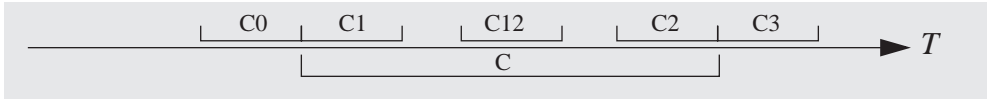


Figure 7.3 - Notion de suite maximale d'états identiques. Les états C1 à C2 sont fusionnés en un seul état maximal C

La traduction en SQL est la suivante.

```
select C1.NCLI, C1.debut, C2.fin, C1.CAT
from   H_CLIENT C1, H_CLIENT C2
where  C1.NCLI=C2.NCLI and C1.CAT=C2.CAT
and    C1.debut <= C2.debut
and not exists (select *
                 from   H_CLIENT C0
                 where   C0.NCLI = C1.NCLI
                        and C0.CAT = C1.CAT
                        and C0.fin = C1.debut)
and not exists (select *
                 from   H_CLIENT C3
                 where   C3.NCLI = C1.NCLI
                        and C3.CAT = C1.CAT
                        and C2.fin = C3.debut)
and not exists (select *
                 from   H_CLIENT C12
                 where   C12.NCLI = C1.NCLI
                        and C12.CAT <> C1.CAT
                        and C1.fin <= C12.debut
                        and C12.fin <= C2.debut)
```

NCLI	debut	fin	CAT
C400	03-11-1998	18-05-2000	B1
C400	18-05-2000	07-04-2001	B2
C400	07-04-2001	01-01-3000	C1
F011	08-10-1997	12-03-2000	B1
F011	12-03-2000	15-03-2001	B2
F011	15-03-2001	22-07-2001	B1

Cette requête est certainement la plus complexe de cet ouvrage. Il faut cependant savoir que la situation étudiée est une simplification du cas général, dans lequel la projection ne reprend pas nécessairement l'identifiant, ici NCLI, des entités dont on a enregistré l'historique (voir [Hainaut, 2001a]).

7.3.4 La jointure temporelle

Le concept de jointure que nous avons étudié est plus complexe lorsqu'il s'applique à des tables temporelles. Considérons qu'il existe une table qui reprend, pour chaque localité, le délégué commercial qui est responsable des clients de cette localité. Cette table pourrait se présenter comme à la figure 7.4.

H_LOCALITE			
LOCALITE	debut	fin	DELEGUE
Lille	01-01-1997	10-08-1999	Langlois
Lille	10-08-1999	24-07-2000	Vermont
Paris	23-12-1999	24-07-2000	Lagarde
Paris	24-07-2000	22-03-2001	Vermont
Paris	22-03-2001	01-01-3000	Langlois
Poitiers	01-01-1997	10-08-1999	Lagarde
Poitiers	10-08-1999	22-03-2001	Langlois
Poitiers	22-03-2001	01-01-3000	Lagarde
Toulouse	24-07-2000	22-03-2001	Lagarde
Toulouse	22-03-2001	01-01-3000	Vermont

Figure 7.4 - Historique des délégués commerciaux responsables des localités

Si nous désirons ajouter le nom des délégués aux informations sur les clients, il est évident que la requête de jointure classique est incorrecte :

```
select NCLI, C.debut, C.fin, C.LOCALITE, CAT, DELEGUE
from   H_CLIENT C, H_LOCALITE L
where  C.LOCALITE = L.LOCALITE                                <incorrect>
```

... et que la suivante l'est tout autant :

```
select NCLI, C.debut, C.fin, C.LOCALITE, CAT, DELEGUE
from   H_CLIENT C, H_LOCALITE L
where  C.LOCALITE = L.LOCALITE
and    C.debut = L.debut                                       <incorrect>
```

En effet, (LOCALITE,debut) n'est pas, au sens strict du terme, une clé étrangère de H_CLIENT vers H_LOCALITE, mais plutôt une *clé étrangère temporelle*, qui obéit à une loi plus complexe⁸ :

pour toute ligne d'état C de H_CLIENT dont LOCALITE n'est pas null, et pour tout instant t tel que $C.debut \leq t < C.fin$, il doit exister une ligne L de H_LOCALITE telle que $C.loc = L.loc$ et $L.debut \leq t < L.fin$.

Reprenons donc notre raisonnement : une ligne C de H_CLIENT et une ligne L de H_LOCALITE sont en correspondance si leurs périodes de validité sont non disjointes⁹, c'est à dire si¹⁰

$$(L.\text{debut} < C.\text{fin}) \quad \text{et} \quad (C.\text{debut} < L.\text{fin})$$

On peut écrire une première requête qui forme les couples de lignes de périodes de validité non disjointes :

```
select C.NCLI,C.debut,C.fin,L.LOCALITE,L.debut,L.fin
from   H_CLIENT C, H_LOCALITE L
where  C.LOCALITE = L.LOCALITE
and    (L.debut < C.fin) and (C.debut < L.fin)
```

Chacun des couples ainsi formés produit une ligne de la jointure dont la période de validité est calculée comme suit :

$$[\max(L.\text{debut}, C.\text{debut}), \min(L.\text{fin}, C.\text{fin}) [$$

La jointure temporelle s'exprime donc comme suit en SQL¹¹ :

```
select C.NCLI,L.LOCALITE,
       case
         when L.debut > C.debut then L.debut
         else C.debut
       end,
       case
         when L.fin < C.fin then L.fin
         else C.fin
       end,
       NOM,CAT,DELEGUE
from   H_CLIENT C, H_LOCALITE L
where  C.LOCALITE = L.LOCALITE
and    (L.debut < C.fin) and (C.debut < L.fin)
```

et produit le résultat :

8. Si, comme on le suppose ici, les états successifs de chaque objet (clients et localités) sont joints, alors cette loi s'exprime d'une manière plus simple : pour toute ligne C de H_CLIENT, il existe une ligne L1 de H_LOCALITE telle que $L1.\text{debut} \leq C.\text{debut}$ et il existe une ligne L2 de H_LOCALITE telle que $C.\text{fin} \leq L2.\text{fin}$.

9. Notons que SQL propose un prédicat overlaps qui indique si deux périodes temporelles contiennent des instants communs.

10. Cette relation s'obtient aisément lorsqu'on considère que les deux périodes sont disjointes *ssi* $C.\text{fin} \leq L.\text{debut}$ ou $C.\text{debut} \geq L.\text{fin}$. Elles sont donc non disjointes dans le cas contraire (section 5.2.6, règle r13).

11. Si la fonction case n'est pas disponible, il faudra se résoudre à une formulation plus complexe qui distingue quatre cas de recouvrement des intervalles. La requête prend alors la forme d'une union de quatre requêtes SFW. Cette version est disponible sur le site Web de l'ouvrage.

NCLI	debut	fin	NOM	LOCALITE	CAT	DELEGUE
C400	03-11-1998	10-08-1999	FERARD	Poitiers	B1	Lagarde
C400	10-08-1999	18-05-2000	FERARD	Poitiers	B1	Langlois
C400	18-05-2000	26-12-2000	FERARD	Poitiers	B2	Langlois
C400	26-12-2000	22-03-2001	FERARD	Paris	B2	Vermont
C400	22-03-2001	07-04-2001	FERARD	Paris	B2	Langlois
C400	07-04-2001	01-01-3000	FERARD	Paris	C1	Langlois
F011	08-10-1997	10-08-1999	PONCELET	Lille	B1	Langlois
F011	10-08-1999	12-03-2000	PONCELET	Lille	B1	Vermont
F011	12-03-2000	24-07-2000	PONCELET	Paris	B2	Lagarde
F011	24-07-2000	15-03-2001	PONCELET	Paris	B2	Vermont
F011	15-03-2001	22-03-2001	PONCELET	Toulouse	B1	Lagarde
F011	22-03-2001	22-07-2001	PONCELET	Toulouse	B1	Vermont

7.3.5 Gestion des données historiques

On conçoit aisément que la gestion d'une table telle que H_CLIENT demande le plus grand soin. L'insertion, la modification et la suppression de lignes doivent s'effectuer de manière que les propriétés illustrées ci-dessus soient respectées rigoureusement. Idéalement, l'utilisateur devrait se contenter d'effectuer des opérations correspondant directement aux événements du domaine d'application, sans se préoccuper des manipulations des informations historiques, qui devraient être effectuées automatiquement. C'est ce que nous allons proposer, pour le modèle de temps le plus simple, le temps physique (voir *Remarques* ci-après) par la technique des *déclencheurs*. Nous définissons d'abord la table des historiques :

```
create table H_CLIENT(
    NCLI      char(8)      not null,
    debut     timestamp not null,
    fin       timestamp default '1-01-3000' not null,
    NOM       char(18)     not null,
    LOCALITE  char(20)     not null,
    CAT       char(2),
    primary key (NCLI,debut));
```

... puis une vue qui présente les états courants :

```
create view CLIENT(NCLI,NOM,LOCALITE,CAT) as
select NCLI,NOM,LOCALITE,CAT
from   H_CLIENT
where  fin = '1-01-3000';
```

C'est sur cette vue, qui masque les colonnes temporelles *debut* et *fin*, que les opérations *insert*, *update* et *delete* vont être effectuées. Les déclencheurs vont compléter ces opérations pour garantir une gestion correcte des données historiques.

Lors d'une insertion dans *CLIENT*, on garnit la colonne *debut* **avant** l'enregistrement dans la table *H_CLIENT*. La colonne *fin* est mise à la valeur par défaut (futur infini) et peut être ignorée. Attention cependant à ne pas déclencher cette opération

lors d'insertions secondaires demandées par les *triggers* d'update et de delete. On observe que dans ces derniers cas, la valeur de fin désigne toujours un instant réel et non le futur infini 1-01-3000, d'où la condition de déclenchement when.

```
create trigger TRG_INSERT_CLI
before insert on H_CLIENT
for each row
when (new.fin='1-1-3000')
begin
    new.debut = current_timestamp;
end
```

La mise à jour d'une ou plusieurs colonnes doit se faire en deux temps, avant et après l'opération. Avant celle-ci, l'état courant actuel est transformé en nouvel état courant (nouvelle valeur de debut + exécution de l'update) :

```
create trigger TRG_B_UPDATE_CLI
before update on H_CLIENT
on each row
begin
    new.debut = current_timestamp;
end
```

Après l'update, il faut recréer l'état précédent qui vient d'être transformé, constitué des anciennes valeurs et de fin mise à l'instant présent :

```
create trigger TRG_A_UPDATE_CLI
after update on H_CLIENT
for each row
begin
    insert into H_CLIENT values (old.NCLI, old.debut,
        new.debut, old.NOM, old.LOCALITE, old.CAT);
end
```

Lors d'une suppression, qui efface effectivement l'état courant, il faut reconstituer celui-ci doté d'une valeur de fin représentant l'instant présent¹² :

```
create trigger TRG_DELETE_CLI
after delete on H_CLIENT
on each row
begin
    insert into H_CLIENT values (old.NCLI, old.debut,
        current_timestamp, old.NOM, old.LOCALITE, old.CAT);
end
```

12. On observe que la valeur de fin est empruntée à l'état précédent et non tirée du registre current_timestamp. La raison est simple : les deux valeurs doivent être identiques, alors que les valeurs du registre extraites par les deux déclencheurs pourraient différer de quelques millisecondes.

Si le résultat ne semble guère complexe¹³, son élaboration demande cependant un peu de soin, ainsi qu'en témoigne l'exercice 7.22.

Remarques

1. Une base de données classique, non temporelle, ne contient rien d'autre que l'état courant actuel du domaine d'application.
2. Nous n'avons considéré que les données historiques. Une base de données *réellement* temporelle peut également représenter des états futurs, c'est-à-dire des états dont *debut* est supérieur à l'instant présent ou qui se terminent dans un futur fini.
3. On peut envisager d'associer à tout état deux types de temps : le *temps logique*, qui indique pendant quelle période cet état était valide **dans le domaine d'application**, et le *temps physique* (ou *de transaction*), qui spécifie pendant quel laps de temps l'information a été courante **dans la base de données**. Considérons, pour distinguer ces deux concepts l'événement suivant : *l'employé C400 quitte Paris le 1-10-2005 pour s'établir à Nevers*. Nous prenons connaissance de ce fait le 15-10-2005 et nous l'enregistrons immédiatement dans la base de données. Le nouvel état de notre employé sera, si on considère le temps logique,

('C400', '1-10-2005', '1-01-3000', 'FERARD', 'Nevers', 'C1')

et, si on considère le temps physique,

('C400', '15-10-2005', '1-01-3000', 'FERARD', 'Nevers', 'C1')

Une table représentant les états selon ces deux types de temps comporte quatre colonnes temporelles; elle permet de consigner sans perte non seulement les changements d'état du domaine d'application, mais également la correction d'informations concernant les états passés. Une telle base de données, dite **bi-temporelle**, est particulièrement utile lorsqu'on veut reconstituer l'état de nos connaissances sur le domaine d'application à une date déterminée. On conçoit aisément que la gestion et le traitement de données bi-temporelles est beaucoup plus complexe que ce que nous avons étudié dans cette section, où nous nous sommes limités au temps physique.

4. Le modèle relationnel-objet permet de représenter de manière plus naturelle l'évolution individuelle de chaque colonne¹⁴. Les requêtes n'en sont cependant pas simplifiées, bien au contraire.
5. Remarquons enfin que la base de données que nous venons de décrire est un très bel exemple de *base de données active*.

13. Signalons tout de même que le traitement proposé est un peu simpliste. Trois exemples : (1) il est interdit de modifier l'identifiant primaire (à défaut de quoi il n'est plus possible de reconstituer l'historique d'un client), (2) lors d'un *update*, au moins une des valeurs doit avoir changé (sinon on crée deux états de mêmes valeurs, ce qui exige un *coalescing*), (3) l'intégrité référentielle doit être vérifiée.

14. L'une des techniques est la suivante : chaque colonne est représentée par un tableau de deux sous-colonnes reprenant une valeur et une période de validité.

Pour en savoir plus

Le domaine des bases de données temporelles a été largement exploré. En raison des déficiences des SGBD relationnels classiques, on a notamment proposé une extension du langage SQL (TSQL2) prenant en charge les aspects temporels, ainsi que des structures de données physiques améliorant les performances des principaux opérateurs de manipulation de données temporelles. Le lecteur intéressé consultera par exemple [Snodgrass, 2000], qui est sans doute l'ouvrage le plus complet à l'heure actuelle. Une étude de cas complète est disponible dans [Hainaut, 2001a]. Le lecteur y trouvera le développement en InterBase des principaux opérateurs de projection, coalescing, jointure et agrégation, à la fois sous forme prédicative et procédurale, ainsi que les déclencheurs de gestion des deux types de temps (validité et transaction).

7.4 LA GÉNÉRATION DE CODE

Les requêtes SFW, qui produisent des tables formées de lignes de données, peuvent tout aussi bien produire de simples lignes de texte, qui ne sont rien d'autre que des valeurs de type `character`. Rien n'empêche d'ailleurs que ces lignes représentent des instructions dans un langage informatique quelconque, ce qu'on appelle du *code*. La génération de code à l'aide de requêtes SQL, qui, nous allons le voir, ne pose pas de difficulté majeure, permet de comprendre des mécanismes et des processus extrêmement importants en informatique moderne, et notamment dans les applications web, dont un des mécanismes fondamentaux est la génération de pages HTML. Nous commencerons par une application très simple pour évoluer progressivement vers des exemples un peu plus exigeants mais plus intéressants.

7.4.1 Migration de données

Posons le problème comme suit : nous disposons d'une table de données (soit `DETAIL` pour les besoins de la discussion) gérée par un SGBD et nous désirons transmettre son contenu vers un autre SGBD. Il est inutile de préciser que les deux formats d'implémentation sont incompatibles. Le problème serait résolu si nous disposions des instructions `insert into` correspondant au contenu de la première table. En effet, le format de ces instructions est (raisonnablement) indépendant du SGBD.

La requête ci-dessous, soumise au SGBD source, construit des lignes qui forment de telles instructions. Il suffit de les rediriger vers un fichier texte et de les soumettre au SGBD cible¹⁵.

15. La quadruple apostrophe est en fait une chaîne formée de deux apostrophes, elles-mêmes délimitées, comme toute chaîne constante, par deux apostrophes. Dans ce contexte quelque peu apostrophant, une double apostrophe indique un caractère apostrophe et non deux délimiteurs. L'ensemble produit donc le caractère *apostrophe*. Dans certains SGBD tels que Access, on définira plutôt ce caractère par la fonction `chr$(39)`.

seul exemplaire et dépend de la table SYS_TABLE. Les fragments du deuxième type définissent chacun un constructeur de valeur qui dépend du type de la colonne correspondante; ils dérivent des informations de la table SYS_COLUMN. Le troisième et dernier fragment "|| ' ');' from **DETAIL**" représente la clôture la requête et dépend de la table SYS_TABLE. Nous produirons ces fragments au moyen de trois requêtes qui rangent leur résultat dans une table de travail comportant deux colonnes : SEQ qui définit le type de fragment (1, 2 ou 3) et LTEXTE qui contient le fragment lui-même.

```
insert into LIGNE
select 1, 'select ''insert into '||TNAME||' values(''
from SYS_TABLE
where TNAME = 'DETAIL';

insert into LIGNE
select 2,
       case CTYPE
       when 'char' then
           '||''''''''||trim('
           ||CNAME
           ||')||''''''''||','''
       when 'decimal' then
           '||trim(cast('
           ||CNAME
           ||' as char('
           ||cast(CLENGTH as char(4))
           ||'))),'
       else '*** type inconnu ***'
from SYS_COLUMN
where TNAME = 'DETAIL'
and T.TNAME = C.TNAME;

insert into LIGNE
select 3, '|| ' ');' from '||TNAME
from SYS_TABLE
where TNAME = 'DETAIL';
```

Il reste alors à reconstituer la requête de conversion en extrayant les fragments dans l'ordre des valeurs croissantes de SEQ :

```
select LTEXTE
from LIGNE
order by SEQ
```

Les lignes qui s'affichent sont redirigées vers un fichier de texte qui contiendra la liste des requêtes de génération des instructions de conversion pour toutes les tables concernées. On peut aisément simplifier la démarche par une procédure SQL qui exécute les requêtes ci-dessus pour chaque valeur de TNAME de SYS_TABLE.

Remarques

1. Rappelons que le procédé ne fonctionne que pour des colonnes déclarées `not null`. L'extension aux colonnes facultatives est suggérée dans l'exercice 7.31.
2. Le résultat ne sera en général guère satisfaisant car les valeurs apparaîtront dans un ordre aléatoire par rapport à celui des colonnes, ce qui risque, au mieux de provoquer des erreurs détectées par le SGBD, et au pire, de corrompre les données. On se référera à l'exercice 7.32 qui aborde cette question.
3. Les instructions `"insert into"` produites par les requêtes générées sont syntaxiquement incorrectes car chaque valeur, y compris la dernière, est suivie d'une virgule. L'exercice 7.33 propose de remédier à ce problème.
4. Si le SGBD l'autorise, il est possible de regrouper ces instructions en une seule requête, sans faire usage de la table `LIGNE`. Si on dénote par `Q1`, `Q2`, `Q3` les requêtes SFW des trois instructions `"insert into LIGNE"`, dans lesquelles les résultats ont reçu les alias `SEQ` et `LTEXTE`, on peut écrire :

```
select LTEXTE
from Q1 union Q2 union Q3
order by SEQ
```

7.4.3 Génération de définitions de bases de données

Nous savons que les tables du catalogue sont garnies lors de l'exécution des instructions de création de structures de données (`create table`, `alter table`, etc). A l'inverse, il devrait être possible de reconstituer ces instructions de création à partir du contenu des tables du catalogue. C'est ce que nous allons montrer à partir des tables décrites à la figure 3.5, que nous utiliserons pour produire la définition des tables et des colonnes. Le traitement des identifiants et des clés étrangères est laissé à l'initiative du lecteur à titre d'exercice. Pour produire les requêtes `create table`, nous devons travailler en plusieurs étapes. En effet, si on les dispose comme ci-dessous, on observe que ces requête comportent trois types de composants distincts, certains étant uniques, `"create table DETAIL(" et ") ;"`, et les autres multiples : les définitions des colonnes.

```
create table DETAIL (
    NCOM char(12) not null,
    NPRO char(15) not null,
    QCOM decimal(8) not null
);
```

A chacun de ces composants va correspondre une requête SFW qui va ranger son résultat dans une table intermédiaire `LIGNE`. Les lignes de `LIGNE` devant sortir dans un ordre déterminé pour former des instructions de création de table correctes, nous leur associerons deux codes : `NOMT` qui donne le nom de la table et `SEQ` qui indique la position de la ligne dans l'instruction `create table`. Le fragment d'instruction lui-même sera rangé dans la colonne `LTEXTE`.

```
create table LIGNE (NOMT char(24) not null,
                  SEQ decimal(2) not null,
                  LTEXTE varchar(80) not null);
```

Les requêtes SFW de création des fragments se présentent comme suit. Pour simplifier, on a limité des types de valeurs à char(n) et integer. On généralisera aisément aux autres types.

```
insert into LIGNE
select TNAME,1,'create table '||trim(TNAME)||'('
from   SYS_TABLE;

insert into LIGNE
select TNAME,9,');'
from   SYS_TABLE;

insert into LIGNE
select TNAME,2,'
          | CNAME
          | case CTYPE
            when 'int' then ' integer'
            when 'char'
              then ' char('
                  ||cast(CLENGTH as char(3))
                  ||')'
            else '** type invalide **'
            end
          | case NULLS
            when 'N' then ' not null' else ''
            end
          | ','
from   SYS_COLUMN
```

Il reste à reconstituer les instructions de création de tables à partir de ces fragments :

```
select LTEXTE
from   LIGNE
order by NOMT,SEQ;
```

Remarque

La définition de la dernière colonne se termine aussi par une virgule, ce qui ne pose pas de problème si on complète la procédure par la déclaration des identifiants et des clés étrangères (exercice 7.28).

7.4.4 Génération de pages HTML

La présentation du résultat d'une requête SFW est fonctionnelle mais n'a rien d'attrayant lorsqu'on la compare aux pages Web auxquelles nous sommes accou-

tumés. On pourrait par exemple afficher des informations relatives à un client de numéro déterminé selon le format de la figure 7.5.

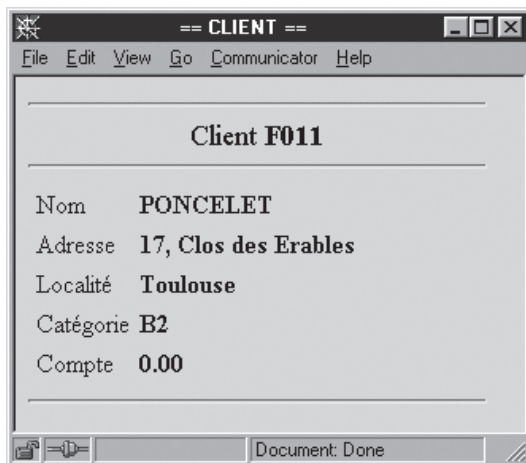


Figure 7.5 - Une présentation plus attrayante des informations sur un client

Cet écran serait défini par le code HTML suivant (les fragments variables correspondant à des données y ont été soulignés) :

```
<HTML>
  <TITLE>== CLIENT ==</TITLE>
  <BODY BGCOLOR="#DDDDDD">
    <HR>
    <CENTER><BIG>Client <B>&u>F011</B></BIG></CENTER>
    <HR>
    <TABLE>
      <TR ALIGN=LEFT><TD>Nom</TD>
        <TD><B>&u>PONCELET</B></TD></TR>
      <TR ALIGN=LEFT><TD>Adresse</TD>
        <TD><B>&u>17, Clos des Erables</B></TD></TR>
      <TR ALIGN=LEFT><TD>Localité</TD>
        <TD><B>&u>Toulouse</B></TD></TR>
      <TR ALIGN=LEFT><TD>Catégorie</TD>
        <TD><B>&u>B2</B></TD></TR>
      <TR ALIGN=LEFT><TD>Compte</TD>
        <TD><B>&u>0.00</B></TD></TR>
    </TABLE>
    <HR>
  </BODY>
</HTML>
```

Il est évidemment hors de question de rédiger ce texte manuellement. Il peut être facilement produit à partir du contenu de la table CLIENT par une requête SFW telle que la suivante, dont la sortie est à rediriger vers un fichier de texte HTML :

```

select 1,
    '<HTML>' || '<TITLE>== CLIENT ==</TITLE>'
    || '<BODY BGCOLOR="#DDDDDD"><HR>'
    || '<CENTER><BIG>Client <B>'
    || trim(NCLI)
    || '</B></BIG></CENTER>'
    || '<HR><TABLE>'
    || '<TR ALIGN=LEFT> <TD>Nom</TD><TD><B>'
    || trim(NOM)
    || '</B></TD></TR>'
    || '<TR ALIGN=LEFT> <TD>Adresse</TD><TD><B>'
    || trim(Adresse)
    || '</B></TD></TR>'
    || '<TR ALIGN=LEFT> <TD>Localité</TD><TD><B>'
    || trim(LOCALITE)
    || '</B></TD></TR>'
    || '<TR ALIGN=LEFT> <TD>Catégorie</TD><TD><B>'
    || trim(CAT)
    || '</B></TD></TR>'
    || '<TR ALIGN=LEFT> <TD>Compte</TD><TD><B>'
    || trim(COMPTE)
    || '</B></TD></TR>'
    || '</TABLE><HR></BODY></HTML>'
from CLIENT
where NCLI = 'F011';

```

La référence [Hainaut, 2001b] décrit une étude de cas développée en InterBase. En production, la génération dynamique de pages HTML se programme via des langages spécialisés tels que ASP, JSP ou PHP [Daspet, 2004].

7.4.5 Génération de documents XML

La maîtrise de la génération d'un langage de balises tel que HTML rend aisé le développement de composants similaires en XML. Pour chaque table, on proposera une DTD XML, puis on rédigera la suite de requêtes produisant, pour les lignes de la table, un document XML reprenant les informations de la table. Pour simplifier le problème, on adoptera un scénario de migration de données, ce qui nous permettra d'ignorer les contraintes d'intégrité telles que les identifiants et les clés étrangères. On consultera par exemple la référence [Michard, 2001].

7.4.6 Génération de générateurs de pages HTML ou de documents XML

Nous pourrions envisager de construire une séquence de requêtes qui produise des générateurs de pages HTML à partir des tables du catalogue. Le principe étant le même que pour le générateur de migrateurs, nous laisserons ce projet à l'initiative du lecteur.

De même, on peut envisager un générateur de DTD XML correspondant à l'ensemble des tables d'une base de données.

7.5 EXERCICES

7.5.1 Les structures d'ordre

- 7.1 Construire une table qui à chaque numéro de client, dont le compte est X, associe le numéro d'un autre client dont le compte Y suit directement X.

Suggestion. On construit la *relation d'ordre partiel* selon COMPTE.

- 7.2 Afficher les clients par couples, tels que le premier a un compte inférieur ou égal au second.

Suggestion. Il s'agit de la fermeture transitive de la relation d'ordre construite à la question 7.1. On peut cependant la construire de manière très simple.

- 7.3 Afficher les couples de PRODUITS consécutifs selon l'ordre croissant des valeurs de QSTOCK*PRIX.

- 7.4 On considère la table BOURSE de la figure 7.1. Rechercher les dates auxquelles le titre a gagné le plus de points par rapport à la veille.

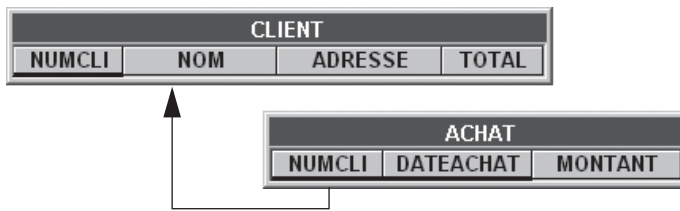
Suggestion. Cette requête serait élémentaire si elle pouvait s'exprimer sur la table des écarts.

- 7.5 On complète la table BOURSE par l'ajout d'une colonne TITRE qui représente un titre. De la sorte, cette table représente l'évolution journalière d'un ensemble de titres plutôt que d'un seul. Modifier les requêtes développées jusqu'ici sur la table BOURSE pour qu'elles prennent en compte cette extension.

- 7.6 Une table RELEVÉ, de colonnes CLIENT, MOIS et INDICE, d'identifiant (CLIENT, MOIS), reprend pour chaque client et pour chaque mois, le relevé de l'indice de consommation du compteur électrique. Garnir la table CONSOMMATION, de colonnes CLIENT, MOIS, KWH, de même identifiant que RELEVÉ, qui reprend la consommation électrique (nombre de kwh = différence entre deux index consécutifs) de chaque client pour chaque mois.

7.5.2 Les bases de données actives

- 7.7 On considère le schéma ci-dessous. La colonne TOTAL de chaque CLIENT représente, à tout instant, la somme des MONTANTS des ACHATs du CLIENT. Définir en SQL un schéma qui traduit ces structures et qui gère les valeurs de la colonne MONTANT via un jeu de déclencheurs. On répertoriera d'abord tous les événements susceptibles d'affecter la définition de cette colonne, puis on rédigera pour chacun le déclencheur correctif¹⁷.



- 7.8 On admet qu'à chaque produit correspond une page HTML qui en décrit les propriétés (exercice 7.29). Rédiger un composant SQL qui rafraîchit automatiquement la page d'un produit dont on vient de modifier une propriété.
- 7.9 A chaque création d'une ligne d'une table, on désire produire un message en XML qui reprend les données introduites. Proposer une DTD XML adéquate. Rédiger les composants qui créent automatiquement ces messages.
- 7.10 On ajoute à la table CLIENT une colonne ETAT_CIVIL facultative, dont les valeurs autorisées sont C, M, S, D, V, etc. (codant respectivement les états célibataire, marié, séparé, divorcé, veuf, etc.). Ecrire les composants SQL qui garantissent (1) que cette colonne contient à tout moment une valeur valide ou null, (2) que les modifications de valeurs respectent les transitions légales; par exemple, les transitions null → C, M → S ou S → D sont valides tandis que C → S ou V → S sont interdites.
- Suggestion.* La contrainte 1 est statique, et peut être représentée par des mécanismes élémentaires. La contrainte 2 est dynamique, et réclame la comparaison des valeurs avant et après modification, ce qui n'est possible que dans un déclencheur.
- 7.11 Si le montant d'une commande dépasse 1.000 alors que le compte du client est inférieur à 3.000, la commande est tronquée de manière que son montant ne dépasse pas le seuil de 1.000.
- Suggestion.* Cette validation peut avoir lieu pour chaque insertion d'un détail. Si le détail viole la règle, son insertion est annulée.
- 7.12 A tout instant, la table PRODUIT contient une ligne dont la valeur de LIBELLE est 'CADEAU'. Elle représente un article dont une unité est gratuitement ajoutée à toute commande dont le montant dépasse 2.000. Rédigez les composants SQL qui automatisent cette règle.
- Suggestion.* Le problème est moins simple qu'il n'y paraît, car la réaction est induite par l'insertion des lignes de DETAIL et non par celle de la ligne COMMANDE, ce qui serait prématuré. Si on utilise un déclencheur, celui-ci

peut ajouter le *cadeau* dès que le montant dépasse 2.000 et que le cadeau n'a pas encore été ajouté.

- 7.13 On ajoute à la table **COMMANDE** une colonne **MONTANT**, qui donne le montant total des détails de chaque commande. Ces valeurs sont évidemment calculables à partir des valeurs de **QCOM** de **DETAIL** et de **PRIX** de **PRODUIT**. Ecrire les déclencheurs qui gère les valeurs de cette nouvelle colonne.
- 7.14 Ajouter au schéma de la base de données 3.5 les éléments suivants : des colonnes **QREAPPRO**, **QCOM_MIN** et **QCOM** de **PRODUIT**, indiquant respectivement le seuil et la quantité minimum de réapprovisionnement, et la quantité commandée aux fournisseurs mais non encore livrée; une table **FOURNISSEUR**; une table **OFFRE** indiquant à quelles conditions un fournisseur peut livrer un produit : quantité minimum, prix unitaire fixe, frais de transport fixes; enfin une table **COM_FOURN** qui décrit les commandes de réapprovisionnement effectuées pour un produit et envoyées à un fournisseur. Lorsqu'un produit tombe en dessous de son seuil de réapprovisionnement, on calcule la quantité à commander, puis on recherche le fournisseur qui peut réapprovisionner au meilleur prix. On passe alors commande à ce fournisseur pour ce produit. Construire un système à base de déclencheur qui crée automatiquement les commandes de réapprovisionnement.

Suggestion. On admet qu'en fin de journée on lance une requête d'ajustement de **QSTOCK** de **PRODUIT** (voir 5.8.3). La modification de cette colonne déclenche une réaction de vérification de la quantité à commander éventuellement, de choix du fournisseur et de création d'une commande. Attention : on ne réagira que si **QSTOCK** + **QCOM** est trop faible. On peut d'ailleurs lancer une commande fournisseur alors que d'autres sont encore en attente.

7.5.3 Les données temporelles

- 7.15 On considère la table temporelle **H_CLIENT** de la figure 7.2. Rechercher les numéros des clients qui ne sont plus actifs.
- 7.16 On considère la table temporelle **H_CLIENT** de la figure 7.2. Rechercher, pour chaque client, la (les) localité(s) dans laquelle (lesquelles) ce client a été classé dans la catégorie de plus haut niveau dans son existence.
- 7.17 La table **COMMANDE** comprend les colonnes (**NCLI**,**DATECOM**) qui pourraient être considérées comme formant une clé étrangère temporelle vers la table **H_CLIENT**. Ecrire une requête qui affiche, pour chaque commande, le nom et la localité du client *au moment indiqué par DATECOM*.

- 7.18 Afficher pour chaque délégué et chaque localité, le nombre total de jours pendant lesquels il a été responsable de cette localité.

Suggestion. L'intervalle de temps entre deux dates est un entier qui se calcule par la différence entre ces dates.

- 7.19 Pendant quelles périodes les clients C400 et F011 ont-ils habité dans la même localité ? Indiquer ces localités.

Suggestion. On effectue une auto-jointure temporelle de H_CLIENT sur LOCALITE. Attention aux états jointifs de mêmes valeurs (*coalescing*).

- 7.20 Donner pour chaque localité les périodes durant lesquelles il y eu au moins un client.

- 7.21 Quels sont les délégués qui ont eu une interruption de carrière ? Indiquer les périodes d'interruption.

Suggestion. Il y a interruption lorsqu'il existe deux états E1 et E2 successifs qui ne sont pas jointifs ($E1.fin < E2.debut$).

- 7.22 Pour contrôler la gestion de la table H_CLIENT, on propose les déclencheurs suivants. Qu'en pensez-vous ?

Suggestion. En partant d'une table H_CLIENT contenant deux ou trois lignes d'historique, et d'une suite de quelques événements d'évolution des entités représentées, construisez soigneusement la trace de tous les événements et de toutes les opérations définies par ces déclencheurs. A tout instant, observer l'évolution de la table et vérifier si les contraintes d'intégrité sont respectées.

```
create trigger TRG_INSERT_CLI
before insert on H_CLIENT
for each row
begin
    /* insérer un état courant si l'entité est inconnue
       (= pas encore d'état pour cette entité) */
    if (not exists(select * from H_CLIENT
                   where NCLI= new.NCLI)) then begin
        new.debut = current_timestamp;
        new.fin = 1-01-3000
    end
end

create trigger TRG_B_UPDATE_CLI
before update on H_CLIENT
on each row
begin
    /* définir le nouvel état courant */
    new.debut = current_timestamp;
    new.fin = 1-1-3000;
    /* créer l'état précédent */
    insert into H_CLIENT values(old.NCLI, old.debut,
```

```

        new.debut, old.NOM, old.LOCALITE, old.CAT);
end

create trigger TRG_DELETE_CLI
after delete on H_CLIENT
on each row
begin
    insert into H_CLIENT values (old.NCLI, old.debut,
        current_timestamp, old.NOM, old.LOCALITE, old.CAT);
end

```

- 7.23 Compléter les déclencheurs de la section 7.3.5 de manière telle qu'ils gèrent la stabilité de l'identifiants d'entité (NCLI), la différence de deux états consécutifs et l'intégrité référentielle (LOCALITE vers H_LOCALITE).
- 7.24 Il existe d'autres représentations des états historiques d'une collection d'entités. L'une d'elles, bien que peu économique en place occupée, offre une grande simplicité de gestion et d'exploitation. Elle consiste à utiliser deux tables : CLIENT et H_CLIENT. La première contient uniquement les états courants tandis que la seconde contient tous les états, courants et passés. Elles sont définies comme suit :

```

create table CLIENT
(NCLI      char(8)    not null,
 NOM       char(18)   not null,
 LOCALITE  char(20)   not null,
 CAT       char(2),
 primary key (NCLI));

create table H_CLIENT
(NCLI      char(8)    not null,
 debut     timestamp  not null,
 fin       timestamp  not null,
 NOM       char(18)   not null,
 LOCALITE  char(20)   not null,
 CAT       char(2),
 primary key (NCLI,debut))

```

Rédiger les déclencheurs de gestion automatique de l'historique à partir des opérations courantes insert, update et delete effectuées sur la table CLIENT.

- 7.25 Soit une table EMPLOYE, d'identifiant MATRICULE, et qui donne pour chaque employé, son nom, sa période d'engagement (debut-fin) et le projet auquel il a été affecté. Pour simplifier, les dates sont représentées par des nombres entiers. Si un employé travaille encore aujourd'hui, la date de fin de période est mise à 999. Par exemple, la première ligne indique que l'employé

A237, de nom Antoine, travaille sur le projet BIOTECH depuis la date 50. Carlier y a travaillé entre 10 et 40, cette dernière date étant exclue.

EMPLOYEE				
MATRICULE	NOM	debut	fin	PROJET
A237	Antoine	50	999	BIOTECH
D107	Delecourt	50	999	SURVEYOR
G96	Godin	20	50	SURVEYOR
C45	Carlier	10	40	BIOTECH
N240	Nguyen	30	70	SURVEYOR
A68	Albert	50	999	BIOTECH
M158	Mercier	40	999	SURVEYOR
D122	Declercq	10	999	SURVEYOR

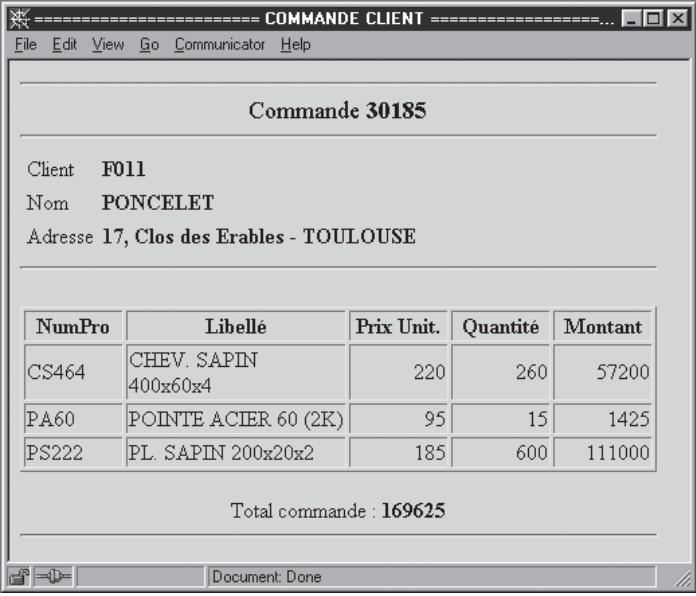
On demande de rédiger un script SQL qui donne, pour chaque projet, l'historique du nombre d'employés.

7.5.4 La génération de code

- 7.26 La section 7.4 nous a appris à écrire des requêtes de migration de données. Supposons que la base de données source est toujours celle de la figure 3.5, mais que la base de données cible est d'un format différent. Par exemple, la table DETAIL comporte, outre les colonnes NCOM, NPRO et QCOM, les colonnes NCLI (le numéro du client de la commande), LIBELLE (le libellé du produit commandé) et MONTANT (le montant du détail = QCOM x PRIX). Ecrire la requête qui crée les instructions de migration pour les détails relatifs aux produits en sapin.
- 7.27 Compléter la procédure de génération d'instructions de création de tables en l'étendant aux autres types de valeurs.
- 7.28 Compléter la procédure de génération d'instructions de création de tables en y incluant la définition des identifiants primaires et des clés étrangères.
Suggestions. La définition d'un identifiant primaire est elle aussi constituée d'un fragment unique, suivi de fragments multiples (les composants) eux-mêmes suivis d'un fragment unique. Une clé étrangère sera déclarée par un instruction alter table. La forme la plus élégante serait sans doute celle d'une procédure SQL (section 6.5) qui encapsule l'ensemble des requêtes de génération et qui permet une gestion algorithmique plus concise et plus lisible des différents cas particuliers (absence d'identifiant par exemple).
- 7.29 Ecrire un jeu de requêtes qui produise une DTD XML pour l'ensemble des table de la base de données. On ignorera les contraintes d'intégrité telles que les identifiants et les clés étrangères.

- 7.30 La requête de génération d'instructions de migration développée à la section 7.4 (*Migration de données*) fonctionne correctement lorsque la ligne source ne contient pas de valeurs null. Modifier cette requête de manière à prendre en compte la présence de valeurs null.
Suggestion. Remplacer la valeur extraite de la ligne source par la chaîne 'null' lorsque cette valeur est null (fonction coalesce par exemple).
- 7.31 Modifier la procédure de génération de requêtes de production d'instructions de migration de manière qu'elle produisent des instructions "insert into" correctes en présence de valeurs null (section 7.4, *Génération de migrateurs de données*).
- 7.32 Modifier la procédure de génération de requêtes de production d'instructions de migration de manière qu'elle produisent les valeurs dans l'ordre de déclaration des colonnes (section 7.4, *Génération de migrateurs de données*).
Suggestion. Il manque manifestement une information dans la table des colonnes : l'ordre de déclaration des colonnes dans leur table. Appelons CSEQ le numéro de séquence de la colonne. Il faut alors ajouter une colonne correspondante dans la table de travail LIGNE et l'ajouter au critère de tri (order by) dans la requête finale de reconstitution des requêtes à partir des fragments. *Remarque :* il faudrait également tenir compte de cet ordre dans la génération des instructions "create table".
- 7.33 Modifier la procédure de génération de requêtes de production d'instructions de migration de manière à éviter la présence d'une virgule derrière la dernière valeur (section 7.4, *Génération de migrateurs de données*).
Suggestion. N'insérer une virgule que pour les colonnes dont le numéro d'ordre (CSEQ) n'est pas maximal pour la table.
- 7.34 A partir des exercices de génération d'instructions de migration et de génération de migrateurs de données, développer des scripts SQL produisant des documents XML.
- 7.35 Les utilisateurs aimeraient visualiser les commandes au moyen de leur navigateur selon la présentation de la figure 7.6.
Rédiger la suite de requêtes SFW qui construit automatiquement la page HTML à partir du numéro de la commande.
Suggestions. Cette page étant constituée de fragments uniques et de fragments multiples, on s'inspirera utilement de la procédure de génération des instructions de création de tables. Ici encore on aura intérêt à construire une procédure SQL prenant le numéro de commande en argument¹⁸.

18. Une solution complète de cet exercice sous forme de procédures SQL en InterBase est disponible dans [Hainaut, 2001b].



The screenshot shows a window titled 'COMMANDE CLIENT' with a menu bar (File, Edit, View, Go, Communicator, Help). The main content area displays client information and a table of products. The client information is as follows:

Client	F011
Nom	PONCELET
Adresse	17, Clos des Erables - TOULOUSE

Below the client information is a table with 5 columns: NumPro, Libellé, Prix Unit., Quantité, and Montant. The table contains three rows of product data:

NumPro	Libellé	Prix Unit.	Quantité	Montant
CS464	CHEV. SAPIN 400x60x4	220	260	57200
PA60	POINTE ACIER 60 (2K)	95	15	1425
PS222	PL. SAPIN 200x20x2	185	600	111000

At the bottom of the window, the total order amount is displayed: 'Total commande : 169625'. The status bar at the very bottom shows 'Document: Done'.

Figure 7.6 - Une présentation élégante d'une commande

- 7.29 Poursuivre l'idée de la section 7.4, *Génération de pages HTML*, par la production d'un ensemble de pages décrivant chacune un client et de pages décrivant chacune un produit.

Compléter ensuite la page d'une commande en y ajoutant un lien vers le client et des liens vers chacun des produits référencés par les détails.

Suggestion. Chaque page sera stockée sous un nom qui contient le numéro du client ou du produit. *Exemple :* le client C400 est représenté par la page `CLI_C400.html` et le produit PA60 par la page `PRO_PA60.html`.

Chapitre 8

Construction d'une base de données

On introduit le problème de la construction d'un schéma de base de données décrivant un domaine d'application, c'est-à-dire la définition des tables, colonnes et contraintes d'intégrité qui représentent les besoins en information d'une communauté d'utilisateurs. On propose de travailler en deux étapes : d'abord l'élaboration du schéma conceptuel qui décrit, sans référence à la technologie informatique, les structures du domaine d'application et, ensuite, la traduction de ce schéma en structures de bases de données, y compris leur expression en SQL. Ce schéma conceptuel s'appuie sur un nouveau modèle, plus abstrait, le modèle Entité-Association.

Une démarche de conception de base de données doit permettre la définition des tables permanentes nécessaires aux besoins d'un ensemble d'utilisateurs. Cette base de données est relative à un domaine d'activité, à la gestion et au contrôle duquel elle doit contribuer ou même parfois qu'elle doit automatiser. Un tel domaine, dit **domaine d'application**, pourrait être une bibliothèque, un service hospitalier, le département de marketing d'une entreprise ou son service du personnel, la production d'énergie ou le relevé des infractions au code de la route.

La base de données doit contenir toutes les données nécessaires à la représentation du domaine d'application, et rien que ces données. Ces données sont groupées en tables de manière à définir une description aisément utilisable du domaine et, en particulier, de manière à permettre leur gestion par un ordinateur. Lorsque le

problème est simple, un utilisateur un tant soit peu habile pourra exprimer directement ses besoins en termes de tables, colonnes et contraintes.

Cependant, lorsque le domaine d'application présente une certaine complexité, il devient difficile, voire périlleux, de raisonner à son sujet en termes de tables et de colonnes. En effet, une base de données réelle contiendra plusieurs centaines de tables, dotées chacune de plusieurs dizaines de colonnes. Sans même envisager de telles bases de données, qui restent le domaine d'activité réservé des informaticiens spécialisés en systèmes d'information, il n'en reste pas moins qu'un niveau de raisonnement indépendant des outils de gestion de données tels que les SGBD SQL s'avère rapidement utile, voire indispensable¹. C'est pour cette raison qu'on fera appel à un autre mode de description du domaine d'application, le **modèle Entité-association**. Ce modèle, qui est le plus populaire à l'heure actuelle, permet de décrire plus naturellement les concepts du domaine, sans s'attacher à la manière dont ils seront représentés en tables et colonnes. Il offre en particulier une représentation graphique des concepts qu'il décrit, ce qui en fait un outil de raisonnement et de spécification particulièrement attrayant. La description du domaine s'exprimera sous la forme d'un **schéma conceptuel**.

Nous utiliserons une variante simplifiée du modèle Entité-association et nous adopterons des conventions graphiques plus simples que celles qui sont communément admises. On consultera par exemple [Bodart, 1994], [Batini, 1992], ou les différentes références sur MERISE [Nancy, 1996] pour une définition de modèles plus complets. Certaines extensions du modèle présenté seront cependant mentionnées dans la section 7.8 tandis que le modèle de classes d'UML² est étudié dans la section 7.9.

Le schéma conceptuel est un modèle mental, abstrait, et n'est donc pas directement implantable dans un ordinateur. Il faut par conséquent le *traduire en un schéma de base de données* sous la forme de tables, de colonnes et de contraintes d'intégrité. Ici encore, nous limiterons le processus de traduction à des règles simples et systématiques. On trouvera dans [Hainaut, 1986], [Batini, 1992], [Blaha, 1998] et [Akoka, 2001] une description plus complète de ce processus.

La démarche de conception de bases de données proposée consiste donc en deux phases (figure 8.1) :

- *l'analyse conceptuelle*, durant laquelle les besoins en information des utilisateurs sont traduits en un schéma conceptuel;
- *la production de la base de données*, par laquelle le schéma conceptuel est traduit en structures de tables exprimées en SQL.

1. Il existe d'autres arguments en faveur d'un niveau d'abstraction supérieur. Citons entre autres le fait que les mêmes structures de données peuvent être réalisées à l'aide d'autres outils que les SGBD SQL : SGBD orientés-objet ou relationnels-objet, SGBD plus anciens tels que CODASYL ou IMS, fichiers classiques, tableurs, XML, etc.

2. Variante du modèle Entité-association dérivée des approches orientées-objets.

Nous examinerons successivement les principes du modèle Entité-Association (chapitre 9), la phase de construction du schéma conceptuel (chapitre 10) et la phase de production des structures de tables SQL (chapitre 11).

Nous terminerons par la résolution de deux cas illustrant l'application de la méthode (chapitre 12).

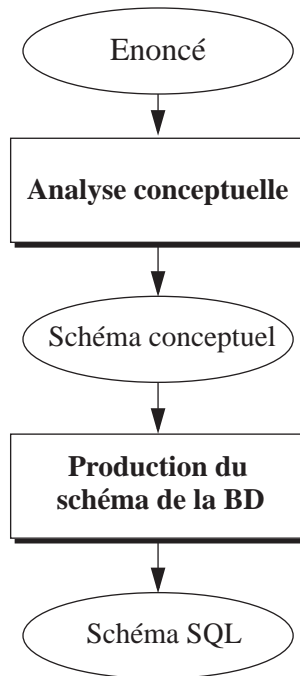


Figure 8.1 - Structure générale de la méthode de construction d'une base de données

Les activités d'analyse et de production peuvent rapidement s'avérer fastidieuses et sujettes à erreurs. C'est pourquoi les informaticiens se font assister par des outils spécialisés, les ateliers de génie logiciel, ou AGL³, de la même manière que les architectes, les graphistes et les ingénieurs disposent d'outils informatiques spécifiques à leur domaine.

Les schémas et les instructions SQL qui illustreront les chapitres de cette première partie ont été réalisés à l'aide de l'AGL DB-MAIN, spécialisé dans les activités d'ingénierie des applications de bases de données. Une version pédagogique est disponible sur le site de l'ouvrage. Elle est accompagnée d'un court tutoriel d'une durée d'une heure, qui permet de parcourir par la pratique l'essentiel des principes des chapitres 9, 10 et 11 de cet ouvrage.

3. Ou outils CASE (*Computer Aided Software Engineering*) en anglais.

Chapitre 9

Le modèle Entité-association

Les aspects importants de la réalité à représenter, ou *domaine d'application*, doivent être décrits d'une manière abstraite, indépendante de toute technologie (ici les structures de base de données). Le modèle Entité-association permet de décrire un domaine d'application sous la forme d'un schéma conceptuel : ensembles d'entités, dotées de propriétés et en association les unes avec les autres, et ce, sans référence aux notions techniques de tables, colonnes et autres index.

Dans ce chapitre, on présente les principales constructions du modèle Entité-association et leur propriétés. On illustre ces concepts par des exemples de schémas complets. Les dernières sections sont consacrées à quelques extensions propres aux modèles utilisés par les professionnels ainsi qu'à une discussion du modèle de classes d'UML.

9.1 TYPES D'ENTITÉS

Le domaine d'application est perçu comme étant constitué d'entités concrètes ou abstraites. Ainsi, dans le contexte de l'assurance automobile, on peut cerner un domaine d'application dans lequel on repère des clients, des véhicules, des contrats et des accidents. On considère que chacun d'eux est une **entité** du domaine et que chaque entité appartient à une classe ou **type d'entités**. On définit naturellement quatre types d'entités qu'on nommera CLIENT, VEHICULE, CONTRAT et ACCIDENT. On représentera ces types d'entités d'une manière graphique comme indiqué dans la figure 9.1.

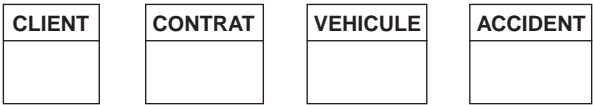


Figure 9.1 - Les quatre types d'entités du domaine d'application

Ces quelques exemples montrent qu'un type d'entités peut correspondre à des objets concrets inanimés (des véhicules), des objets concrets animés (des clients), des conventions abstraites (des contrats) ou des événements (des accidents).

Contrairement aux tables d'une base de données, les types d'entités n'ont pas à proprement parler de *contenu*. Chaque type d'entités *représente* une population, souvent variable d'un moment à l'autre. Il est cependant pratique, pour comprendre et analyser les caractéristiques d'un schéma, d'imaginer ces populations, classées par types. La figure 9.2 représente de manière imagée une population de chaque type, où une entité est représenté par une pastille¹.

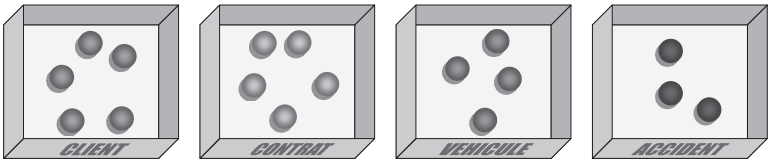


Figure 9.2 - Exemples de populations représentées symboliquement.

9.2 ATTRIBUTS

Chaque client est caractérisé par un numéro, un nom et une adresse. On modélisera ces faits en dotant le type d'entités CLIENT des **attributs** NumClient, Nom, Adresse (figure 9.3). De même, un accident étant caractérisé par un numéro, une date et un montant de sinistre, on donnera à ACCIDENT les attributs NumAcc, DateAcc et Montant.

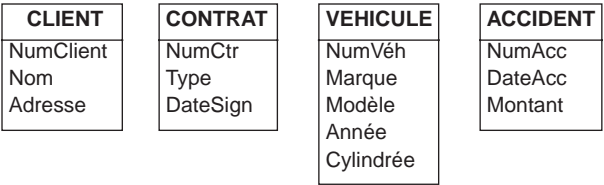


Figure 9.3 - Les quatre types d'entités et leurs attributs

On spécifiera le type de chaque attribut : numérique, caractère, date, etc. ainsi que sa longueur (figure 9.4). On indiquera aussi, si nécessaire, son unité de mesure.

1. Un auteur plus doué graphiquement aurait dessiné, fût-ce sommairement, quelques clients, des véhicules, des contrats, voire des accidents.

VEHICULE
NumVéh: char (16)
Marque: char (30)
Modèle: char (30)
Année: num (4)
Cylindrée: num (6)

Figure 9.4 - A chaque attribut correspond un type de valeurs

Il se peut que la valeur d'un attribut ne soit pas connue au moment où on décrit une entité. Si on admet que cet attribut puisse ne pas avoir de valeur pour certaines entités, on le déclarera *facultatif*. Sinon, cet attribut est *obligatoire*. Un attribut facultatif est indiqué par le symbole de cardinalité $[0-1]$, qui indique qu'à toute entité est associée de 0 à 1 valeur (figure 9.5). Notons que les attributs obligatoires sont caractérisés par une cardinalité $[1-1]$, mais qui est implicite, et donc n'est pas montrée dans les schémas.

ACCIDENT
NumAcc
DateAcc
Montant[0-1]

Figure 9.5 - Attributs obligatoires et facultatifs

9.3 TYPES D'ASSOCIATIONS

Un contrat est lié au client qui l'a signé; il existe donc une **association** entre ce contrat et ce client. On dira que toutes les associations de cette nature appartiennent au **type d'associations** *signe* entre les types d'entités CLIENT et CONTRAT. On définira également un type d'associations *appartient* entre CLIENT et VEHICULE, indiquant qu'un véhicule appartient à un client, ainsi qu'un type d'associations *couvre* entre CONTRAT et VEHICULE, indiquant qu'un contrat couvre les risques d'un véhicule. De même, en admettant qu'un accident implique un nombre quelconque de véhicules, on définira un type d'associations *implique* entre ACCIDENT et VEHICULE. Ces types d'associations sont représentés graphiquement dans la figure 9.6.

Lorsqu'un type d'entités intervient dans un type d'associations, on dit qu'il y joue un **rôle**. On utilisera d'ailleurs ce terme pour désigner une des extrémités d'un type d'associations. Comme nous le verrons plus loin, un même type d'entités peut jouer deux rôles dans un même type d'associations.

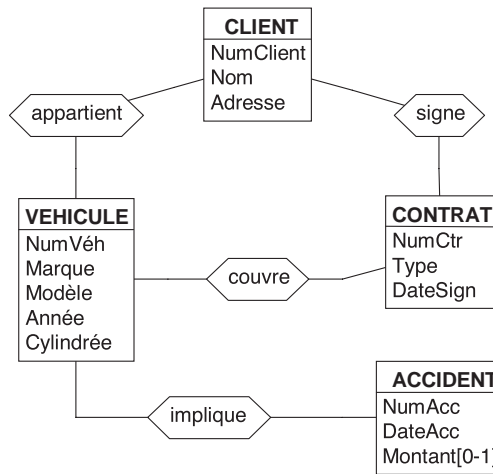


Figure 9.6 - Types d'entités, attributs et types d'associations

9.3.1 Propriétés d'un type d'associations

Nous allons chercher à préciser les propriétés d'un type d'associations R entre A et B en indiquant, pour chacun des types d'entités A et B, à combien d'associations R chaque entité peut et doit participer. Plus précisément, on mesurera cette propriété en indiquant, pour chaque type d'entités participant, le nombre minimum et le nombre maximum d'associations auxquelles toute entité participe. Nous présenterons ces propriétés en deux catégories : la *classe fonctionnelle* et le *caractère obligatoire/facultatif* des types d'associations, propriétés qui seront synthétisées sous la forme d'une propriété de *cardinalité*.

a) Classe fonctionnelle d'un type d'associations

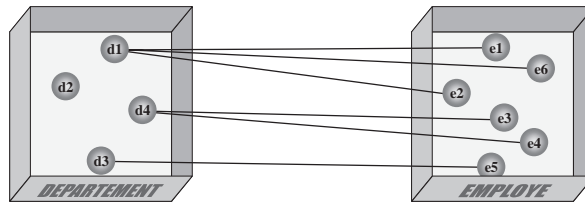
Cette propriété décrit le nombre maximum d'entités B pour chaque entité A, et inversement. On est ainsi amené à définir trois classes fonctionnelles² de types d'associations : *un-à-plusieurs*, *un-à-un* et *plusieurs-à-plusieurs*. Analysons ces trois classes de manière plus précise.

► Type d'associations un-à-plusieurs

Le schéma ci-dessous représente un ensemble de 4 départements d'une entreprise, symboliquement représentés par les pastilles étiquetées d1, d2, d3 et d4, ainsi qu'un ensemble de 6 employés, désignés par e1, e2, e3, e4, e5 et e6. Entre ces ensembles existe une relation qui exprime qu'*un département occupe des employés* et qu'*un employé est occupé par un département*. Un arc tracé entre les entités d4 et e3 indique

2. Le terme *fonctionnel* fait référence au concept de *fonction*, au sens mathématique du terme, que le type d'associations peut définir.

que le département d4 occupe l'employé e3. Il s'agit d'un type d'associations **un-à-plusieurs**.



On représente ce fait en indiquant, sur chacune des branches (ou rôles) du type d'associations, le nombre maximum d'associations dans lesquelles une entité peut apparaître, c'est-à-dire le nombre d'arcs issus de cette entité. On admet deux valeurs : un et plusieurs (qu'on notera respectivement 1 et N).

Le schéma de la figure 9.7 indique clairement qu'un département peut occuper plusieurs (N) employés, mais qu'un employé n'est occupé que par un seul (1) département.

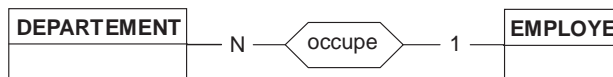
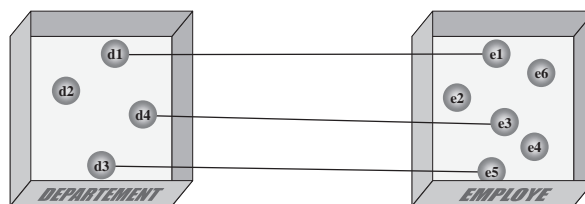


Figure 9.7 - Type d'associations un-à-plusieurs. Un employé est occupé par **un** département et un département peut occuper **plusieurs** employés

► Type d'associations un-à-un

Le schéma ci-dessous représente les mêmes ensembles de départements et d'employés. Entre ces ensembles existe une autre relation qui exprime qu'*un département a un directeur qui est un employé* et qu'*un employé peut être directeur d'un département*. Il s'agit d'un type d'associations **un-à-un**.



A chaque entité DEPARTEMENT correspond une seule entité EMPLOYE (dans le rôle de directeur), et à chaque entité EMPLOYE ne peut correspondre qu'une seule entité DEPARTEMENT.

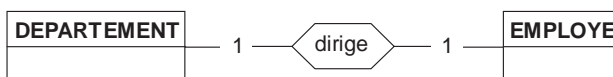
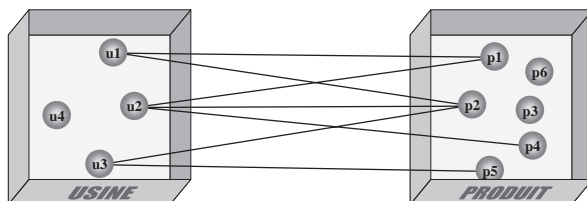


Figure 9.8 - Type d'associations un-à-un. Un employé peut diriger au plus **un** département et un département ne peut être dirigé que par **un** seul employé

On indiquera le symbole 1 sur chacune des branches du type d'associations (figure 9.8).

► Type d'associations plusieurs-à-plusieurs

Le schéma ci-dessous décrit un ensemble d'usines et un ensemble de produits. Entre ces ensembles existe une relation qui exprime qu'*une usine fabrique plusieurs produits* et qu'*un produit peut être fabriqué par plusieurs usines*. Il s'agit d'un type d'associations **plusieurs-à-plusieurs**.



A chaque entité USINE peuvent correspondre plusieurs entités PRODUIT, et à chaque entité PRODUIT peuvent correspondre plusieurs entités USINE. On indiquera donc le symbole N, représentant le nombre *plusieurs*³, sur chaque branche du type d'associations (figure 9.9).

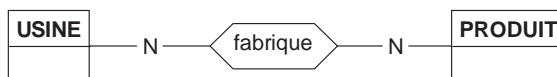


Figure 9.9 - Type d'associations plusieurs-à-plusieurs. Une usine fabrique **plusieurs** produits et un produit est fabriqué par **plusieurs** usines

La figure 9.10 complète le schéma de la figure 9.6 en y précisant la classe fonctionnelle de chaque type d'associations. Le lecteur cherchera à interpréter précisément ces indications en termes du domaine d'application.

b) Type d'associations obligatoire ou facultatif

On peut imposer qu'un type d'associations soit **obligatoire** pour un type d'entités qui y participe.

3. Il est essentiel de noter qu'on ne peut caractériser un type d'associations qu'après avoir répondu à **deux questions** : combien d'entités A pour chaque entité B, et combien d'entités B pour chaque entité A. Une erreur classique chez les « modélisateurs » débutants est de se limiter à une formulation du type suivant : à *un* département correspondent *plusieurs* employés, donc le lien est *un-à-plusieurs*. On jugera des conséquences de cette analyse incomplète dans le cas où on se contente de la formulation : à *une* usine correspondent *plusieurs* produits, donc ... ! Le lecteur mathématicien aura reconnu dans la classe *un-à-plusieurs* une relation fonctionnelle (ou du moins son inverse) et dans la classe *un-à-un* une bijection.

C'est ainsi que *occupe* sera déclaré obligatoire pour EMPLOYE (figure 9.11), imposant que toute entité EMPLOYE participe à une association *occupe*, ou encore qu'elle soit associée à une entité DEPARTEMENT.

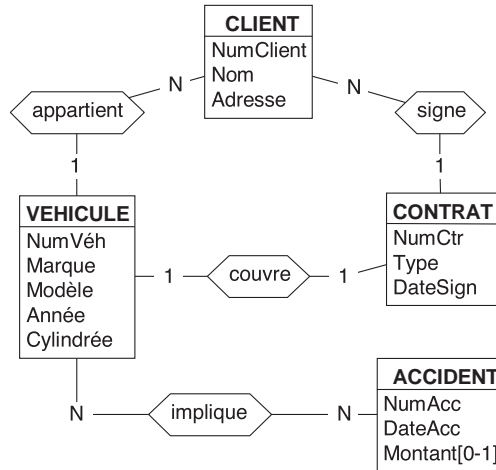


Figure 9.10 - Caractérisation des types d'associations (cardinalités partielles)

On exprime par là que tout employé est occupé par un département. De même qu'on a indiqué, par 1 ou N, le nombre maximum d'arcs associés, on indiquera par 0 ou 1, le nombre minimum d'arcs issus de toute entité. La figure 9.11 peut alors s'interpréter comme suit : toute entité DEPARTEMENT est associée, *via occupe*, à un nombre quelconque (de 0 à N) d'entités EMPLOYE, et toute entité EMPLOYE est associée, *via occupe*, à exactement une (de 1 à 1) entité DEPARTEMENT.

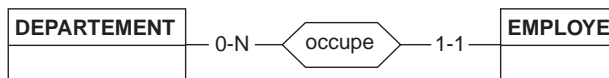


Figure 9.11 - Le type d'associations *occupe* est obligatoire (1) pour EMPLOYE et facultatif (0) pour DEPARTEMENT

c) Cardinalités d'un type d'associations

Chaque type d'entités apparaissant dans un type d'associations y est caractérisé par un couple de valeurs *min-max* appelé **cardinalité**. On dira que *occupe* est de cardinalités [0-N,1-1] de DEPARTEMENT vers . On n'admettra dans cet exposé que trois valeurs de cardinalité⁴ : 0-1, 1-1 et 0-N. La figure 9.12 reprend les cardinalités complètes.

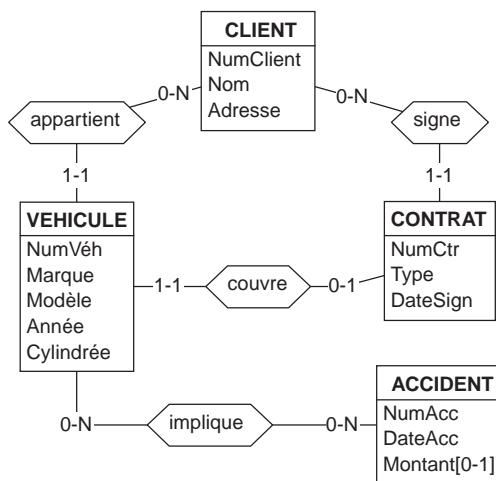


Figure 9.12 - Caractérisation des types d'associations : cardinalités complètes

Il est parfaitement possible d'établir un type d'associations entre un type d'entités et lui-même, définissant par là un type d'associations **cyclique**. La figure 9.13 illustre ce cas par la représentation d'une relation hiérarchique entre les personnes : une personne peut superviser un certain nombre d'autres personnes, qui sont ses subordonnés, et dont elle est le responsable.

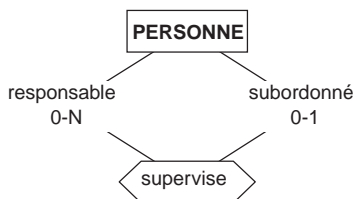


Figure 9.13 - Exemple de type d'associations cyclique

Si deux personnes ont une telle relation, on représentera celle-ci par une association *supervise* entre les entités correspondant à ces personnes. Il convient cependant de distinguer le rôle que joue chacune des personnes de cette association : l'une joue le rôle de *responsable* et l'autre celui de *subordonné*. D'ailleurs, une même personne peut jouer le rôle de subordonné dans une association et celui de responsable dans

4. Elles ne permettent pas de représenter avec précision toutes les situations qui peuvent se présenter en pratique. On pourrait admettre toute valeur $i-j$ telle que $i \leq j$, comme dans les modèles Entité-Association plus complets (voir section 9.9). Les valeurs retenues sont cependant suffisantes pour modéliser raisonnablement des domaines d'application simples, et ne poseront pas de problèmes lors de la traduction d'un schéma conceptuel en structures de base de données. Nous reviendrons sur ces limitations.

d'autres. Nous indiquerons chacun de ces rôles par des étiquettes attachées aux extrémités du type d'associations.

d) Représentation graphique des populations

La figure 9.14 représente de manière imagée un exemple de populations interconnectées entre lesquelles les arcs représentent les associations conformément au schéma de la figure 9.12.

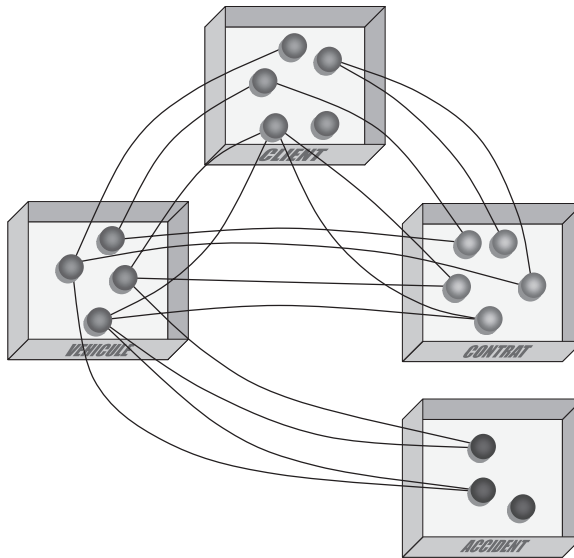


Figure 9.14 - Exemple de populations et des liens qui connectent les entités

9.4 LES IDENTIFIANTS

En général⁵, un type d'entités est doté d'un attribut qui identifie (c'est-à-dire distingue) les entités de ce type. Le type d'entités CLIENT, par exemple (figure 9.15) possède un attribut NumClient tel qu'à tout instant les entités CLIENT ont des valeurs de NumClient distinctes. En d'autres termes, étant donné une valeur quelconque de NumClient, on a la garantie qu'il n'y aura, à aucun moment, pas plus d'une entité CLIENT possédant cette valeur. On dira que NumClient est un **identifiant** de CLIENT. Il en est de même de NumVeh pour VEHICULE et de NumAcc pour ACCIDENT. Un identifiant peut également être formé de plusieurs attributs. En outre, il peut arriver qu'un type d'entités possède plus d'un identifiant. Dans ce cas, l'un d'eux peut être déclaré **primaire** tandis que tous les autres sont **secondaires** (figure 9.15).

5. Afin de simplifier l'exposé, on admettra qu'un type d'entités possède toujours un identifiant.

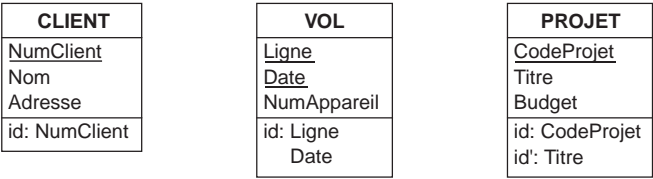


Figure 9.15 - Identifiants primaires (id) et secondaires (id') constitués d'attributs

On indiquera un identifiant primaire par une clause `id: NumClient` et un identifiant secondaire par la clause `id': Titre` dans le troisième compartiment du type d'entités. En outre, les attributs d'un identifiant primaire sont soulignés, du moins quand ce dernier n'est composé que d'attributs.

9.4.1 Les identifiants hybrides

Le cas du type d'entités **CONTRAT** (figure 9.16) est un peu plus complexe. On admet en effet que chaque contrat reçoit un numéro qui permet de le distinguer parmi ceux que le client a signés. Les contrats d'un client déterminé portent les numéros 1, 2, 3, ...

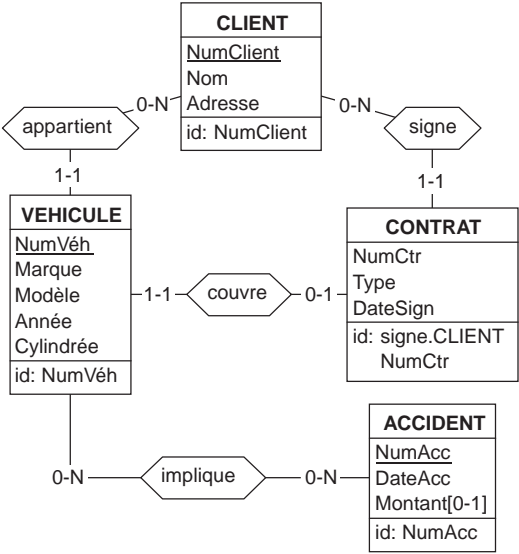


Figure 9.16 - Illustration de la notion d'identifiant (1)

Plusieurs contrats peuvent être signés par un même client et plusieurs contrats peuvent avoir la même valeur de NumCtr, mais il ne peut exister plus d'un contrat signé par le même client **et** portant le même numéro (NumCtr). On peut donc dire que **CONTRAT** est identifié par (CLIENT, NumCtr). Il s'agit d'un **identifiant hybride** constitué de types d'entités voisins et d'attributs. On indiquera un tel identifiant par la clause :

id: signe.CLIENT
NumCtr

La figure 9.16 spécifie les identifiants des types d'entités du schéma 9.12.

En toute généralité, un identifiant peut être constitué d'un nombre quelconque d'attributs et d'un nombre quelconque de types d'entités voisins.

Dans l'exemple de la figure 9.17 (le lecteur attentif aura reconnu le schéma conceptuel d'une base de données relationnelle bien connue), on exprime par l'identifiant de DETAIL que les détails d'une commande spécifient des produits distincts, ou encore que les détails spécifiant un produit déterminé appartiennent à des commandes différentes.

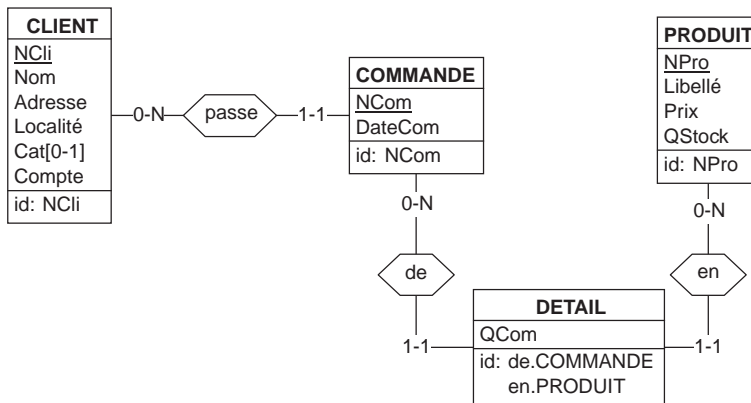


Figure 9.17 - Illustration de la notion d'identifiant (2)

La figure 9.18 propose d'autres exemples d'identifiants hybrides. On comparera le schéma de droite avec le schéma de la base de données de la figure 5.5.

Remarque

Le lecteur pourrait s'étonner de ce qu'on ne spécifie pas tout simplement l'identifiant de CONTRAT sous la forme des deux attributs (NumClient, NumCtr). Ce principe est possible pour le type d'entités CONTRAT, mais serait plus complexe, inadéquat ou même impossible dans d'autres cas.

Par exemple, le type d'entités SERVICE du schéma de la figure 9.22 aurait pour identifiant un ensemble de trois attributs, issus de trois types d'entités.

D'autre part, il faudrait, pour chaque type d'entités associé intervenant dans l'identifiant, soit choisir arbitrairement l'un de ses identifiants, soit déclarer autant d'identifiants qu'il y a de combinaisons possibles. Le nom d'attribut peut être ambigu : d'une part, il peut désigner plusieurs attributs différents, et d'autre part il peut être accessible par plusieurs chemins (voir les deux schémas de la figure 9.18 par exemple). Enfin, mais ceci concerne le modèle étendu présenté dans la section 9.9, un type d'entités n'a pas nécessairement d'identifiant.

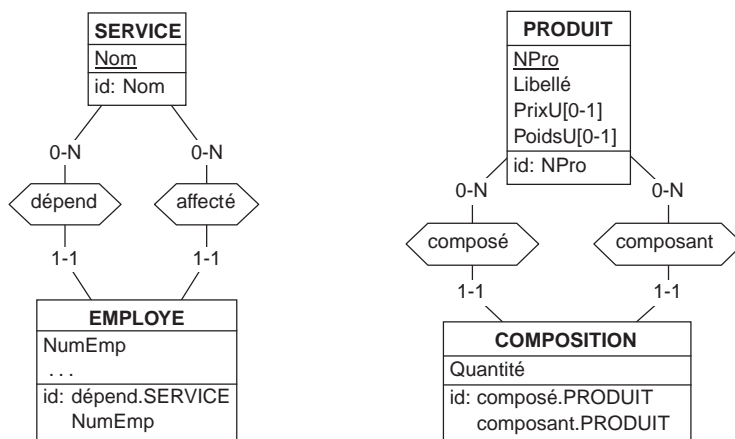


Figure 9.18 - Exemples additionnels d'identifiants hybrides

9.4.2 Composition des identifiants

En résumé, un identifiant d'un type d'entités E peut être d'une des compositions suivantes :

1. un attribut de E ,
2. plusieurs attributs de E ,
3. un ou plusieurs attributs de E + un ou plusieurs types d'entités E_i associés à E *via* le ou les types d'associations R_i ,
4. plusieurs types d'entités E_i associés *via* R_i à E .

Dans les compositions 3 et 4, la cardinalité de E dans chacun des R_i est $0-1$ ou $1-1$, tandis que la cardinalité des E_i est de $0-N$. Les types d'associations sont non cycliques. Si tous les composants d'un identifiant sont obligatoires, l'identifiant est déclaré obligatoire. Si tous ses composants sont facultatifs, l'identifiant est déclaré facultatif. On n'admet pas d'autres cas. Tous les composants d'un identifiant primaire sont obligatoires.

9.4.3 Identifiants minimaux et identifiants implicites

Lorsqu'un identifiant comprend plus d'un composant, il faut s'assurer que ceux-ci sont tous indispensables pour garantir l'unicité. On observe en effet que tout groupe de composants qui inclut ceux d'un identifiant est lui-même un identifiant. Ainsi, les attributs (NumAcc, Date) forment un identifiant d'ACCIDENT. On qualifiera de **minimal** un identifiant tel qu'aucun de ses composants ne puisse lui être retiré sans qu'il perde son statut d'identifiant. On veillera à ce qu'un schéma conceptuel ne contienne des identifiants minimaux.

Cette règle nous permet de justifier certaines des formes d'identifiants définies ci-dessus. Considérons par exemple le schéma du haut de la figure 9.19. On y indique que l'historique d'un client est identifié par le client et la date d'enregistrement.

Quelle que soit la signification de cette date, elle constitue un composant inutile de l'identifiant d'HISTORIQUE. En effet, le type d'association **a** étant *un-à-un*, a.CLIENT constitue à lui seul un **identifiant implicite** d'HISTORIQUE (à un client ne correspond qu'un seul historique). L'identifiant déclaré est donc *non minimal* et doit être corrigé par le retrait de DateEnreg. Ce qui reste étant un identifiant implicite, nous pouvons supprimer l'identifiant lui-même (figure 9.19, bas).

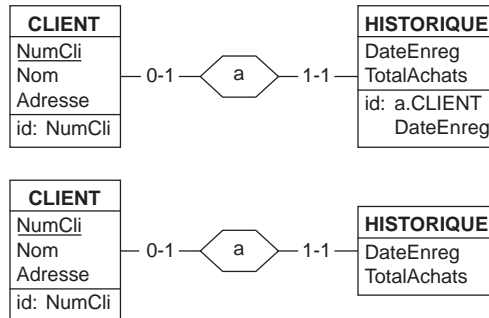


Figure 9.19 - En haut : l'identifiant d'HISTORIQUE est non minimal et doit être réduit. Ce qui reste (a.CLIENT) étant un identifiant implicite, doit aussi être supprimé (en bas).

9.4.4 Importance du concept d'identifiant

Les identifiants d'un type d'entités sont bien plus que de simples ornements de celui-ci. Ils contribuent à en définir la sémantique. Déterminer un identifiant de E revient à définir précisément ce qu'est une entité E, puisqu'il s'agit de trouver en quoi une entité se distingue des autres du même type. La figure 9.17 fournit un exemple représentatif. Une analyse rapide pourrait amener à confondre le concept de *détail* et celui de *produit*, et conduire au schéma de la figure 9.20.

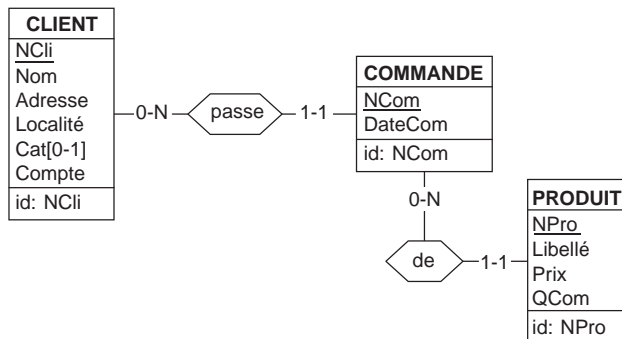


Figure 9.20 - Une mauvaise analyse conduit à confondre détail et produit. Cette erreur se traduit par un identifiant erroné

En effet, ce qui apparaît dans un bon de commande semble bien être un produit. Or si ce produit est identifié par son numéro (NPro), alors, selon ce schéma, un produit

ne pourrait apparaître que dans une seule commande, ce qui serait absurde. Le type d'entités **PRODUIT** ne représente donc pas les produits, mais un autre concept qui reste à découvrir.

Il est aussi important de noter qu'il existe plusieurs natures d'identifiants. Certains identifiants n'ont pas de signification particulière, et servent essentiellement à distinguer les entités d'un type. Tel est le cas de *NCli*, *NCom* et *NPro* dans le schéma 9.17. On dira de ces identifiants qu'ils servent de **désignateur**. Ils sont en général constitué d'un attribut technique (numéro, code, etc.). L'identifiant de **DETAIL** est d'une autre nature. Bien qu'il puisse aussi servir à désigner une entité **DETAIL**, il exprime surtout une **contrainte d'intégrité** qui correspond à une loi du domaine d'application : *une commande enregistrée ne peut citer un même produit plus d'une fois*.

La question se pose souvent du choix de l'identifiant primaire d'un type d'entités. Il conviendrait que sa valeur soit **stable** durant la durée de vie d'une entité. Or, s'il est formé d'attributs qui représentent des propriétés évolutives au cours du temps, ses valeurs seront naturellement instables. Une règle est souvent appliquée par les concepteurs de bases de données : l'identifiant primaire est formé d'un identifiant non significatif, et donc stable. Cette règle favorise les identifiants primaires de désignation.

Citons un exemple (réel) qui clarifie le propos. Une base de données comportait un répertoire des établissements scolaires. On avait défini pour ces derniers l'identifiant primaire comme suit : code postal de la commune de l'établissement + initiales du nom de l'établissement. On obtenait ainsi un code compact qui offrait en outre deux avantages : il était facile à reconstituer pour un établissement connu et il était possible d'en extraire une information utile sans consulter des données, à savoir la commune d'implantation. Ce choix qui semblait raisonnable au départ s'est avéré catastrophique à l'usage. En effet, d'une part, il n'est pas rare qu'un établissement change de nom et donc d'initiales, et d'autre part, des regroupements de communes entraînent une modification du code postal de certains établissements. Après quelques années, le logiciel de gestion d'écoles utilisant cette base de données a dû être transformé suite à cette décision maladroite et en apparence anodine.

9.5 AUTRES CONTRAINTES D'INTÉGRITÉ

Identifiants, cardinalités et attributs obligatoires ne constituent qu'un échantillon réduit des contraintes dont on voudrait disposer pour modéliser avec précision les propriétés des entités du domaine d'application. Pour exprimer les autres contraintes, celles qui n'ont reçu ni nom ni graphisme spécifiques, il faudra faire appel à un mode descriptif général, informel (en français) ou formel (rédigés sous forme de prédicats). Nous en donnerons quelques exemples sur lesquels nous reviendrons plus tard.

Précisons d'abord ce qu'on entend par *contrainte d'intégrité* : il s'agit d'une propriété que les objets décrits par un schéma (les entités, les associations et les valeurs d'attributs) doivent respecter de manière à représenter les situations et le

comportement du domaine d'application. Nous distinguerons les contraintes statiques des contraintes dynamiques.

9.5.1 Les contraintes d'intégrité statiques

Les contraintes d'intégrité statiques décrivent les états ou les situations valides. Les contraintes vues jusqu'à présent sont statiques. Quelques exemples supplémentaires :

- le prix unitaire d'un produit doit être supérieur à 0;
- il existe quatre valeurs de catégories de clients : B1, B2, C1, C2;
- un client ne peut être de catégorie C2 que si son compte est non négatif;
- toute commande doit avoir au moins un détail.

9.5.2 Les contraintes d'intégrité dynamiques

Une contrainte d'intégrité dynamique spécifie les changements d'états valides. Elle précise les modifications autorisées. Quelques exemples :

- un nouveau client peut entrer dans les catégories B1 ou C1; il peut ensuite passer de la catégorie B1 à B2 et inversement, ou de la catégorie C1 à C2 et inversement; il n'y a pas d'autres changements autorisés;
- on ne peut ajouter de détails qui référencent un produit en rupture de stock;
- on ne peut augmenter le prix d'un produit de plus de 5%;
- le changement d'état civil d'une personne obéit à des règles de transition précises : une personne peut passer du statut "célibataire" au statut "marié" mais pas aux statuts "divorcé" ou "veuf".

9.6 CONTENU INFORMATIONNEL D'UN SCHÉMA

La représentation graphique de populations représentatives des types d'entités d'un schéma conceptuel, telle que proposée à la figure 9.14, nous permet de vérifier qu'un schéma permet de répondre aux questions que se posent les utilisateurs.

Certaines questions sont à ce point simples qu'il suffit d'une simple inspection du schéma pour être convaincu que celui-ci contient l'information recherchée. Par exemple, retrouver les *accidents dans lesquels un véhicule est impliqué*, ou le *propriétaire d'un véhicule* sont des questions auxquelles le schéma peut de toute évidence répondre correctement. Pour vérifier l'aptitude d'un schéma à répondre à des demandes plus complexes, on fera appel à un graphe de population.

Considérons la recherche suivante : *trouver les signataires des contrats couvrant les véhicules impliqués dans un accident déterminé*.

Pour répondre à cette demande, on procède comme suit (figure 9.21) :

- on repère l'accident en question, qu'on marque de manière visible;
- en suivant les arcs du type d'associations implique on sélectionne les véhicules impliqués, qui sont également marqués;

- à partir des véhicules marqués, on sélectionne, via les associations couvre, les contrats qui couvrent les véhicules impliqués, et on les marque;
- à partir de ces contrats, et en suivant les associations signe, on obtient finalement les clients signataires.

Cette manière de conduire une recherche se généralise aisément à des conditions plus complexes portant par exemple sur les valeurs des attributs des entités (on ne retient que les véhicules de modèle X), ou sur le nombre d'arcs (les véhicules n'ayant eu qu'un seul accident). A titre d'exemple plus complexe, on imaginera un procédé similaire permettant de vérifier qu'on peut retrouver *les véhicules dont le propriétaire n'est pas le client qui a signé le contrat qui les couvre*.

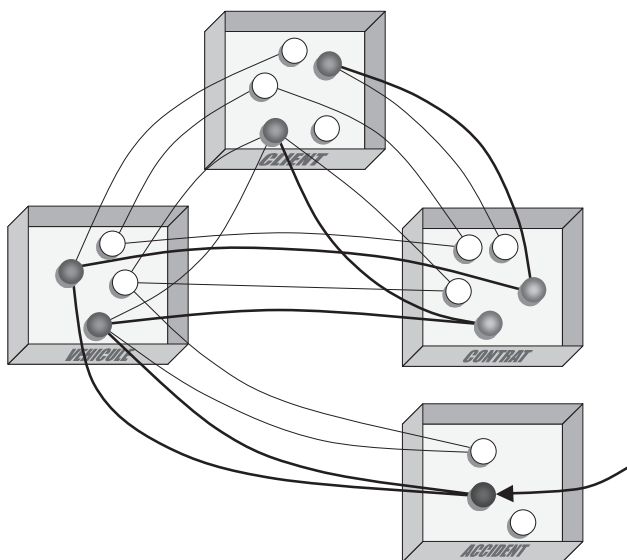


Figure 9.21 - Interrogation d'un ensemble de populations: *quels sont les clients qui ont signé les contrats couvrant les véhicules impliqués dans un accident déterminé ?*

9.7 EXEMPLES

Nous proposerons quelques exemples supplémentaires illustrant la notion de schéma conceptuel exprimé dans le modèle Entité-Association⁶. Nous donnerons pour chacun un énoncé en français, suivi d'un exemple de schéma conceptuel et d'une discussion. Il n'est pas exclu qu'à un même domaine d'application correspondent plusieurs schémas conceptuels.

6. Le lecteur intéressé par des schémas conceptuels d'entreprise consultera par exemple [Silverston, 1997].

9.7.1 Une structure administrative

On considère un sous-ensemble d'une structure administrative. D'une direction (caractérisée par un nom identifiant et le nom de son président-directeur général) dépendent plusieurs départements (dotés chacun d'un nom identifiant dans sa direction et de sa localisation). Un département est découpé en services, dotés chacun d'un nom (identifiant dans son département) et d'un responsable. Un service a la charge d'un certain nombre de dossiers identifiés par un numéro et dotés d'un titre et d'une date d'enregistrement. Dans chaque service travaillent des employés identifiés par un numéro et caractérisés par leur nom et leur adresse. La figure 9.22 représente un exemple de schéma conceptuel décrivant cette situation.

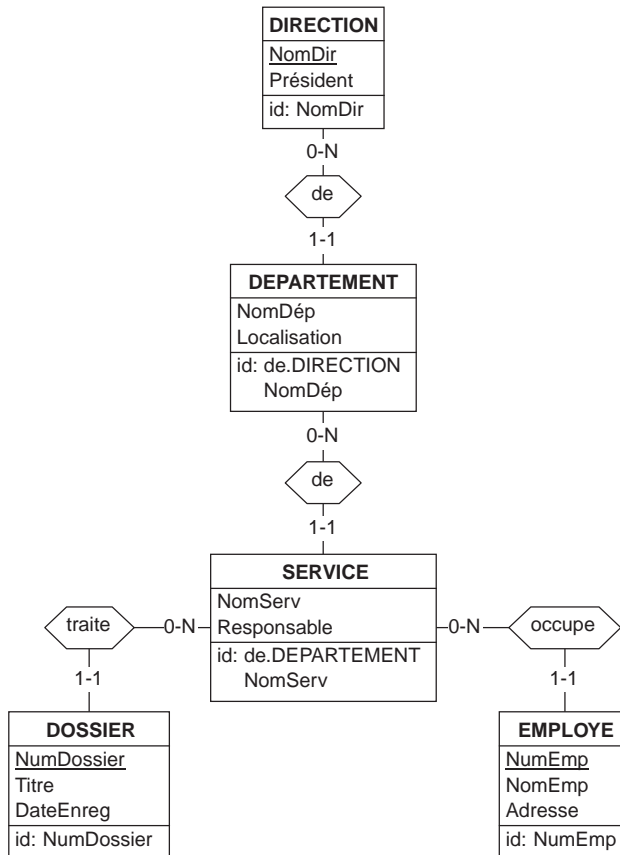


Figure 9.22 - Structure administrative élémentaire

Cette structure permet de répondre à des questions immédiates telles que les suivantes :

- quel est le service d'un employé déterminé ?
- dans quelle direction se trouve tel département ?
- quelle est la date d'enregistrement de tel dossier ?

Elle peut également fournir une réponse à des questions plus complexes, pour lesquelles on appliquera la procédure suggérée en 9.6 :

- dans quelle direction travaille un employé déterminé ?
- quels sont les dossiers traités dans tel département ?
- quels sont les dossiers que *pourraient* traiter les employés habitant à Lille ?
- quels sont les collègues de tel employé (dans le service, dans le département, dans la direction) ?
- quels sont les employés qui travaillent dans le même département qu'un employé déterminé, mais pas dans le même service ?
- dans quels départements y a-t-il des services qui ne traitent aucun dossier ?

On montrera qu'en revanche, ce schéma ne permet pas de déterminer sur quels dossiers travaille *effectivement* un employé déterminé, ni quels employés traitent *réellement* un dossier déterminé.

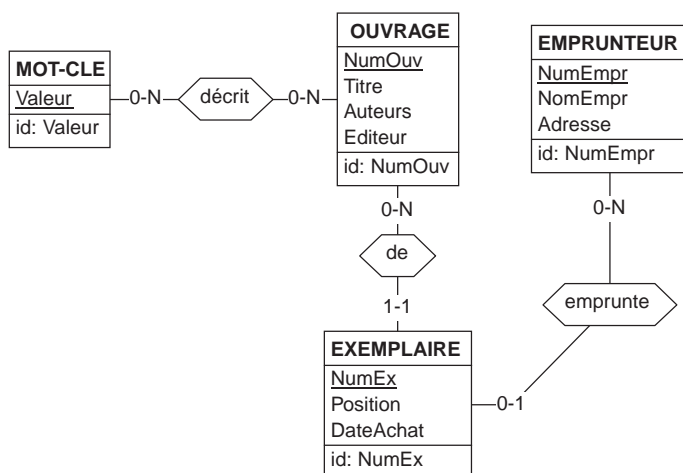


Figure 9.23 - Gestion de bibliothèque élémentaire

9.7.2 Gestion d'une bibliothèque

Considérons dans ce deuxième exemple une petite bibliothèque contenant des ouvrages pouvant être empruntés. Un ouvrage est caractérisé par un numéro identifiant, un titre, une liste d'auteurs et un éditeur. En outre, on décrit un ouvrage par un certain nombre de mots clés qui indiquent les sujets qui y sont traités. La bibliothèque dispose en général d'un ou plusieurs exemplaires de chaque ouvrage. L'exemplaire, qui est en quelque sorte la *matérialisation* d'un ouvrage, est identifié par un numéro et caractérisé par sa position dans les rayonnages et sa date d'achat. Un exemplaire peut être emprunté par un emprunteur. Ces derniers sont identifiés par un numéro d'emprunteur et sont caractérisés par un nom et une adresse. Un schéma conceptuel possible serait celui de la figure 9.23.

Outre les informations immédiates, cette structure permet d'obtenir des informations dérivées telles que les suivantes :

- quels auteurs lit tel emprunteur ?
- quels sont les emprunteurs intéressés par tel sujet (= valeur de mot clé) ?
- quels sont les ouvrages empruntés par plus de deux emprunteurs ?
- quels sont les ouvrages pour lesquels il reste des exemplaires disponibles (non empruntés) ?
- quels sont les emprunteurs *monomaniaques* (ayant emprunté au moins cinq exemplaires, caractérisés par un même mot clé) ?

9.7.3 Voyages en train

Cet exemple concerne des voyages en train. Un voyage est effectué par un train (dont on connaît le numéro identifiant et le dépôt d'origine), à une date donnée et à une heure de départ donnée, suivant une ligne déterminée (identifiée par un code de ligne et caractérisée par une date de mise en activité). Le conducteur effectuant le voyage est un agent.

Ce dernier possède un identifiant d'agent, un nom et une adresse. Une ligne est formée de sections consécutives de longueur déterminée. Une section d'une ligne part d'une station pour arriver à une autre. Une station est identifiée par un nom, est localisée dans une commune et est dirigée par un agent.

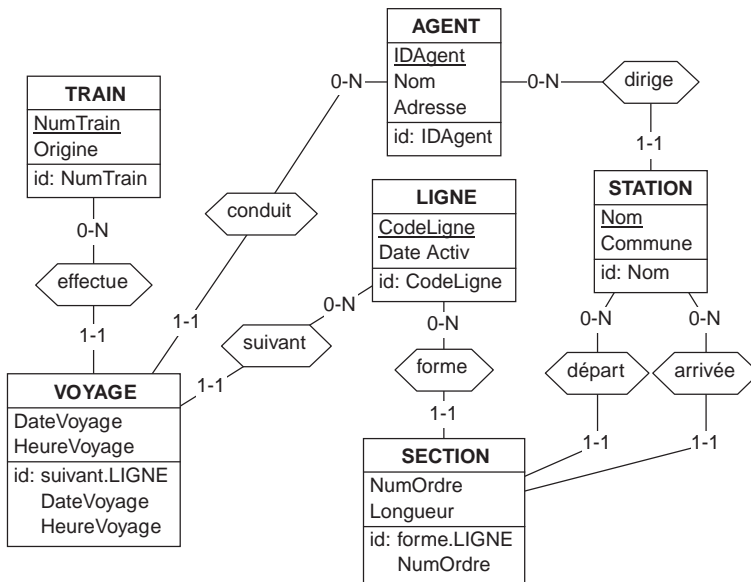


Figure 9.24 - Représentation de voyages en train

On peut proposer un schéma conceptuel des informations relatives à cette situation tel que celui de la figure 9.24. Ce schéma est un support adéquat pour répondre aux requêtes suivantes :

- Établir l'itinéraire d'un voyage déterminé.
- Quelle est la longueur d'un voyage ?

- Quels sont tous les voyages qui partent de la même station ?
- Quels sont les conducteurs qui ont fait plus de 100 000 km en 1999 ?
- Quelle distance totale a effectuée tel train ?
- Y a-t-il des agents qui sont à la fois conducteurs et directeurs de station ?

9.8 QUELQUES RÈGLES DE PRÉSENTATION

Un schéma peut devenir rapidement illisible dès qu'il inclut quelques dizaines de types d'entités⁷. On peut améliorer la lisibilité d'un schéma en cherchant à respecter quelques règles simples de positionnement.

- Si deux types d'entités sont reliés par un type d'associations *un-à-plusieurs*, on placera celui dont la cardinalité est 0-1 ou 1-1 plus bas que l'autre. On respecte ainsi la structure hiérarchique induite par ce type d'associations (cfr. schéma 9.22).
- Si deux types d'entités sont reliés par un type d'associations *plusieurs-à-plusieurs*, on les placera sensiblement au même niveau pour suggérer l'absence de hiérarchie (cfr schéma 9.23).
- Si deux types d'entités sont reliés par un type d'associations *un-à-un*, on les placera également au même niveau. En revanche, si l'un joue un rôle de cardinalité 1-1, on le disposera plus bas que l'autre, indiquant ainsi la relation de dépendance.
- Dans la liste des attributs d'un type d'entités, on placera les composants de l'identifiant primaire en premier.
- On adoptera des conventions cohérentes dans le choix des noms des types d'entités, des types d'associations et des attributs. Dans cet ouvrage, on a essayé d'assigner aux constructions des schémas Entité-association des noms respectant les règles suivantes :
 - *types d'entités* : substantifs en capitales,
 - *attributs* : substantifs en minuscules, première lettre en capitale,
 - *types d'associations* : verbes, prépositions ou conjonctions en minuscules.

En cas de règles contradictoires ou conduisant à une topologie inadéquate (schéma trop étendu par exemple), on fera appel au bon sens. On observera d'ailleurs que certains schémas de cet ouvrage ne respectent pas toutes ces règles.

9.9 EXTENSIONS DU MODÈLE ENTITÉ-ASSOCIATION

Ainsi que nous l'avons signalé à plusieurs reprises, le modèle qui vient d'être présenté est un sous-ensemble des formalismes utilisés par les informaticiens en conception de systèmes d'information, et plus particulièrement en conception de

7. Un schéma réaliste contiendra plusieurs centaines à plusieurs milliers de types d'entités.

bases de données. Ce modèle simplifié est tout à fait suffisant pour résoudre la plupart des problèmes qui peuvent se poser en pratique à l'utilisateur non professionnel de l'informatique. Il est en tout cas assez proche des modèles utilisés dans les méthodes Merise de la première génération. Nous voudrions dans cette section évoquer quelques extensions qui enrichissent les modèles Entité-association utilisés dans le développement de systèmes d'information tels que ceux qui sont proposés par [Bodart, 1994], [Nanci, 1996], [Elmasri, 2000], [Batini, 1992] ou [Ceri, 1997]. La discussion de ces extensions est basée sur les schémas des figures 9.25, 9.26 et 9.27.

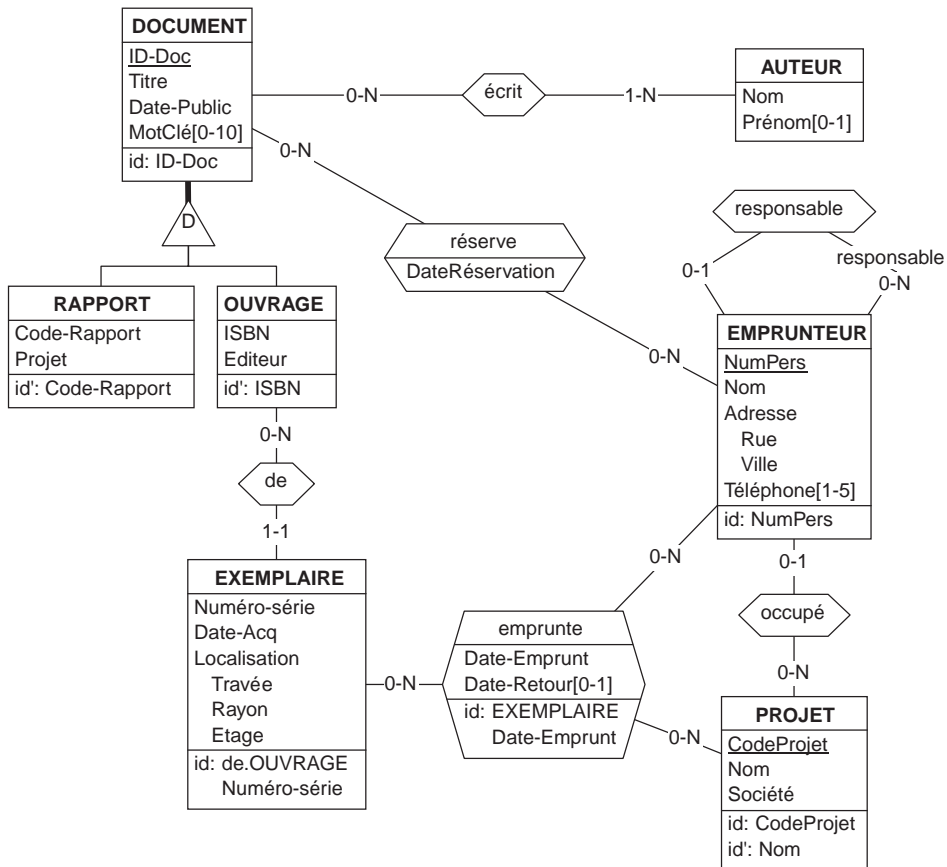


Figure 9.25 - Un exemple de schéma Entité-association étendu

- **Les relations surtype/sous-type ou relation IS-A.** Une entité peut appartenir à plus d'un type. Par exemple, les rapports forment une classe spéciale de documents, de même que les ouvrages. On dira que *tout rapport est un document* (ou *is a* en anglais, d'où le terme de relation IS-A). Dans le schéma conceptuel, on déclarera que les types d'entités **RAPPORT** et **OUVRAGE** sont deux *sous-types* de **DOCUMENT**, ou que **DOCUMENT** est un *surtype* des deux premiers. En conséquence de quoi, les entités **RAPPORT** héritent des caracté-

- ristiques du type DOCUMENT. Par exemple, les attributs, les types d'associations et les identifiants de DOCUMENT sont automatiquement ceux de RAPPORT et OUVRAGE sans qu'il soit nécessaire de l'indiquer explicitement. On parlera par exemple du titre et de la date de publication d'un ouvrage ou d'un rapport. Le symbole D de la relation surtype/sous-type indique que ces sous-types sont disjoints (un ouvrage ne peut être un rapport)⁸.
- **Identifiants.** Un type d'entités peut n'avoir aucun identifiant. Tel est le cas d'AUTEUR.
 - **Types d'associations N-aires.** Un type d'associations peut relier plus de deux types d'entités. On peut ainsi décrire explicitement le fait que les *emprunteurs* empruntent des *exemplaires* pour le compte de *projets* par un type d'associations *emprunte* entre les trois types d'entités correspondants. On appelle N-aires, où N est le nombre de rôles, de tels types d'associations
 - **Généralisation des cardinalités.** Les contraintes de cardinalité ne sont pas limitées aux trois combinaisons proposées. Tout couple min-max est autorisé pour autant que $0 \leq \min, 1 \leq \max \leq N$ et $\min \leq \max$.
 - **Attributs de types d'associations.** Un type d'associations peut recevoir des attributs. Par exemple, le type d'associations *réserve*, qui indique qu'un document est réservé par un emprunteur, est caractérisé par l'attribut *Date Réservation*, qui spécifie la date à laquelle cette réservation a été enregistrée.
 - **Identifiants d'un type d'associations.** Chaque association *emprunte* représente le fait qu'un emprunteur a emprunté un exemplaire d'ouvrage pour le compte d'un projet, à une date d'emprunt déterminée, et, si l'emprunt est clôturé, jusqu'à une date de retour déterminée. La gestion de la bibliothèque est telle qu'un exemplaire restitué n'est remis en circulation que le lendemain au plus tôt. Il n'y a donc pas deux emprunts du même exemplaire le même jour, d'où l'identifiant d'emprunte.

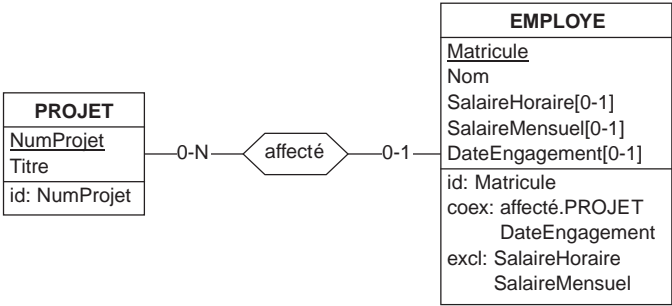


Figure 9.26 - Contraintes d'intégrité spécifiant que (1) si un employé est affecté à un projet, alors il possède aussi une date d'engagement, et inversement (ces deux informations sont *coexistantes*), (2) un employé a un salaire horaire ou un salaire mensuel, mais pas les deux (*exclusive*)

8. Le symbole T (total) aurait indiqué que toute entité DOCUMENT doit aussi appartenir à au moins un de ses sous-types, tandis que le symbole P (partition) aurait indiqué que toute entité DOCUMENT doit appartenir à un et un seul de ses sous-types.

- **Attributs composés.** Une valeur d'un attribut composé est constituée d'un assemblage de valeurs plus élémentaires. L'attribut Adresse d'EMPRUNTEUR est ainsi constitué des attributs plus élémentaires Rue et Ville.
- **Attributs multivalués.** En principe, chaque entité d'un type possède **une** valeur de chacun de ses attributs. Si un attribut est multivalué, alors une entité peut posséder plusieurs valeurs de cet attribut. Tel serait le cas de l'attribut Mot Clé de DOCUMENT et Téléphone d'EMPRUNTEUR. Le nombre de valeurs par entité est caractérisé par une contrainte de cardinalité min-max.
- **Contraintes d'intégrité.** Les modèles professionnels sont dotés d'un jeu de contraintes d'intégrité plus riche que celui que nous avons défini dans ce chapitre. La figure 9.26 illustre deux de ces contraintes.

Les modèles Entité-association ont aussi intégré certains concepts des approches orientées-objets. C'est ainsi qu'il est permis de spécifier le comportement dynamique d'un type d'entités sous la forme d'un ensemble de **méthodes**.

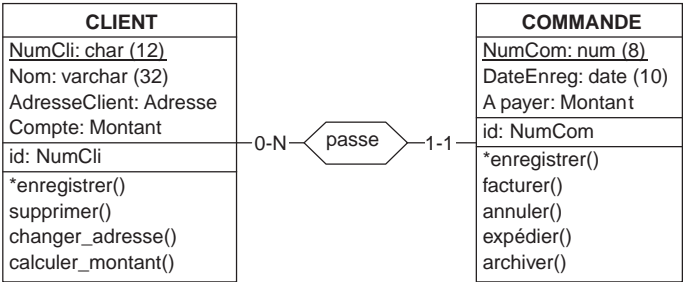


Figure 9.27 - Un type d'entités peut être doté de méthodes qui définissent son comportement. Les attributs *Compte* et *A payer* sont définis sur le domaine *Montant*, défini par l'utilisateur

Une méthode est un service qui peut être demandé à toute entité d'un certain type. Elle se présente sous la forme d'une procédure dont l'exécution est déclenchée par un message envoyé à l'entité. Le schéma de la figure 9.27 spécifie que les actions dont les commandes peuvent faire l'objet sont : son enregistrement⁹, la production d'une facture, son annulation, l'expédition des produits commandés (non illustrés dans le schéma), son archivage. Dans certains modèles récents, il est possible de spécifier des domaines de valeurs (exemple Montant) sur lesquels des attributs peuvent être définis.

9. Le préfixe * indique qu'il s'agit d'une méthode de classe et non d'instance : on demande à la classe COMMANDE d'enregistrer une nouvelle entité, mais on demande à une entité de s'archiver.

9.10 ... ET UML ?

UML (*Universal Modelling Language*) est un ensemble de notations permettant de représenter divers aspects d'une application informatique. Intégrant les concepts des trois méthodes OMT [Rumbaugh, 1991], OOSE/Objectory [Jacobson, 1992] et OOAD [Booch, 1994], UML offre cinq classes de modèles permettant de représenter les aspects du développement d'une application orientée-objet qui vont des scénarios d'utilisation jusqu'aux processus d'implémentation et de déploiement.

Nous discuterons plus particulièrement les principaux composants du *modèle des Diagrammes des structures statiques*, ou plus simplement *modèle de classes*, qui décrit les classes d'objets qui sont à la base de l'application. Souvent proposé comme substitut du modèle Entité-association, le modèle de classes d'UML mérite qu'on lui consacre quelques pages, bien qu'au départ il ne dispose pas des caractéristiques nécessaires à la modélisation conceptuelle des données¹⁰.

9.10.1 Le modèle de classes d'UML

Le concept central de ce modèle est celui de *classe d'objets*. Une classe comporte des *attributs*, peut faire l'objet d'*opérations* et former des *associations* (figure 9.28).

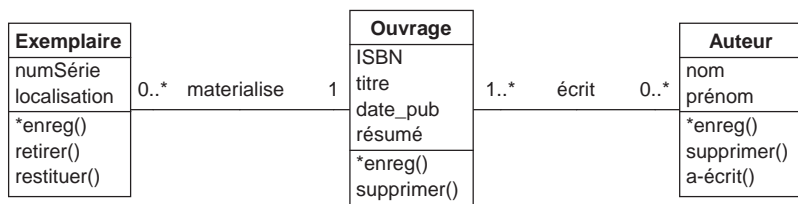


Figure 9.28 - Un diagramme de classes UML

a) Les classes d'objets

Une classe correspond approximativement à un type d'entités. A une classe peuvent être associées des sous-classes tout comme dans le modèle Entité-association.

b) Les associations

Une association UML correspond à un type d'associations du modèle Entité-association. UML distingue les associations binaires et les associations N-aires.

Une *association binaire* possède deux extrémités (rôles), éventuellement nommées, caractérisées par leur multiplicité (correspondant aux cardinalités). Les conventions de dénotation des multiplicités sont légèrement différentes de celles des cardinalités : $\min\text{-}\max$ se note $\min.. \max$, N se note $*$, $1-1$ peut se noter $1..1$ ou 1 et $0-N$ peut s'écrire $0..*$ ou simplement $*$. Il faut être attentif au fait que la multiplicité UML est une propriété du rôle opposé, et se note donc sur l'extrémité

10. Une première version de cette section incluait une critique plus approfondie de la notation UML comme formalisme conceptuel. Le lecteur intéressé la retrouvera sur le site de l'ouvrage.

opposée à celle à laquelle elle s'applique. Dans le cas de la figure 9.28, la multiplicité 1 (soit 1..1) de l'extrémité Ouvrage de matérialise signifie qu'à partir d'un Exemple, on voit un et un seul Ouvrage. On retient donc que *les multiplicités UML sont inversées par rapport aux cardinalités Entité-association*.

Une *association N-aire* possède plus de deux extrémités. Elle se représente par un losange, auquel un nom peut être attaché. Une extrémité d'une association N-aire peut porter une multiplicité, mais l'interprétation de ce concept et son intérêt sont loin d'être évidents¹¹. Il est intéressant de noter de plusieurs auteurs ignorent les associations n-aires [D'Souza, 1998]. Certains de ceux qui les prennent en compte suggèrent de s'en tenir au degré 3 et d'ignorer les multiplicités, ou de les remplacer par des identifiants d'association [Blaha, 1998].

c) Les attributs

Les documents UML ne fournissent que peu d'information sur la nature et la structure des attributs. Tout au plus suggèrent-ils l'existence d'attributs multivalués (caractérisés par une multiplicité, similaire aux cardinalités d'attributs) ou basés sur des types définis par ailleurs.

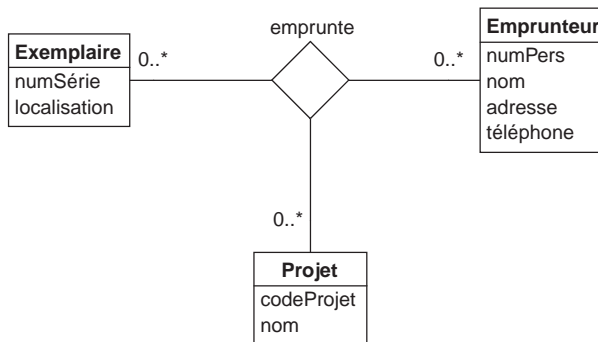


Figure 9.29 - Une association n-aire UML

d) Les opérations

Une opération est un service qu'un objet ou une classe peut rendre suite à une demande externe (par exemple, l'opération supprimer de la classe Ouvrage dans le diagramme 9.28).

11. La multiplicité d'une extrémité occupée par la classe C indique combien d'objets C sont vus à partir de toute combinaison de N-1 objets correspondant aux N-1 autres extrémités. Les propositions UML avouent lucidement que *Multiplicity for N-ary associations may be specified, but it is less obvious than binary multiplicity. The multiplicity on a role represents the potential number of instances tuples in the association when the other N-1 values are fixed.* [UML, 1999]. Si cette interprétation peut avoir un intérêt, même mineur, pour la multiplicité maximale, elle s'avère particulièrement ésotérique pour ce qui concerne la multiplicité minimale.

e) Les associations qualifiées

Il s'agit d'une des curiosités les plus pittoresques d'UML. Le *qualifieur* d'une association R entre les classes A et B est un attribut T (ou un groupe d'attributs) attaché à l'extrémité A et dont les valeurs définissent, pour chaque objet A, une *partition* des objets B qui lui sont attachés via R. T est considéré comme un attribut de R. Un qualifieur peut induire une modification de la multiplicité de l'extrémité B de R. En effet, en présence d'un qualifieur, cette multiplicité représente le nombre d'objets B dans chaque sous-ensemble défini par la partition. Le cas le plus fréquent est celui dans lequel cette multiplicité se réduit à 1 ou 0..1.

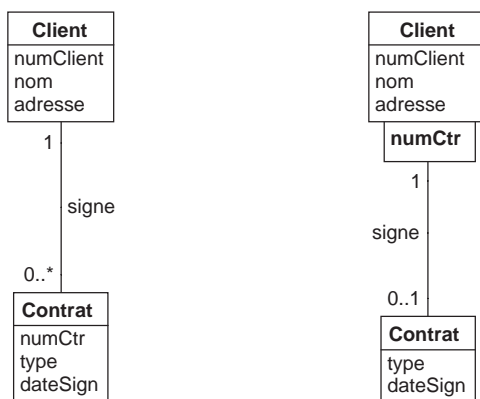


Figure 9.30 - Association non qualifiée (gauche) et qualifiée (droite). L'attribut *numCtr* qualifiant l'association *signe* permet d'indiquer qu'un client ne peut avoir plus d'un contrat de même valeur de *numCtr*

L'exemple de la figure 9.30 illustre ce dernier cas. Le schéma de gauche indique qu'un client peut signer un nombre quelconque de contrats. Celui de droite ajoute qu'à un même client et une valeur déterminée de *numCtr* ne correspond qu'un seul contrat, et donc que les contrats d'un client ont des valeurs de *numCtr* distinctes. Ce concept est à la fois mal défini et incomplet. Dans certains cas particuliers assez fréquents, il conduit à des schémas confus ou erronés.

f) Les classes associations

Une association ne peut posséder d'attributs. En revanche, on peut définir une *classe association*, c'est-à-dire une association qui est également une classe (cfr emprunte et réserve dans le schéma 9.32).

g) Les associations d'agrégation et de composition

Une association d'agrégation associe à chaque objet composite l'ensemble de ses composants. On peut ainsi indiquer qu'une équipe *inclut* des personnes ou que les actes d'un colloque *sont composés d'* articles (figure 9.31).

Si un objet composant peut exister indépendamment de l'objet composite, on parlera de simple *agrégation*. Si en revanche un objet n'existe que comme composant d'un objet composite, on parlera d'une association de *composition*.

h) Les contraintes

A une exception près, UML ne propose pas de contraintes d'intégrité explicites, laissant à l'analyste le soin d'annoter son schéma avec des commentaires en texte libre spécifiant ces contraintes. Pour des raisons non élucidées, UML définit une (et une seule) contrainte : l'*exclusion* d'associations (improprement notée $\circ x$). On notera qu'UML ignore la plus importante des contraintes d'intégrité de ce domaine : l'**identifiant**¹².

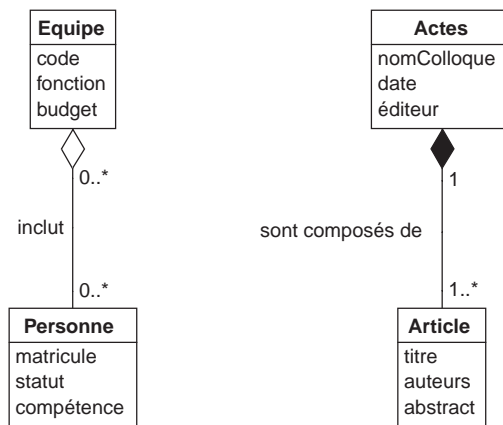


Figure 9.31 - Associations d'agrégation (*inclut*) et de composition (*sont composés de*)

Le modèle de classe d'UML inclut d'autres concepts propres aux langages orientés-objet mais qui sont non pertinents dans le domaine des bases de données. Etant donné la portée limitée de cette section, nous ne les décrirons pas.

9.10.2 Un exemple de schéma de classes en UML

Dans la figure 9.32, nous exprimerons en UML l'essentiel des constructions du schéma de la figure 9.25. En raison des limitations d'UML, certains éléments importants sont manquants, tels que les identifiants, à l'exception de celui d'Ouvrage.

Il existe à l'heure actuelle chez tous les éditeurs un grand nombre de références en langue française traitant d'UML. Au lecteur de faire son choix.

12. Du moins si on excepte les associations qualifiées dont la multiplicité dégénère en 1 ou 0..1, et qui définissent un cas particulier d'identifiant.

9.10.3 Le modèle de classes d'UML revisité

Si on accepte quelques accommodements par rapport aux recommandations de l'OMG, il est possible de définir une variante du modèle de classes d'UML parfaitement adaptées à la représentation de schémas conceptuels. Un tel modèle serait constitué notamment des concepts suivants.

- Les classes d'objets.
- Les sous-classes.
- Les domaines définis par l'utilisateur.
- Les attributs de classe; obligatoires ou facultatifs, simples ou composés, mono-valués ou multivalués.

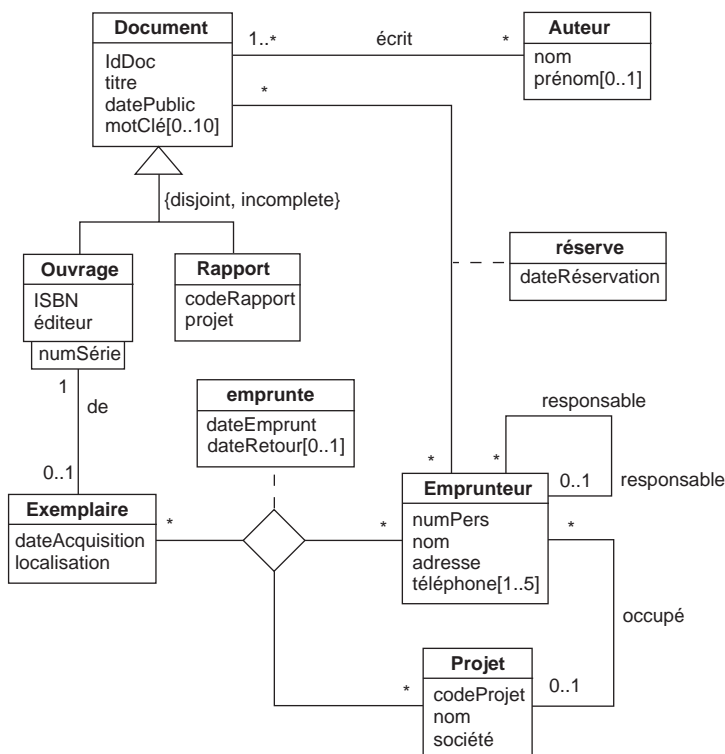


Figure 9.32 - Traduction approximative du schéma 9.25 en UML standard. Les identifiants n'ont pas été exprimés, à l'exception de celui d'Exempleire qui correspond à une association qualifiée. Les attributs composés n'apparaissent pas en tant que tels

- Les identifiants de classe.
- Les (autres) contraintes d'intégrité de classe.
- Les opérations associées aux classes.
- Les associations binaires.
- Les associations de composition.

Cette variante ne reprendrait pas les constructions suivantes, mal définies et/ou inutiles car exprimables plus simplement.

- Les associations N-aires.
- Les associations qualifiées.
- Les associations d'agrégation.
- Les classes-associations.

Le schéma de la figure 9.32 pourrait alors s'exprimer en UML sans perte d'information sous la forme du schéma 9.33.

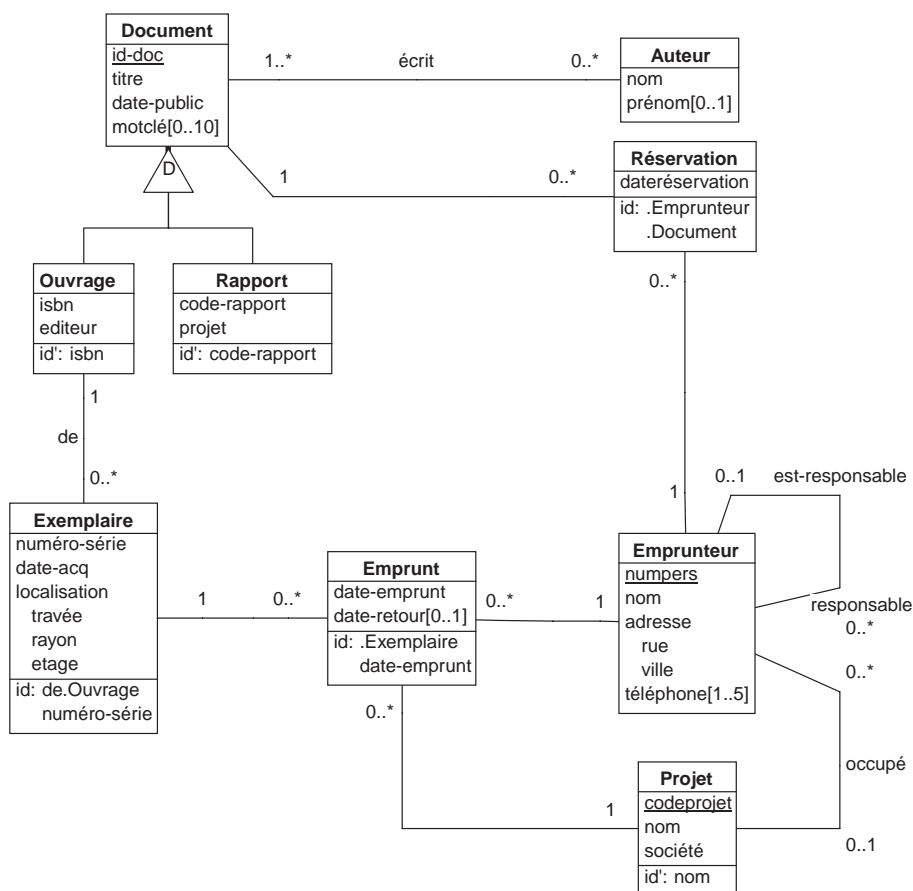


Figure 9.33 - Une variante du modèle de classe d'UML permet de représenter la plupart des structures d'information au niveau conceptuel

9.11 EXERCICES

- 9.1 On considère le schéma de la figure 9.16. Admettons à présent que, toutes choses restant égales par ailleurs, (1) un véhicule puisse être couvert par un nombre quelconque de contrats, (2) on désire connaître, chaque fois qu'un véhicule est impliqué dans un accident, le pourcentage de responsabilité qui lui a été attribué. Modifier le schéma en conséquence.
- 9.2 Dans l'interprétation du schéma 9.22, on précise qu'un service est réputé traiter un dossier dès qu'au moins un de ses employés est en charge de ce dossier. Modifier le schéma pour tenir compte de cette précision.
- 9.3 Modifier le schéma de la figure 9.23 de manière à y représenter
 - a) la réservation d'un ouvrage (ou d'un exemplaire ?) par un emprunteur;
 - b) l'historique des emprunts d'ouvrages (ou d'exemplaires ?).
- 9.4 Dessiner un graphe de populations représentatives pour le schéma de la figure 9.24. On y représentera une situation réelle, qu'on trouvera dans un horaire de chemin de fer.
- 9.5 Pourquoi dans le schéma de la figure 9.24 un agent conduit-il un voyage et non un train, comme il est d'usage ?
- 9.6 Le schéma de la figure 9.24 permet-il de résoudre le problème suivant ? Lorsqu'un conducteur X désire transmettre un colis à un autre conducteur Y, il dépose ce colis à une station par laquelle il passe, et par laquelle Y passera plus tard. X et Y se posent la question suivante à la date Z : où (quelle station) et à quelle date au plus tôt, Y pourra-t-il prendre livraison d'un colis que X veut lui transmettre. On suppose qu'un voyage ne s'étend pas sur plus d'une journée, et qu'un colis déposé à une date déterminée pourra être enlevé dès le lendemain.
- 9.7 Pourquoi dans le schéma de la figure 9.24 les sections ne sont-elles pas identifiées par leurs stations de départ et d'arrivée ?
- 9.8 Le schéma de la figure 9.24 représente des sections de ligne de telle manière que deux lignes empruntant le même tronçon de voie définissent deux sections différentes. Donner au concept de *tronçon de voie* une définition précise et modifier le schéma conceptuel de manière qu'il représente non seulement les sections mais également les tronçons de voie.
- 9.9 Toujours relativement à ce même schéma, affiner le concept de *train*, en considérant que celui-ci est constitué de motrices, de voitures de voyageur, de wagons de marchandise, de fourgons, *etc.*, dans un ordre déterminé.

Chapitre 10

Élaboration d'un schéma conceptuel

On aborde dans ce chapitre le problème de l'élaboration progressive d'un schéma conceptuel à partir d'un énoncé en langage courant. On montre le principe de décomposition de cet énoncé en propositions élémentaires, la traduction de ces propositions en structures Entité-association, l'intégration de ces structures dans le schéma en cours de constitution, la validation et la finalisation du schéma.

10.1 INTRODUCTION

Le processus d'élaboration du schéma conceptuel d'une base de données est à la fois crucial, car il conditionne la qualité de celle-ci, et complexe, en ce qu'il représente une activité de modélisation de situations issues de la réalité. Modéliser consiste à construire une représentation abstraite formelle, généralement de nature mathématique, des aspects de ces situations sur lesquels porte notre intérêt¹. Un modèle d'un **domaine d'application** nous permet de mieux comprendre celui-ci, de raisonner à son sujet et de construire des systèmes techniques, tels qu'une base de données ou une application informatique, qui représente, contrôle ou pilote ce domaine d'application.

1. Dans le domaine des bases de données, on désigne ces aspects sous le terme de *réel perçu* ou *univers du discours*. Nous lui avons préféré le terme plus spécifique de *domaine d'application*.

Le formalisme de représentation que nous utiliserons est le modèle Entité-association¹ présenté au chapitre 9. Il n'est pas possible dans le cadre de cet exposé de décrire tous les problèmes qui se posent dans l'élaboration d'un schéma conceptuel, ni de proposer une démarche de construction complète et systématique. La littérature abonde dans le domaine de la conception de base de données. On citera en particulier [Batini, 1992], [Bodart, 1994], [Halpin, 1995], [Nancy, 1996], [Blaha, 1998] [Akoka, 2001] pour ce qui est de la construction de schémas conceptuels. Nous présenterons cependant quelques éléments méthodologiques permettant au lecteur d'aborder des problèmes simples de manière systématique.

Les sources d'information utilisées lors de l'élaboration d'un schéma conceptuel peuvent être extrêmement diverses. En accord avec l'objectif de cet ouvrage, nous nous limiterons à un seul type d'information constitué d'un énoncé en langage courant. En toute généralité, on se servira également d'interviews, de textes légaux, de formulaires ou d'écrans s'il en existe, ou encore de l'observation du fonctionnement de l'organisation.

Le processus peut s'exprimer comme suit : *à partir d'un énoncé, construire un schéma Entité-association représentant les concepts et les faits exprimés dans cet énoncé, ou qui sont sous-jacents, concepts et faits au sujet desquels on désire enregistrer des informations*. Les principaux problèmes que nous allons rencontrer surgissent lorsque les propositions contenues dans l'énoncé sont incomplètes, redondantes, mutuellement contradictoires ou même fausses. En outre, il peut exister plusieurs façon de représenter un concept de l'énoncé.

On propose de procéder à l'élaboration du schéma conceptuel comme suit. On suppose que le processus est en cours, et qu'on dispose déjà d'un embryon de schéma qui traduit les concepts de l'énoncé jusqu'au point courant. On appellera cet embryon le *schéma courant* :

- on décompose l'énoncé en phrases, ou propositions, élémentaires;
- chaque proposition, si elle est pertinente pour le domaine d'application, est comparée aux types de faits déjà incorporés dans le schéma courant et, si le type de faits qu'elle exprime en est absent et est non contradictoire, il est traduit en termes du modèle Entité-association et incorporé dans ce schéma courant;
- on documente le schéma;
- on vérifie que le schéma est complet;
- on vérifie que le schéma ne contient pas d'erreurs.

1. Pour des raisons historiques malheureuses, tant le formalisme d'expression que l'expression particulière d'un domaine d'application à l'aide de ce formalisme sont appelés *modèles*. Afin d'éviter cette confusion nous parlerons du *modèle E-A* (formalisme) et du *schéma conceptuel E-A* (modèle d'un domaine d'application exprimé au moyen du formalisme E-A). Les spécialistes diront que le modèle E-A est en réalité un *méta-modèle*, c'est-à-dire un modèle de représentation de modèles.

10.2 DÉCOMPOSITION DE L'ÉNONCÉ

La technique proposée est similaire à celles qui sont décrites notamment dans [Bodart, 1994] et [Halpin, 1995]. L'énoncé est décomposé en propositions élémentaires décrivant chacune un concept ou un type de faits.

a) *Notion de proposition élémentaire*

La forme standard que nous privilégierons correspond à la composition suivante :

sujet *verbe* complément

Quelques exemples :

- tout client *a* un nom
- une commande *est passée par* un client
- un voyage *est effectué par* un train
- un service *traite* des dossiers

Les propositions de ce type, qu'on qualifiera de *binaires*, puisqu'elles comportent deux termes, affirment l'existence de deux concepts, représentés respectivement par le sujet et le complément, et d'un lien, représenté par le verbe, entre ces deux concepts. La première phrase, par exemple, exprime trois types de faits (qui ne sont pas nécessairement tous inconnus) :

- l'existence du concept de *client*;
- l'existence du concept de *nom*;
- l'existence d'un lien de *possession* entre client et nom.

D'autres formes plus simples encore sont envisageables, telles les propositions *unaires*, qui définissent essentiellement l'existence d'un concept :

- *il existe* des fournisseurs;
- *on s'intéresse aux* accidents.

On est souvent amené à reformuler certaines phrases qui définissent des propositions complexes ou multiples. C'est ainsi que la phrase :

- un employé *est identifié par* un numéro matricule *et caractérisé par* un nom *et* une adresse

sera éclatée en trois propositions élémentaires :

- un employé *est identifié par* un numéro matricule,
- un employé *est caractérisé par* un nom,
- un employé *est caractérisé par* une adresse.

Ou encore, la phrase

- Le coût du produit *devra* ...

recèle en fait deux propositions :

- le produit a un coût,
- le coût (du produit) *devra* ...

On veillera aussi à expliciter les raccourcis du langage que constituent les pronoms personnels ou possessifs, par exemple :

- il peut contracter une assurance (= *le client* peut contracter une assurance),
- son département ... (= *l'employé* a un département + le département ...)

b) Cardinalité

Pour les propositions du type «A *verbe* B», où A et B désignent deux concepts pertinents et où *verbe* indique l'existence d'un lien entre ces concepts, on cherchera à obtenir la réponse aux questions suivantes :

1. pour un exemplaire de A, combien trouve-t-on d'exemplaires de B, au minimum et au maximum;
2. pour un exemplaire de B, combien trouve-t-on d'exemplaires de A, au minimum et au maximum.

Ainsi, la proposition «une commande est passée par un client» sera-t-elle affinée par la réponse aux questions :

1. par combien de clients une commande est-elle passée? (réponse de 1 à 1)
2. combien de commandes un client peut-il passer? (réponse de 0 à plusieurs¹).

De même, la proposition «un client a une adresse» doit-elle être analysée de manière plus approfondie par les questions :

1. combien un client a-t-il d'adresses?
2. combien de clients peut-on trouver à une adresse déterminée?

On sera attentif à la différence d'interprétation des pronoms *un* ou *une* dans les deux dernières phrases. Ainsi, dans la proposition «une commande est passée par un client», **une** commande signifie **toute** commande, tandis que **un** client est interprété comme **un seul** client.

La réponse à de telles questions se trouve parfois dans la formulation même de la proposition. Des locutions telles que *tout*, *certain*, *chaque*, les formes verbales utilisant *pouvoir* ou *devoir*, les expressions *au moins un*, *un seul*, *au plus un*, *des* ou *les*, permettent de déterminer l'information recherchée.

Dans d'autres cas nous ferons appel, soit à des informations complémentaires (interview dirigée), soit à notre connaissance du domaine d'application, soit encore au simple bon sens.

1. Le terme *plusieurs* indique un nombre quelconque arbitrairement grand. On parlerait d'*infini* si les mondes qu'il est possible de décrire n'étaient pas finis.

c) Propositions générales et propositions particulières

Les propositions énoncées jusqu'ici sont générales, en ce sens qu'elles sont vraies pour tous les faits de même nature, présents et futurs. Elles s'expriment au niveau des concepts généraux, et non au niveau des exemples et des cas spécifiques. Une proposition telle que

- toute voiture a un numéro minéralogique

spécifie un type de faits général dans le domaine d'application choisi, tandis que la proposition

- ma voiture a le numéro minéralogique HBG910-82

est, elle, *particulière*. Elle constitue un fait élémentaire qui n'est qu'un exemple de la première¹.

Le terme « voiture » de la première proposition désigne un **concept**, alors que le terme « ma voiture » désigne un objet concret qui relève de ce concept, c'est-à-dire qui en est une **occurrence**, un **exemplaire** ou une **instance**. Il en est de même de « numéro minéralogique », qui désigne un concept et de « HBG910-82 » qui en est une occurrence.

Les propositions particulières ne sont pas à modéliser comme telles (ce rôle sera assigné aux données). Cependant, elles ne manquent pas d'intérêt dans la mesure où elle peuvent suggérer, par induction, l'existence d'un type de faits plus général : si ma voiture a un numéro, n'en serait-il pas de même de toutes les voitures ?

d) Attention aux propositions complexes irréductibles !

Certaines propositions *non binaires* ne peuvent, sans qu'on perde de l'information, être réduites aussi simplement que nous l'avons suggéré en propositions binaires élémentaires.

Considérons par exemple la proposition suivante, que nous qualifierons de *ternaire*, puisqu'elle connecte trois termes :

- les clients achètent des produits chez des fournisseurs

Si on admet que les clients ont la possibilité d'acheter n'importe quel produit chez n'importe quel fournisseur, alors *on ne peut réduire* cette proposition à celles-ci :

- les clients achètent des produits,
- les fournisseurs vendent des produits,
- les clients achètent chez des fournisseurs.

Montrons-le par un exemple. On considère les trois faits suivants comme des exemples de la proposition complexe initiale :

1. Notons que le terme de proposition ne fait pas référence au calcul propositionnel (section 5.2.6), auquel cas les propositions générales devraient s'appeler prédicats.

JEAN achète du SUCRE chez MIGRO
 ANNE achète du SUCRE chez UNIC
 JEAN achète du SEL chez UNIC

En réduisant chaque fait à trois faits élémentaires, on obtient :

JEAN achète du SUCRE
 ANNE achète du SUCRE
 JEAN achète du SEL

 MIGRO vend du SUCRE
 UNIC vend du SUCRE
 UNIC vend du SEL

 JEAN achète chez MIGRO
 ANNE achète chez UNIC
 JEAN achète chez UNIC

Ces neuf faits binaires ne sont pas équivalents aux trois faits ternaires d'origine. Imaginons qu'ils le soient. Dans ce cas, ils doivent permettre de retrouver les trois faits d'origine *et eux seulement*. Par exemple, à partir de

JEAN achète du SUCRE
 MIGRO vend du SUCRE
 JEAN achète chez MIGRO,

on pourrait déduire le fait d'origine :

JEAN achète du SUCRE chez MIGRO

Mais selon le même procédé, on pourrait alors aussi déduire de

JEAN achète du SUCRE
 UNIC vend du SUCRE
 JEAN achète chez UNIC

que

JEAN achète du SUCRE chez UNIC,

ce qui est manifestement faux. En poursuivant ce mécanisme de composition, on retrouverait ainsi, non seulement tous les faits d'origine, mais également des propositions fausses¹.

1. On suggère au lecteur de représenter graphiquement les propositions binaires par des arcs reliant leurs deux termes et les faits ternaires par des étoiles à trois branches reliant leurs trois termes. On voit alors clairement comment trois faits binaires formant un triangle peuvent être transformés en un fait ternaire, et réciproquement. On voit tout aussi clairement comment un triangle accidentel (dont les trois côtés ne sont pas issus d'une même étoile d'origine) se transforme en une fausse étoile.

On en conclut que les faits élémentaires dérivés, tout en étant corrects, sont moins précis que les faits complexes d'origine, et qu'ils ne peuvent en aucune manière les remplacer. On ne peut donc pas décomposer comme on vient de le voir.

Que faire alors si on désire s'en tenir à des propositions élémentaires¹? La solution est de mettre en évidence un concept nouveau, qui représente la proposition complexe tout entière et qui serait dans le cas présent le *concept d'achat*, qui concerne un produit, qui est effectué par un client, chez un fournisseur. On transforme le verbe acheter en acte d'achat, qui correspond dès lors à un concept spécifique qui peut intervenir dans un nombre quelconque de propositions.

À partir de ces *quatre concepts* d'achat, de produit, de client et de fournisseur, on reformule la proposition d'origine sous la forme :

- les clients font des achats de produits chez des fournisseurs

qui donne trois propositions binaires basées sur la notion d'achat (= acte d'acheter) :

- un achat est effectué par un client,
- un achat concerne un produit,
- un achat s'effectue chez un fournisseur.

Les trois exemples de faits ternaires d'origine peuvent alors s'exprimer comme suit :

l'achat X est effectué par JEAN

l'achat Y est effectué par ANNE

l'achat Z est effectué par JEAN

X concerne du SUCRE

Y concerne du SUCRE

Z concerne du SEL

X s'effectue chez MIGRO

Y s'effectue chez UNIC

Z s'effectue chez UNIC

On vérifie aisément que, par composition (on forme des étoiles à trois branches dont le centre est un achat, qu'on élimine ensuite), on retrouve les trois faits d'origine, et eux seulement. Les dénominations X, Y et Z sont arbitraires; elles ont été choisies aux fins de l'exemple uniquement. Comme la construction du schéma ne se préoccupe pas des faits élémentaires, mais seulement des types de faits, nous n'aurons pas à nous préoccuper de ces dénominations.

e) Les propositions non binaires réductibles

En revanche, certaines propositions en apparence irréductibles doivent impérativement être décomposées en propositions binaires sans passer par un concept intermédiaire comme décrit ci-dessus.

1. Il existe des modèles plus puissants qui permettent de représenter explicitement des faits irréductibles basés sur plus de deux concepts. Cfr. les types d'associations N-aires de la section 9.9.

Considérons à titre d'exemple la phrase suivante :

- un trajet est organisé entre une ville de départ et une ville d'arrivée.

On observe que, contrairement à la situation précédente, à un trajet correspond *une seule* ville de départ et *une seule* ville d'arrivée. Cette proposition ternaire est donc équivalente aux **deux** propositions binaires suivantes :

- un trajet a une ville de départ.
- un trajet a une ville d'arrivée.

On réduira de la même manière à un couple de propositions binaires chacun des énoncés complexes suivants :

- un médecin dirige un service dans son hôpital.
- les entreprises inscrivent certains de leurs employés à des sessions de formation.

La première proposition concerne trois concepts : médecin, service et hôpital. A un médecin correspond *un seul* service et à un service correspond *un seul* hôpital. On peut décomposer l'énoncé en deux propositions binaires symbolisées par les couples (médecin, service) et (service, hôpital)¹.

De même, la seconde proposition suppose qu'un employé travaille dans *une seule* entreprise, ce qui permet d'y repérer les deux propositions binaires (employé, entreprise) et (employé, session).

En revanche, la proposition étudiée plus haut,

- les clients achètent des produits chez des fournisseurs

n'est pas décomposable parce qu'aucune des affirmations suivantes n'est correcte :

- un client n'achète qu'un seul produit,
- un client n'achète que chez un seul fournisseur,
- un produit n'est acheté que par un seul client,
- un produit n'est acheté que chez un seul fournisseur,
- un fournisseur ne vend qu'à un seul client.
- un fournisseur ne vend qu'un seul produit

1. Ces raisonnements sont basés sur le principe de décomposition décrit dans la Section 3.9.3. En effet, une proposition ternaire peut être vue comme une table à trois colonnes dont chaque ligne représente une proposition particulière (*une instance*). Si la proposition dont on discute est représentée par la formule dirige(médecin, service, hôpital) analogue au schéma d'une table, dire qu'à un service correspond un seul hôpital, revient à affirmer qu'il existe dans dirige une dépendance fonctionnelle service \rightarrow hôpital. D'où la décomposition suggérée.

10.3 PERTINENCE D'UNE PROPOSITION

Certaines propositions apportent de toute évidence une contribution au schéma conceptuel. D'autres en revanche doivent être considérées comme du *bruit* dans ce processus et sont ignorées. Certaines cependant peuvent cacher des structures qu'il faut mettre en lumière par une analyse plus approfondie.

Considérons les fragments d'énoncés suivants, dans un domaine d'application relatif à une compagnie d'assurance automobile.

Exemple 1

- un véhicule est couvert par une police

Cette proposition est certainement pertinente et doit être retenue.

Exemple 2

- on enregistre le véhicule dès que la prime est payée

Cette proposition décrit une contrainte d'exécution d'activités de gestion. Elle sera certainement utile lors de la conception des programmes informatiques, mais semble moins directement exploitable ici. Cependant, une analyse plus approfondie pourrait conduire à en retenir les propositions dérivées suivantes :

- un véhicule a une date d'enregistrement,
- une prime a une date de paiement.

Il serait alors aisé de vérifier, dans un programme, la règle d'antériorité exprimée par la proposition d'origine.

Exemple 3

- les véhicules que la compagnie assure ...

Étant donné qu'on ne s'intéresse qu'aux véhicules assurés par la compagnie et que cette (unique) compagnie constitue, ou contient, le domaine d'application, cette proposition n'apporte aucune information, sinon l'existence, probablement déjà connue, de la notion de véhicule. Il serait sans utilité¹ d'en conclure que :

- la compagnie assure des véhicules

10.4 REPRÉSENTATION D'UNE PROPOSITION

Rappelons que la plupart des propositions mettent en évidence des concepts et/ou des liens entre concepts. On cherchera donc à représenter chaque concept cité s'il

1. Encore que la question de la représentation du domaine d'application lui-même sous la forme d'un type d'(une seule) entité peut être pertinente dans la mesure où ce domaine peut avoir des propriétés utiles. On pense par exemple à la raison sociale, l'adresse, l'URL du site ou le logo de l'entreprise.

n'est pas déjà exprimé dans le schéma courant, ainsi que le lien entre ces concepts. Nous donnerons ci-dessous quelques règles simples et générales couvrant les situations les plus fréquentes. On doit cependant admettre que le processus ne peut être entièrement formalisé et que le bon sens est souvent le meilleur guide en cette matière.

a) Nouveau type d'entités et son attribut

D'une manière générale, un concept nouveau se représentera par un type d'entités s'il apparaît comme important, ou par un attribut s'il apparaît comme une simple propriété d'un autre concept. Dans l'exemple ci-dessous, il est raisonnable de représenter les dossiers par un type d'entités DOSSIER et les titres par un attribut TITRE attaché au premier (figure 10.1) :

- tout dossier possède un titre



Figure 10.1 - Traduction d'une proposition exprimant un concept nouveau et une de ses propriétés

b) Type d'associations entre types d'entités existants

Une proposition qui établit un lien entre deux concepts déjà représentés par des types d'entités se traduira par un type d'associations entre ces derniers. Ainsi, la proposition suivante est-elle intégrée au schéma courant comme indiqué à la figure 10.2. On affinera la représentation en déterminant la classe fonctionnelle du type d'associations (ici *plusieurs à plusieurs*) et son caractère obligatoire/facultatif pour chacun des deux types d'entités.

- les clients peuvent acheter des produits



Figure 10.2 - La proposition se traduit par un type d'associations entre types d'entités existant

c) *Attribut d'un type d'entités existant*

Une proposition qui établit un lien entre un concept déjà représenté par un type d'entités et une propriété nouvelle de ce concept se traduira en un nouvel attribut attaché à ce type d'entités. La figure 10.3 illustre l'affectation de la nouvelle propriété *Modèle* au type d'entités *VOITURE*, suite à l'interprétation de la proposition suivante :

- chaque voiture est d'un modèle déterminé

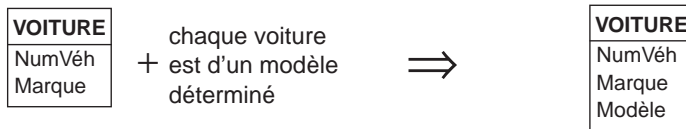


Figure 10.3 - La proposition se traduit en affectation d'un attribut à un type d'entités existant

d) *Nouveau type d'entités et ses deux attributs*

Une proposition peut exprimer un lien entre deux concepts qui se traduisent naturellement en deux attributs. On représentera la proposition par un nouveau type d'entités auquel on affecte ces attributs. La proposition suivante, au sujet de laquelle on suppose qu'aucun concept n'a encore été mis en évidence, se représentera par les deux attributs *Date* et *Montant* qu'on affectera à un nouveau type d'entités représentant les *dépenses* (figure 10.4).

- le montant est dépensé à une date déterminée

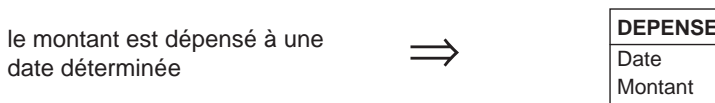


Figure 10.4 - Représentation d'un lien entre concepts par un type d'entités

e) *Restructuration*

L'intégration d'un nouveau type de faits au schéma courant peut poser des problèmes qui ne peuvent être résolus que par la **restructuration** d'une partie de ce schéma. Par exemple, certaines situations semblent réclamer des modifications non licites du schéma courant, telles que

1. l'affectation d'un attribut AA à un attribut A existant (un attribut ne peut avoir d'attributs);
2. l'établissement d'un lien entre un type d'entités E1 existant et un attribut A existant qui appartient déjà à un autre type d'entités E2 (un attribut ne peut être associé qu'à un seul type d'entités);
3. l'affectation d'un attribut A à un type d'associations R existant (un type d'associations ne peut avoir d'attributs);

4. ajouter un troisième type d'entités à un type d'associations existant (un type d'associations ne peut avoir plus de deux rôles).

Les problèmes 1 et 2 pourraient se résoudre si l'attribut A était au préalable transformé en un type d'entités. Les problème 3 et 4 pourraient se résoudre si le type d'associations R était au préalable transformé en un type d'entités.

Nous suggérerons donc deux *transformations* simples qui permettent de résoudre les problèmes évoqués ci-dessus.

La *première transformation* (figure. 10.5) remplace un type d'associations (a acheté) par un type d'entités (ACHAT) en liaison avec les types d'entités d'origine (CLIENT et PRODUIT). Le nom a été modifié pour des raisons de lisibilité, le verbe ayant été remplacé par le substantif correspondant.

Il est à présent possible d'affecter un attribut à ACHAT, tel que celui qui serait suggéré par la proposition : ... *la quantité totale de chaque produit achetée par chaque client* ... En effet, cette quantité ne peut être affectée à CLIENT (ni à PRODUIT), auquel cas chaque entité recevrait plusieurs valeurs qu'on ne saurait relier aux produits (resp. clients) auxquels chacune est relative. La quantité dépend bien de l'association entre un client et un produit, et l'attribut Quantité qui la représente doit être affecté au nouveau type d'entités ACHAT.

On veillera à bien préciser la signification de ce nouveau type d'entités. Dans l'exemple présenté, un achat est *le fait qu'un client déterminé a déjà acheté un produit déterminé*.

Il y a un seul client et un seul produit par achat, mais il peut exister plusieurs achats par client et plusieurs achats par produit. En revanche, il n'existe qu'un seul lien entre un client et un produit, d'où l'identifiant du nouveau type d'entités¹.

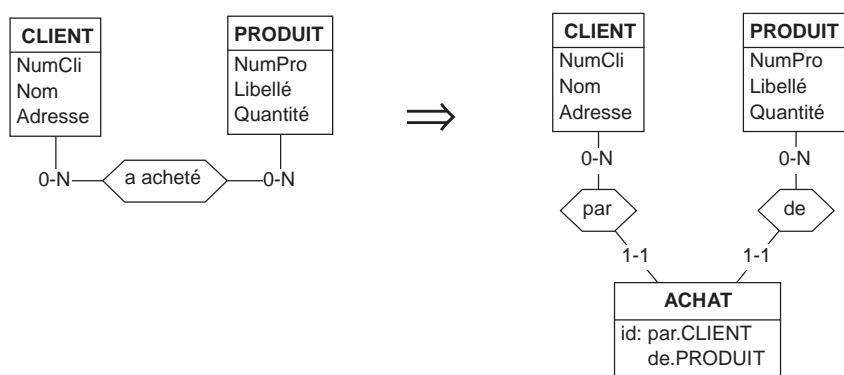


Figure 10.5 - Transformation du type d'associations *a acheté* en type d'entités ACHAT

On vérifiera, à l'aide de l'exemple de la figure 10.6, que les deux schémas ont exactement le même contenu informationnel, au sens de la section 9.6 : il n'existe aucune question à laquelle l'un des schémas pourrait répondre et pas l'autre.

1. On notera l'existence de types d'entités sans attributs.

On vérifiera, à l'aide de l'exemple de la figure 10.6, que les deux schémas ont exactement le même contenu informationnel, au sens de la section 9.6 : il n'existe aucune question à laquelle l'un des schémas pourrait répondre et pas l'autre.

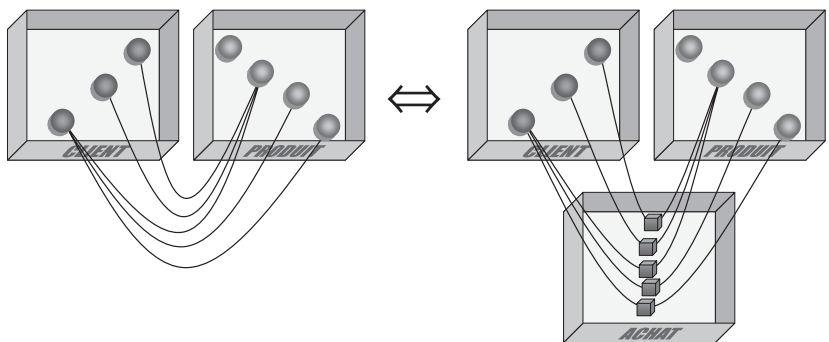


Figure 10.6 - Illustration de la transformation d'associations en entités

La deuxième transformation (figure 10.7) extrait un attribut (ou même plusieurs) d'un type d'entités pour former un nouveau type d'entités possédant ces attributs. Les deux types d'entités sont liés par un type d'associations *un-à-plusieurs*¹.

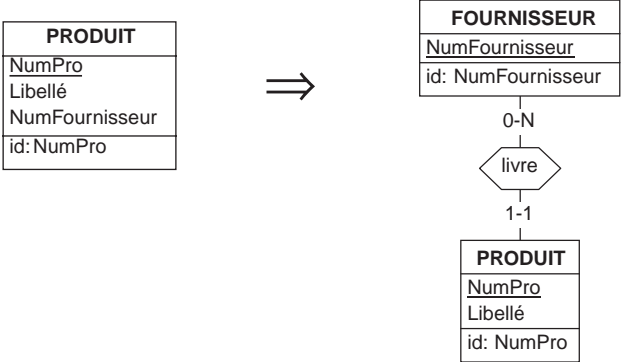


Figure 10.7 - Transformation d'un attribut en type d'entités

On observe que la mise en évidence du concept de fournisseur *via* le type d'entités FOURNISSEUR permet désormais de lui affecter des attributs et de le relier à d'autres types d'entités, ce qui était impossible dans le schéma de gauche. On peut à titre d'exemple représenter aisément le nom et l'adresse des fournisseurs.

1. Si l'attribut était identifiant, ce type d'associations serait *un-à-un* (à démontrer). D'autre part, pour que cette transformation donne un schéma strictement équivalent à celui de gauche, il aurait fallu que la cardinalité de FOURNISSEUR soit *1-N*. En effet, selon le schéma de gauche, un fournisseur ne peut être représenté qu'associé à un produit, alors que le schéma de droite tolère des fournisseurs n'offrant aucun produit. On admettra cependant cette légère distorsion, puisque nous n'utilisons pas cette cardinalité dans cet ouvrage.

On sera attentif à la signification du nouveau type d'associations. On observera en particulier qu'il y a un et un seul fournisseur par produit (de même qu'il y avait une et une seule valeur de NumFournisseur par entité PRODUIT), mais qu'il peut y avoir plusieurs produits par fournisseur (de même que plusieurs entités PRODUIT pouvaient se partager la même valeur de NumFournisseur). Si NumFournisseur avait été facultatif dans le schéma de gauche, alors livre aurait été facultatif pour PRODUIT dans le schéma de droite.

Application.

Considérons le schéma de la figure 10.8. Il représente notamment le fait que *des unités fabriquent des types de pièces* et qu'*un type de pièces est stockée dans un dépôt*.

Supposons ensuite que les deux propositions suivantes soient à représenter dans ce schéma :

- 1. *on connaît la quantité journalière de chaque type de pièces fabriquée par chaque unité;*
- 2. *chaque dépôt a une localisation et une capacité.*

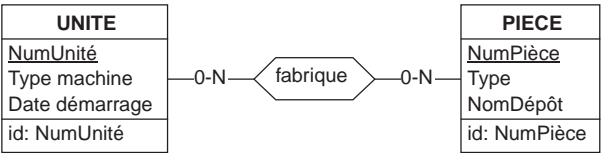


Figure 10.8 - Le schéma avant enrichissement

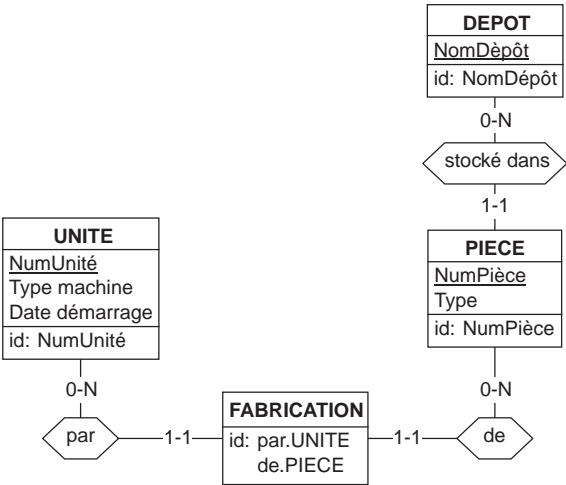


Figure 10.9 - Le schéma est transformé afin d'accueillir les nouveaux attributs

La première proposition suggère d'affecter l'attribut Quantité au type d'associations fabrique tandis que la seconde correspond à l'ajout des attributs Localisation et

Capacité à l'attribut Dépôt. Ces actions n'étant pas admises dans notre modèle, on peut les rendre possibles grâce à la transformation préalable du type d'associations fabrique et de l'attribut Dépôt (figure 10.9).

Il est alors permis d'ajouter les trois nouveaux attributs (figure 10.10).

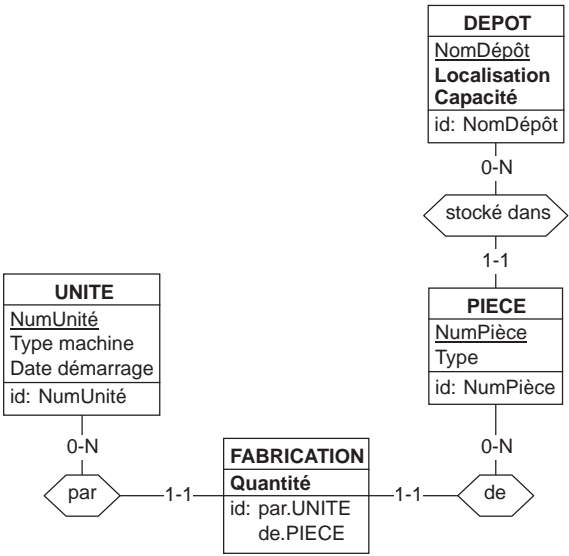


Figure 10.10 - Les nouveaux attributs ont été ajoutés

f) Propriétés multiples

Un problème particulier peut se poser lors de la mise en évidence de **propriétés multiples** attachées à un concept. Considérons deux cas représentatifs :

- une personne a des prénoms
- un ouvrage a des mots clés

Un concept majeur, représenté vraisemblablement par un type d'entités tel que **PERSONNE** et **OUVRAGE**, est caractérisé par une propriété (prénom, mot clé) se présentant non pas comme une simple valeur, mais comme un ensemble ou une liste de valeurs. Une telle situation nous suggère une représentation des caractéristiques sous la forme d'un type d'entités (**PRENOM**, **MOT-CLE**) attaché à celui du concept majeur par un type d'associations (a pour prénom, décrit).

Dans le premier cas (une personne a des prénoms), cette représentation peut paraître particulièrement lourde. Un type d'entités **PRENOM** semble donner à une simple propriété un statut privilégié qui n'est normalement réservé qu'aux concepts majeurs du domaine d'application. On pourrait lui préférer une représentation plus modeste (figure 10.11), soit en quatre attributs, représentant chacun un des quatre prénoms de la personne, et dénommés Prénom1, Prénom2, ..., Prénom4 par exemple, soit sous la forme d'un attribut unique qui représente la liste des prénoms,

et dénommé Prénoms. Ces deux dernières représentations induisent cependant des problèmes qui nous feront préférer la solution initiale, malgré sa lourdeur apparente.

PERSONNE	PERSONNE
NumPers: num (8)	NumPers: num (8)
Nom: char (32)	Nom: char (32)
Prénom1[0-1]: char (16)	Prénoms: char (64)
Prénom2[0-1]: char (16)	
Prénom3[0-1]: char (16)	
Prénom4[0-1]: char (16)	

Figure 10.11 - A éviter : deux représentation problématiques d'une propriété multiple

Dans le second cas (un ouvrage a des mots clés...), on peut envisager des solution similaires, sous la forme d'un type d'entités MOT-CLE, d'une suite d'attributs Mot-clé1, Mot-clé2, ... ou d'un attribut unique Mots-clés. Si le nombre de mots clés par ouvrage n'est pas limité, on adoptera obligatoirement une représentation sous la forme d'un type d'entités.

Il faut cependant être attentif à la signification de ce type d'entités. En effet, deux représentations sont possibles, qu'il faut distinguer sans ambiguïté.

Représentation des valeurs

Selon la première technique on représente *chaque valeur distincte de mot clé* (ou de *prénom*) par une entité distincte du type MOT-CLE (figure 10.12, gauche), et on établit un type d'associations plusieurs-à-plusieurs, décrit, entre OUVRAGE et MOT-CLE. Le mot clé *jardinage* sera représenté par une unique entité, dont la valeur de l'attribut *Libellé* est « jardinage », quel que soit le nombre d'ouvrages disponibles sur ce thème.

Représentation des instances

Selon la seconde technique (figure 10.12, droite), c'est *la présence de chaque mot clé* décrivant chaque ouvrage qui est représentée par une entité distincte d'un type qu'on dénommera DESCRIPTION et qui aura un attribut *Libellé*. C'est ainsi qu'il y aura autant d'entités DESCRIPTION dont la valeur de *Libellé* est « jardinage », qu'il y a d'ouvrages de jardinage. Les mots clés d'un ouvrage étant distincts, on en déduit l'identifiant de DESCRIPTION¹.

1. Le choix des noms dans les deux schémas n'est pas le fruit du hasard. On peut en effet démontrer que ces schémas sont (presque) équivalents (voir remarque ci-dessous), bien qu'ils conduisent à des structures de tables différentes. Partant du schéma de droite, on peut transformer *Libellé* en type d'entités MOT-CLE d'attribut *Libellé* selon la technique de la figure 10.7. Le type d'entités DESCRIPTION apparaît alors comme l'expression d'un type d'associations *plusieurs-à-plusieurs* qui pourrait s'appeler *décrit* (figure 10.5). Réexprimant ce dernier, on obtient le schéma de gauche.

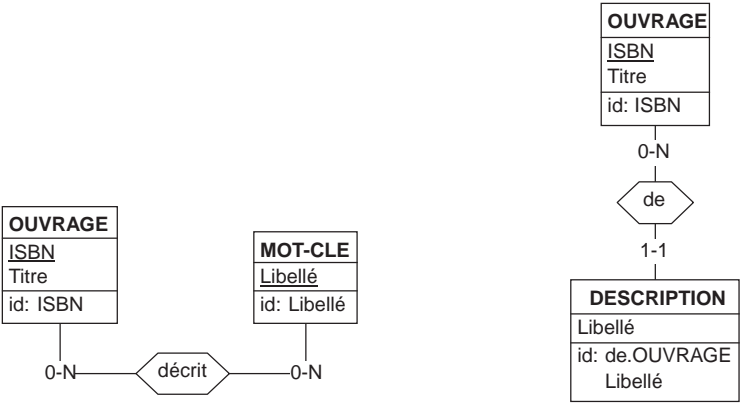


Figure 10.12 - Deux techniques de représentation d'une propriété multiple par un type d'entités¹

g) Recommandations pratiques

Nous terminerons cette section par quelques recommandations utiles.

- Si un concept n'est décrit que par une de ses caractéristiques, on envisagera de ne pas le représenter par un type d'entités. Si la seule mention du concept de *fournisseur* est celle de son *nom* attaché à la description des *produits* qu'il livre, il peut être suffisant de le représenter par le simple attribut *NomFournisseur* de *PRODUIT*. En cas d'hésitation, il est toujours possible de choisir la solution la plus expressive et la plus riche qui consiste à définir un type d'entités *FOURNISSEUR*, doté de l'attribut *Nom* et en association avec *PRODUIT*.
- Si en revanche, on cite au moins deux caractéristiques d'un concept, il est préférable de le représenter par un type d'entités doté des attributs correspondant à ces caractéristiques. Si on cite le nom du fournisseur à un certain endroit et son adresse à un autre, il est préférable de définir un type d'entités *FOURNISSEUR*.
- Si une même caractéristique d'un concept est citée à plusieurs endroits, il est préférable de représenter ce concept par un type d'entités doté de l'attribut correspondant à cette caractéristique. Si le *nom* du *fournisseur* apparaît à plus d'un endroit, il est probable qu'un type d'entités *FOURNISSEUR* doté d'un attribut *NOM* permettra d'obtenir un schéma plus clair.
- Quand on hésite sur le type d'entités auquel il faut associer un attribut, on choisira une solution qui n'entraîne pas de redondance (il y a redondance si une information élémentaire est présente plusieurs fois). Par exemple, l'adresse d'un client peut faire l'objet d'un attribut *Adresse* qu'on peut attacher au type d'entités

1. Ces deux schémas sont *raisonnablement* équivalents. Il existe une petite différence : à gauche, un mot clé peut exister indépendamment des ouvrages alors qu'à droite, seuls les mots clés effectivement attachés à au moins un ouvrage peuvent être représentés. Pour garantir l'équivalence, la cardinalité de MOT-CLE à gauche aurait dû être 1-N.

- CLIENT ou au type d'entités COMMANDE. L'associer à COMMANDE entraîne une redondance puisqu'on répétera l'adresse du client autant de fois que ce dernier aura passé de commandes. Il convient donc d'associer l'attribut ADRESSE au type d'entités CLIENT. On reprendra ce problème à la section 10.10.
- Si parmi les attributs d'un type d'entités on repère un groupe homogène en relation étroite, il convient probablement de l'extraire pour en constituer un type d'entités autonome. Supposons que dans le type d'entités COMMANDE, on observe les attributs NumClient, NomClient et AdresseClient. Il est clair que NomClient et AdresseClient dépendent plus de NumClient que de COMMANDE qui les accueille. On pensera alors à extraire ces attributs et à les affecter à un nouveau type d'entités que l'on nommera CLIENT par exemple. On établira aussi un nouveau type d'associations entre COMMANDE et CLIENT. La situation initiale pose en outre un problème de redondance semblable à celui qu'on a évoqué au point précédent.

10.5 NON-REDONDANCE DES PROPOSITIONS

Une proposition élémentaire exprime des types de faits sous la forme de l'existence de concepts, l'existence d'un lien entre ceux-ci, ou les deux. Ce (ou ces) types de faits ont peut-être déjà été exprimés et sont donc déjà représentés dans le schéma courant. Avant d'ajouter leur représentation au schéma courant, on s'assure qu'elle ne s'y trouve pas déjà. On appellera **redondance** le phénomène selon lequel une construction du schéma courant et une proposition expriment le même type de faits. Nous distinguerons divers cas de figure.

a) Redondance explicite

Dans les cas simples, la redondance apparaît explicitement, schéma et proposition exprimant le type de faits de manière similaire. Dans la figure 10.13, il est clair que la proposition est déjà représentée dans le schéma courant, et doit être ignorée.

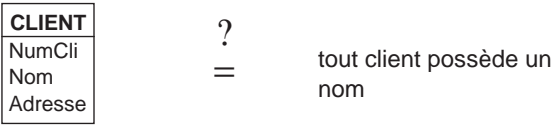


Figure 10.13 - La proposition est explicitement redondante par rapport au schéma

b) Variantes d'expression

Une proposition peut exprimer un type de faits déjà représenté dans le schéma courant, mais sous une forme différente. Les variantes voie active/voie passive en sont un exemple. Dans la figure 10.14, le schéma indique qu'*un client possède des véhicules*; la proposition exprime la même idée, mais relativement aux véhicules. Elle est donc redondante.

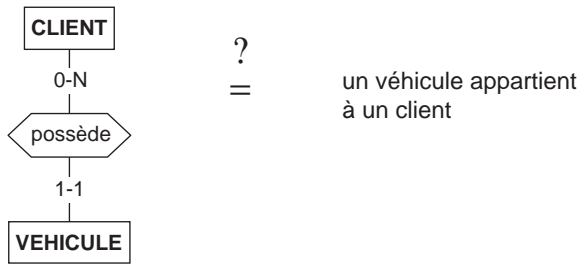


Figure 10.14 - La proposition décrit une variante du type de faits exprimé dans le schéma

c) *Redondance indirecte*

La redondance peut être plus difficile à détecter car elle concerne des types de faits qui ne sont pas explicitement exprimés dans le schéma courant, mais qui peuvent s'en déduire. La figure 10.15 montre un premier exemple. La proposition parle des employés des départements, alors que le schéma, qui inclut pourtant les types d'entités **DEPARTEMENT** et **EMPLOYE**, n'exprime aucun lien direct entre ceux-ci. A l'analyse, cependant, il apparaît que les employés d'un département sont en fait les employés des services de ce département. Il y a redondance indirecte.

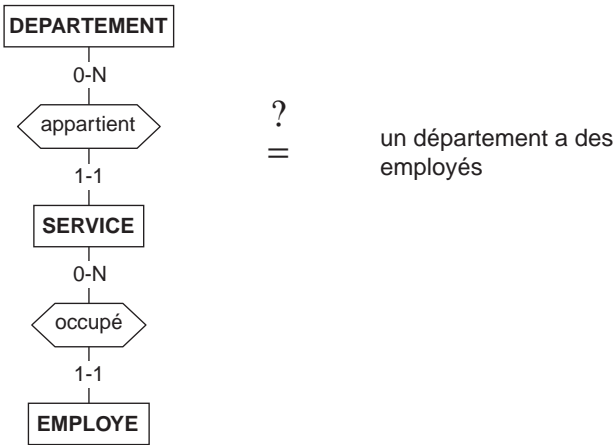


Figure 10.15 - La proposition recouvre un type de faits dérivable (par composition) des constructions de schéma courant

L'exemple de la figure 10.16 montre une proposition qui exprime l'existence d'un attribut Montant total, qui s'avère dérivable à partir des attributs Qcom et Prix Unitaire déjà présents dans le schéma courant.

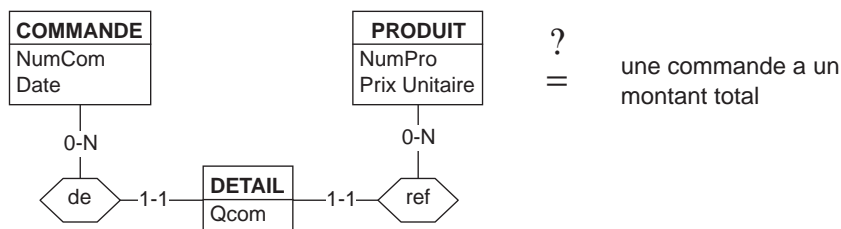


Figure 10.16 - La proposition exprime une propriété des commandes qui est dérivable des constructions existant dans le schéma courant

d) Redondance apparente

En revanche, la redondance peut n'être qu'apparente, et par conséquent inexistante. L'exemple de la figure 10.17 suggère que le concept d'adresse de client est déjà présent dans le schéma courant. A l'analyse, il apparaît qu'il n'en est rien, cette adresse étant celle d'expédition, qui peut être différente de l'adresse de facturation. Il faut donc ajouter au type d'entités **CLIENT** un nouvel attribut Adresse Facturation.

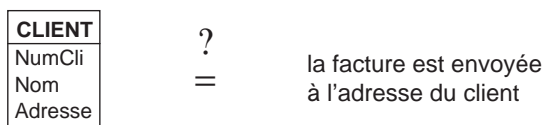


Figure 10.17 - Il faut distinguer *adresse d'expédition* (dans le schéma) et *adresse de facturation* (dans la proposition)

e) Synonymes et homonymes

Dans l'analyse du domaine d'application, deux problèmes vont apparaître, qui devront être résolus par une compréhension profonde du domaine d'application : les synonymes et les homonymes.

Des termes différents apparaissant dans des propositions sont **synonymes** s'ils désignent le même concept du domaine d'application. Ainsi en serait-il probablement des termes *assuré* et *client* dans le contexte des assurances évoqué dans cet ouvrage. On choisira une et une seule dénomination pour chaque concept.

Des termes sont **homonymes** lorsqu'ils sont identiques, mais qu'ils désignent des concepts différents dans certaines parties du domaine d'application. Le terme d'*adresse*, déjà évoqué ci-dessus, recouvrira en général des concepts distincts dans le domaine de la distribution : *adresse* du client qui commande, *adresse* d'expédition, *adresse* de facturation. On veillera à mettre en évidence ces variétés et à leur donner des dénominations distinctes.

Citons deux autres exemples : dans un contexte hospitalier, l'*unité* de soins et l'*unité* de fabrication de produits pharmaceutiques; un *centre* de frais (comptabilité) et un *centre* administratif.

10.6 NON-CONTRADICTION DES PROPOSITIONS

Toute construction élémentaire d'un schéma conceptuel est l'expression formelle d'une proposition élémentaire et peut être réinterprétée sous la forme de cette proposition. On peut donc confronter toute proposition candidate aux constructions élémentaires du schéma courant. Cette confrontation peut conduire à une contradiction. Une source fréquente de contradiction est l'évaluation du nombre d'éléments en liaison dans les propositions du type « A verbe B ». A titre d'exemple, si le schéma affirme qu'*une commande peut être passée par plusieurs clients*, il entre en contradiction avec une proposition candidate qui dirait qu'*une commande est passée par un seul client*. De même, une proposition qui suggérerait qu'un contrat est identifié par son numéro serait en contradiction avec le schéma de la figure 9.16, qui indique que ce numéro est unique pour les contrats d'un même client seulement.

Dans les deux exemples de la figure 10.18, les propositions sont en contradiction avec le schéma courant. En effet (exemple supérieur), le schéma indique qu'il ne peut exister qu'une facture par commande tandis que la proposition suggère qu'il puisse y en avoir plusieurs. Selon l'exemple inférieur, si tout service a un directeur et si le directeur d'un service est un des employés de ce service, alors ce dernier doit avoir au moins un employé. Il y a donc contradiction entre cette conclusion et la cardinalité 0-N de SERVICE dans travaille.

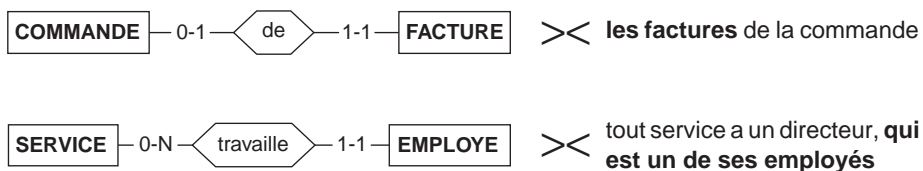


Figure 10.18 - Deux exemples de contradiction : (1) une commande a-t-elle une ou plusieurs factures ? (2) un service doit avoir un directeur, qui est un employé, mais n'a pas nécessairement d'employés !

On peut résoudre une contradiction par diverses techniques :

- en rejetant l'éventuelle proposition erronée, le cas échéant ;
- si aucune des propositions n'est erronée, en faisant le choix de la plus générale : *une seule facture* étant un cas particulier de *plusieurs*, on retiendra cette dernière proposition ;
- en modifiant le schéma de manière que les deux propositions soient en accord : on admet qu'il existe une classe particulière de services : les *services actifs*, dotés d'employés et d'un directeur.

La détection systématique des contradictions et les techniques de résolution constituent un domaine complexe qui relève, entre autres choses, de la théorie de l'intégration de schémas [Spaccapietra, 1992] [Batini, 1992]. Nous nous contenterons d'attirer l'attention sur le problème.

10.7 LES CONTRAINTES D'INTÉGRITÉ

Les contraintes d'intégrité formalisent certaines propriétés importantes du domaine d'application que les données devront respecter. Le plus souvent, c'est l'énoncé lui-même qui indiquera ces contraintes de manière explicite. Il sera parfois nécessaire de recourir à des sources d'information complémentaires telles qu'une conversation avec les utilisateurs, la connaissance que nous pourrions avoir du domaine ou encore le simple bon sens.

Pratiquement, on examinera avec soin chaque type d'entités, chaque type d'associations et chaque attribut afin d'y relever les contraintes à retenir. Il faudra rester réaliste dans ce repérage. En effet, chaque contrainte déclarée entraîne un bénéfice et un coût. Le *bénéfice* est une plus grande précision des données et un meilleur contrôle de la qualité. Le *coût* sera celui de la complexité et des ressources informatiques nécessaire à la validation des données qui doivent se conformer à cette contrainte. Les opérations de mise à jour seront d'autant plus coûteuses qu'il y aura de contraintes à vérifier.

a) Les contraintes statiques (section 9.5.1)

Les principales contraintes statiques sont les identifiants, les attributs obligatoires, et les contraintes de cardinalités portant sur les rôles. D'autres contraintes pourraient s'avérer utile, auquel cas on les indiquera par des annotations ajoutées au schéma conceptuel (figure 10.19).

b) Les contraintes dynamiques (section 9.5.2)

Les contraintes sur les changements autorisés de valeurs peuvent être extrêmement variées, et une analyse approfondie peut amener à en définir un grand nombre. Leur validation par le SGBD fera appel au mécanisme des déclencheurs, dont on sait qu'ils est très puissant mais complexe et délicat à manier (section 6.6). Il faudra donc, plus encore que pour les contraintes statiques, rester raisonnables. Ceci d'autant plus que la frontière entre la validation des contraintes dynamique par le SGBD et la validation par les programmes d'application peut être floue.

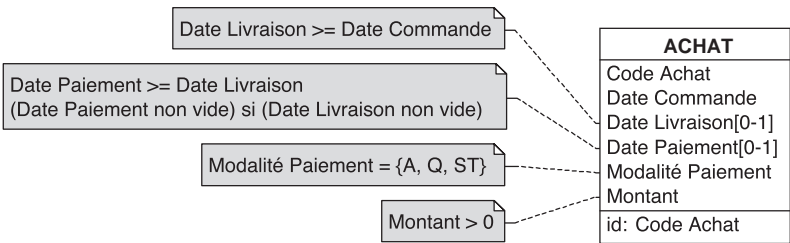


Figure 10.19 - Description de contraintes statiques au moyen d'annotations

c) Intégrité et plausibilité

Il est utile de distinguer une *contrainte d'intégrité* d'une *propriété de plausibilité*. La première est impérative et ne souffre aucune exception : toute tentative de violation doit être rejetée. La seconde décrit une situation à ce point habituelle que tout écart par rapport à celle-ci devrait attirer l'attention. Toute tentative de violation ne doit pas nécessairement être rejetée, mais devrait donner lieu à un avertissement. Par exemple, il est anormal que deux clients distincts ayant mêmes noms, prénoms et dates de naissance soient en plus domiciliés à la même adresse. Si, par extraordinaire, un tel cas devait se produire, alors il serait inopportun de le refuser, mais il serait prudent d'en avertir le responsable des données afin qu'il vérifie. Il serait en tout cas maladroit de traduire cette règle par un identifiant (figure 10.20).

CLIENT
<u>NumCli</u>
Nom
Prénom
Date Naissance
Code Postal
Rue
Numéro
Localité
Compte
id: NumCli
id': Nom
Prénom
Date Naissance
Code Postal
Rue
Numéro
Localité

Figure 10.20 - Schéma erroné : l'identifiant secondaire n'est qu'une simple propriété de plausibilité qui ne peut être déclarée comme un identifiant

EXPEDITION	EXPEDITION
<u>Date</u>	<u>NumExp</u>
<u>NumCli</u>	Date
<u>NumPro</u>	NumCli
<u>NumTransport</u>	NumPro
Poids	NumTransport
Prix	Poids
id: Date	Prix
NumCli	id: NumExp
NumPro	id': Date
NumTransport	NumCli
	NumPro
	NumTransport

Figure 10.21 - L'identifiant (gauche) est sans doute trop complexe et instable pour être primaire. Il faut cependant conserver la contrainte d'unicité (droite)

d) Substitution abusive d'un identifiant

Il peut arriver qu'un identifiant primaire soit jugé trop complexe ou instable, de sorte qu'on est alors tenté de le remplacer par un identifiant technique (figure 10.21, gauche).

De deux choses l'une. Soit l'identifiant complexe constitue une contrainte sans intérêt, et il est légitime de l'abandonner au profit de l'identifiant technique, soit il présente un intérêt pour garantir la qualité des données, et il est impératif de le conserver, quitte à lui donner le statut secondaire (figure 10.21, droite).

e) Relaxation de contraintes trop fortes

La tentation est grande, lorsqu'on fixe les contraintes, de considérer la base de données comme fonctionnant en *régime normal*. C'est oublier qu'elle évolue, et qu'elle passe par des phases où son état est incomplet tout en étant valide. Il importe aussi d'envisager la manière dont l'information sera gérée, et des questions d'ergonomie ou de réglementation propres à l'organisation peuvent imposer des états a priori imprévus des données.

Considérons l'exemple des commandes, dont on sait qu'elles référencent chacune au moins un produit. Toute commande doit donc posséder au moins un détail (figure 10.22, gauche). Ce raisonnement ignore deux situations liées au fonctionnement de l'entreprise. La première est que les bons de commandes sont encodés par un premier service qui enregistre l'entête et vérifie l'existence du client. Plus tard, un second service encode les détails des commandes préenregistrées. La seconde situation, rare mais pas impossible, est celle d'une commande dont tous les produits viennent d'être retirés des stocks. Le concept d'une commande sans détail, a priori absurde, ne doit donc pas être interdit (figure 10.22, droite).

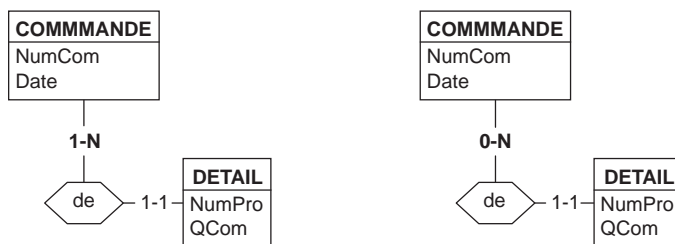


Figure 10.22 - Relaxation d'une contrainte pour tenir compte du comportement organisationnel

10.8 DOCUMENTATION DU SCHÉMA

Chaque objet du schéma (type d'entités, type d'associations, attribut) reçoit une description qui précise la signification exacte et complète du concept qu'il représente. Cette description est généralement exprimée en langage courant. Elle prendra par exemple la forme suivante :

- Si **cli** est un CLIENT, alors **cli** représente *toute personne physique ou morale qui a passé au moins une commande honorée depuis moins de 5 ans ...*
- Si **com** est une COMMANDE, alors **com.DATE** est la date à laquelle **com** a été validée et enregistrée ...
- Si **com** est une COMMANDE et **cli** un CLIENT, et si (**com,cli**) est une association passe, alors **cli** est réputé avoir passé la commande **com** et est responsable du paiement de l'expédition éventuelle...

Cette étape sert souvent de validation du schéma chez les concepteurs débutants : un concept peu clair ou mal défini est impossible à décrire de manière précise.

10.9 COMPLÉTUDE DU SCHÉMA

Certaines parties du schéma ne sont peut-être encore qu'à l'état d'ébauche. On les complétera par des vérifications systématiques telles que les suivantes :

Type d'entités.

- Son nom est-il suffisamment significatif ?
- A-t-il des attributs ? Si ce n'est pas le cas, est-ce normal ?
- A-t-il au moins un identifiant ? L'identifiant primaire est-il constitué de composants obligatoires ?
- Ses identifiants sont-ils minimaux ? Ne comportent-ils pas de rôles de cardinalité 0-1 ou 1-1 ?
- Lui a-t-on affecté une description précise et complète ?

Attribut.

- Son nom est-il suffisamment significatif ?
- Son domaine de valeurs est-il précisé ?
- Est-il monovalué ?
- A-t-on précisé s'il est obligatoire ou facultatif ?
- Lui a-t-on affecté une description précise et complète ?

Type d'associations.

- Son nom est-il suffisamment significatif ?
- Pour chaque rôle
 - son nom (implicite/explicite) est-il unique ?
 - sa cardinalité minimale (0 ou 1) est-elle correcte ?
 - sa cardinalité maximale (1 ou N) est-elle correcte ?
 - la cardinalité est-elle d'un des trois types admissibles : 0-1, 1-1, 0-N ?
- Lui a-t-on affecté une description précise et complète ?

Les lacunes rencontrées seront comblées par une relecture de l'énoncé, par des interviews complémentaires ou par la connaissance du domaine d'application (ou le simple bon sens).

10.10 NORMALISATION DU SCHÉMA

On peut effectuer un nettoyage du schéma obtenu jusqu'à présent soit dans un but de simplification, soit pour éliminer certaines anomalies qui pourraient subsister.

a) Simplification du schéma

Un premier exemple consiste à remplacer par un attribut un type d'entités qui n'aurait lui-même qu'un seul attribut et qui ne serait lié (par un type d'associations *un-à-plusieurs*) qu'à un seul autre type d'entités. Tel serait le cas du schéma de gauche de la figure 10.23 où, dans le schéma final, il apparaît que le concept de localité, jugé utile en son temps, n'a fait l'objet d'aucun enrichissement. On peut estimer inutile de lui conserver le statut de type d'entités, alors qu'il n'apparaît manifestement que comme complément d'information de CLIENT. On pourra alors préférer le schéma de droite. Un tel type d'entités est appelé *type d'entités-attribut*.

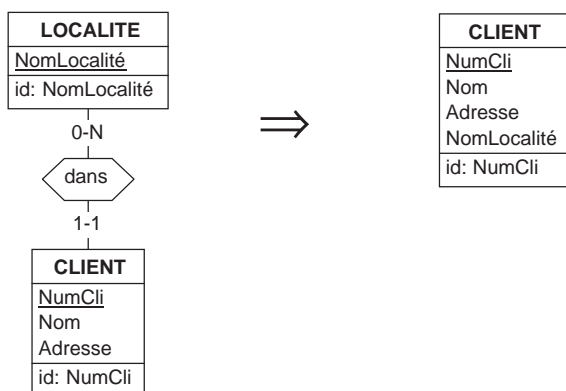


Figure 10.23 - Si le type d'entités LOCALITE n'a pas d'autre raison d'être que d'indiquer la localité de chaque client, alors on peut suggérer de le remplacer par un simple attribut. Remarquons que l'équivalence n'est pas totale, à cause de la cardinalité 0-N à gauche. Cfr. discussion associée à la figure 10.7

De même, un schéma pourra contenir un type d'entités sans attributs, attaché à deux autres types d'entités, et identifié par ceux-ci (ou implicitement par l'un d'entre eux seulement). Ainsi, dans la figure 10.24, le type d'entités EXPORTATION peut-il être remplacé par le type d'associations *exporte*. EXPORTATION est un *type d'entités-association*.

b) Elimination des redondances internes

Le type d'anomalies que nous allons ensuite évoquer constitue une erreur très fréquente chez les débutants. Elle est illustrée par la figure 10.25 (schéma de gauche), qui décrit des employés, caractérisés par leur matricule, leur nom, le nom de leur département et leur localisation. Supposons qu'on apprenne que la localisation d'un employé n'est rien d'autre que l'adresse (unique) de son département.

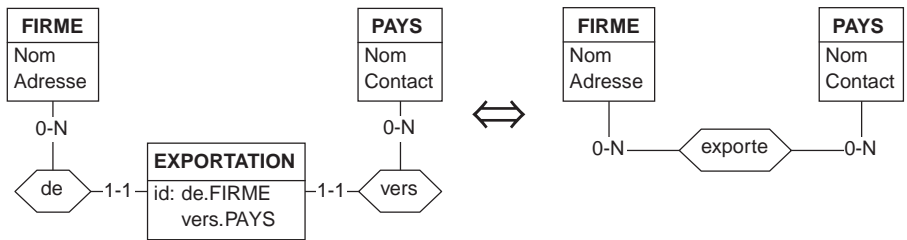


Figure 10.24 - Si le type d'entités EXPORTATION sert uniquement de lien entre les types d'entités FIRME et PAYS, son expression par un type d'associations simplifiera le schéma et le rendra plus expressif

Dans ce cas, tous les employés qui appartiennent au même département ont la même localisation. Il en résulte qu'on enregistrera l'adresse d'un département déterminé autant de fois qu'il possède d'employés, ce qui constitue une redondance d'information, qu'on qualifiera d'interne, car elle est locale à un type d'entités. Il y a à l'origine une erreur qui consistait à affecter Localisation à EMPLOYE, alors que cet attribut dépendait en fait de NomDépart. La situation était alors celle d'un attribut (NomDépart) auquel on tente d'affecter un autre attribut (Localisation), situation qui a été traitée à la figure 10.7.

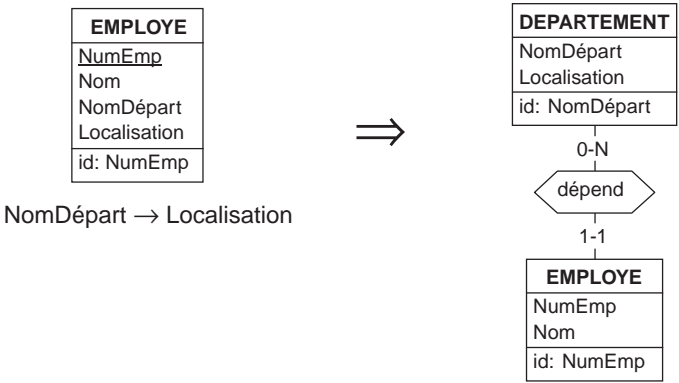


Figure 10.25 - Élimination d'une redondance interne. L'expression "NomDépart → Localisation" indique qu'à une même valeur de NomDépart sera toujours associée la même valeur de Localisation

Le schéma de droite de la figure 10.25 montre comment cette anomalie peut être éliminée par l'extraction des attributs litigieux sous la forme d'un nouveau type d'entités.

Nous proposerons un autre exemple de ce type d'anomalie, basé sur le schéma 9.23 relatif à la gestion d'une bibliothèque. Supposons que l'analyse du domaine d'application conduise au schéma de gauche de la figure 10.26, qui associe aux livres les propriétés ISBN, NumExemplaire, Titre, Auteurs, DateAchat et Localisation. Il apparaît ensuite que tous les exemplaires de même ISBN ont aussi le même titre et les mêmes auteurs. On répétera donc ces informations autant de fois qu'un

numéro ISBN aura d'exemplaires, ce qui constitue une redondance. On transformera ce schéma en celui de droite, où le concept d'ouvrage regroupe les attributs ISBN, Titre et Auteurs.

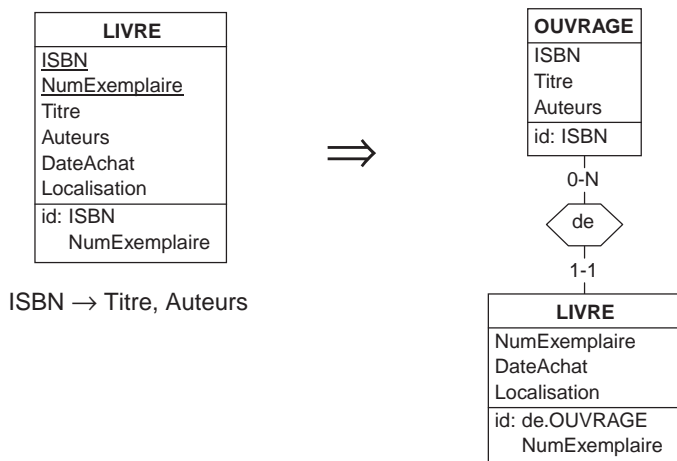


Figure 10.26 - Un autre cas d'élimination d'une redondance interne

Le lecteur attentif aura évidemment fait le lien entre ce type de situation et le principe de normalisation d'une table (section 3.9). Il s'agit en effet du même phénomène, mais qui s'exprime ici dans un autre modèle. Le traitement est lui aussi similaire : on extrait les attributs qui forment le déterminant et le déterminé de la dépendance fonctionnelle anormale sous la forme d'un nouveau type d'entités, qu'on relie au type d'entités d'origine.

10.11 VALIDATION DU SCHÉMA

Il faut ensuite s'assurer que le schéma contient tous les concepts pertinents du domaine d'application, et eux seulement. On le fera valider par les utilisateurs de la future base de données. On vérifiera notamment les identifiants et les caractéristiques des types d'associations (leur classe fonctionnelle et leur caractère obligatoire/facultatif). On vérifiera également l'utilité des concepts retenus et les lacunes éventuelles.

Deux techniques sont actuellement utilisées de manière courante¹ :

- le *prototypage* : on produit rapidement une base de données dans laquelle on range quelques données pertinentes et on confie à l'utilisateur le soin d'expérimenter ce prototype à l'aide de requêtes typiques ;

1. Notamment parce qu'elles sont réalisables par les ateliers de génie logiciel actuels.

- la *réinterprétation*, ou *paraphrase* du schéma conceptuel : on reformule en langage courant chaque concept présent dans le schéma conceptuel et on demande à l'utilisateur de juger de la pertinence de ces propositions.

10.12 EXERCICES

- 10.1 Proposer un schéma conceptuel qui représente le domaine d'application suivant :

Un club vidéo propose des cassettes et des DVD en location à ses membres. Pour chaque membre, on enregistre le nom, l'adresse, le numéro de téléphone. On lui donne un numéro d'inscription qui l'identifie. Chaque support est caractérisé par son type (cassette ou DVD), un code identifiant et la date d'achat. Pour le film enregistré sur le support, on enregistre le titre (identifiant), son réalisateur, l'acteur vedette et le genre. Plusieurs supports peuvent être disponibles pour un même film, alors que pour certains films, il n'existe pas encore de supports proposés à la location. A tout instant, un support peut être loué par un membre du club.

- 10.2 L'énoncé qui suit complète celui du premier exercice :

Chaque support est en outre caractérisé par le nombre d'emprunts, le titre du film, son réalisateur, ses principaux acteurs (nom et prénom, tous deux identifiants, ainsi que la date de naissance), la date de sortie du film. Pour chaque location, on connaît le support, la date d'emprunt, la date normale de restitution, la date de restitution effective et l'emprunteur. Une location dure un nombre entier de jours (au moins un). On conserve l'historique des locations.

Suggestion. On représentera les emprunts en cours et les emprunts clôturés par un même objet.

- 10.3 Proposer un schéma conceptuel qui représente le domaine d'application suivant :

*On désire gérer une bibliographie constituée d'articles (code identifiant, type, titre). Chaque article est écrit par un nombre quelconque d'auteurs. Chaque auteur est caractérisé par son nom (supposé identifiant) et l'organisme dont il dépend. En outre, chaque article est extrait d'un ouvrage dont on donne le titre, l'éditeur, la date de parution, ainsi qu'un numéro identifiant. Dans cet ouvrage, chaque article commence en haut d'une page dont on connaît le numéro. **Exemple** : l'article de code 13245, du type «THEORIE», intitulé «Non-monotonic reasoning in operational research», écrit par Baxter (Stanford Univ.) et Randon (Bell Labs) est extrait (page 340) de l'ouvrage numéro 556473, intitulé «Advanced Topics in Decision Support Systems», publié par North-Holland en 1998.*

Suggestion. La phrase qui précède l'exemple devrait attirer votre attention.

- 10.4 Proposer un schéma conceptuel qui représente le domaine d'application suivant :

Les patients d'un hôpital sont répartis dans les services (caractérisés chacun par un nom identifiant, sa localisation, sa spécialité) de ce dernier. A chaque patient peuvent être prescrits des remèdes. Un remède est identifié par son nom et caractérisé par son type, son fabricant et l'adresse de ce dernier. Chaque prescription d'un remède à un patient est faite par un médecin à une date donnée pour une durée déterminée. On ne peut rédiger plus d'une prescription d'un remède déterminé pour un même patient le même jour. Chaque patient est identifié par un numéro d'inscription. On en connaît le nom, l'adresse et la date de naissance. Chaque médecin appartient à un service. Il est identifié par son nom et son prénom.

Suggestion. On sera particulièrement attentif à la notion de prescription.

- 10.5 Proposer un schéma conceptuel qui représente le domaine d'application suivant :

Une entreprise de distribution dispose d'un certain nombre de véhicules (identifiés par leur numéro et caractérisés par leur capacité et le nom du conducteur). Chaque jour, chaque véhicule effectue une (et une seule) tournée de distribution, d'une longueur déterminée. Durant cette tournée, le véhicule emporte des colis (décrits chacun par un numéro identifiant et un poids). Chaque colis doit être livré à un destinataire. Un destinataire est identifié par un numéro de destinataire et caractérisé par un nom et une adresse.

Suggestion. Attention à l'identifiant des tournées.

- 10.6 Proposer un schéma conceptuel qui représente le domaine d'application suivant :

Une bibliothèque contient des ouvrages, repérés par un numéro unique et caractérisés chacun par son titre, sa maison d'édition et sa date d'édition. Un ouvrage peut avoir été écrit par un ou plusieurs auteurs. Un auteur est identifié par son nom et son prénom; il est caractérisé par sa date de naissance. Un ouvrage peut en outre être caractérisé par un ou plusieurs thèmes. Il existe un répertoire de thèmes standard, chaque thème étant décrit par un code concis (par exemple ROM.HIST) et son libellé en clair (par exemple ROMAN HISTORIQUE). Tous les codes concis sont différents. Lorsqu'on caractérise un ouvrage, on lui attribue le ou les thèmes qui le décrivent le mieux. Il se peut qu'un ouvrage ait été acquis suite à la demande personnelle d'un lecteur. Un lecteur est identifié par son numéro d'inscription et caractérisé par son nom et son adresse.

- 10.7 Proposer un schéma conceptuel qui représente le domaine d'application suivant :

Les clients de l'entreprise sont caractérisés par leur nom, leur adresse et le montant de leur compte. Ils reçoivent un numéro qui permet de les distinguer.

Un client a passé un certain nombre de commandes, caractérisées chacune par un numéro qui les identifie, une date de réception et l'identification de l'employé qui l'a introduite. Chaque commande contient un certain nombre de détails de commande numérotés, qui spécifient chacun le produit commandé, la quantité désirée, la quantité déjà livrée et la date de la dernière livraison. Un produit est identifié par un numéro et caractérisé par un libellé et un prix unitaire. Les clients reçoivent des factures. Chaque facture correspond à un colis expédié à une date déterminée. Un colis comporte, pour un ou plusieurs détails de commande, la quantité expédiée (tout ou partie de la quantité commandée) en fonction de la disponibilité du produit. Une facture comporte un numéro identifiant, la date d'expédition du colis et un montant total. Elle spécifie aussi la composition du colis et les coordonnées du client : nom, adresse et téléphone. L'étiquette du colis reprend le numéro de sa facture, sauf lorsqu'il ne contient que des produits cadeaux. Les sous-colis d'un colis correspondent à des détails de commande distincts.

Suggestion. Attention aux propositions redondantes. Cet énoncé fait penser à un schéma bien connu : prudence tout de même.

- 10.8 Construire un schéma conceptuel correspondant au fonctionnement d'une société de formation.

Une société de formation désire informatiser la gestion des inscriptions aux sessions qu'elle organise, ainsi que la facturation. Il existe un certain nombre de séminaires de formation, chacun consacré à un thème différent et facturé à un tarif déterminé. Un séminaire peut être organisé plus d'une fois, ce qui correspond à autant de sessions. Les sessions d'un séminaire se tiennent à des dates différentes. Des entreprises inscrivent certains de leurs employés à certaines sessions. Il existe un nombre maximum de participants pour les sessions de chaque séminaire (quelle que soit la date de la session). Tous les mois, la société facture à chaque entreprise concernée le montant correspondant à la participation de ses employés aux sessions du mois écoulé.

Suggestion. On ajoutera les attributs que le bon sens suggère pour permettre d'effectuer la facturation.

- 10.9 Construire un schéma conceptuel décrivant la structure d'une institution universitaire. On retiendra les faits suivants.

L'institution est constituée de facultés. A chaque faculté sont rattachés des professeurs et des assistants (qui chacun dépend d'un professeur). Ces personnes sont regroupées en départements au sein de la faculté. Le programme d'enseignement d'une faculté est décomposé en cycles (premier ou Bachelor¹, deuxième ou Master, troisième ou doctorat) eux-mêmes

1. Le choix de la nomenclature anglaise évite toute confusion avec les acceptions anciennes (mais toujours vivaces) des termes *baccalauréat* et *maîtrise*.

comportant des années d'études (2e Bachelor en chimie, 1r Master en sciences politiques et Sociales, etc.) et est constitué de cours, dispensés dans une ou plusieurs années d'études, voire même dans d'autres facultés. Un cours, d'une durée déterminée, est pris en charge par un professeur. Chaque étudiant peut être inscrit à certains cours (on admet l'existence de dispenses, cours supplémentaires, etc.). Si l'étudiant a subi un examen relatif à un cours, on lui attribue la note qu'il a obtenue.

Remarque. Cet énoncé est (in)volontairement ambigu et incomplet. On explicitera les hypothèses choisies pour préciser l'énoncé.

- 10.10 Construire un schéma conceptuel correspondant à une gestion budgétaire élémentaire décrite comme suit.

Une entreprise est constituée de différents départements. Le budget de l'entreprise est décomposé en postes de natures distinctes (petit matériel, déplacements, biens d'équipement, logiciels, frais de personnel, etc.). Chaque département reçoit une part propre de chacun de ces postes. Toute demande de dépense doit suivre la procédure suivante : le département désirant faire l'acquisition de biens ou consommer un service fait une demande d'engagement d'un certain montant, pour une nature de dépense relative au budget qui lui est propre; cette demande est vérifiée puis, si elle est acceptée, fait l'objet d'une commande auprès du fournisseur; à ce moment, le montant engagé est soustrait du poste budgétaire propre à ce département; à la réception du bien ou service, la facture est vérifiée puis payée. On admet qu'une commande puisse être annulée. Le montant engagé est alors à nouveau disponible.

- 10.11 Construire le schéma conceptuel décrivant un cabinet de médecins.

Chaque médecin reçoit des patients en consultation. Lors de chaque consultation de chaque patient, un certain nombre d'actes médicaux sont effectués par le médecin. Chaque acte est répertorié dans une liste standard qui en donne le tarif. C'est ainsi que le médecin calcule le prix de la visite. Le patient peut payer lors de sa visite ou plus tard. Toute consultation non payée au bout d'un mois fait l'objet d'un rappel de paiement. Les sommes perçues durant chaque mois constituent le revenu du médecin. Cette information lui permet de remplir aisément sa déclaration fiscale annuelle.

Le médecin connaît la date de naissance, le sexe et la profession de chaque patient. Il sait également si le patient souffre d'un certain nombre d'affections réparties selon des classes standards. Lors de chaque visite d'un patient, le médecin inscrit la date, les prestations et les médicaments prescrits. Si un patient est décédé, il indiquera la date du décès.

- 10.12 Un *carnet d'adresses* est une petite base de données qui ne semble pas poser de problèmes particuliers. Voire ! Proposer un schéma conceptuel des informations de contact décrites dans l'énoncé ci-dessous.

Pour toute personne, on reprend son nom, son prénom (lorsqu'il est connu), son titre (M., Mme, Maître, etc.), son adresse, son (ses) numéro(s) de téléphone fixe ou mobile et/ou de fax, son adresse électronique. Si la personne occupe une fonction dans une entreprise (ou organisme), on indique aussi le nom et l'adresse de celle-ci. Une personne peut avoir des coordonnées privées et des coordonnées professionnelles. Elle peut aussi occuper plus d'une fonction, dans la même entreprise ou dans des entreprises différentes, et pour chacune de ces fonctions, avoir des coordonnées différentes. Dans certains cas, ces coordonnées sont tout simplement celles de l'entreprise. Il est fréquent qu'une personne change d'adresse, de fonction ou téléphone. Il est important de pouvoir retrouver les caractéristiques d'une personne dans le passé (où travaillait Dupont l'année dernière, et quel est le fax de son ancien employeur ?). On veut aussi savoir depuis quand chaque information a été enregistrée, pour en évaluer la validité.

10.13 Construire le schéma conceptuel décrivant un réseau de distribution d'eau.

Un réseau de distribution d'eau est constitué de sections de canalisation. Une section relie deux nœuds : le nœud amont (alimentant) et le nœud aval (alimenté); l'eau s'écoule donc dans un sens déterminé, de l'amont vers l'aval. Une section est caractérisée par sa longueur, son diamètre, sa date d'installation et la liste des réparations qu'elle a subies (date, distance par rapport au nœud amont, type). A l'exception des nœuds terminaux ("racines" et "feuilles"), un nœud relie une section amont (alimentant le nœud) et une ou plusieurs sections aval (alimentées par le nœud), l'ensemble formant une forêt (ensemble d'arbres).

Un nœud comprend une vanne dont on connaît le modèle. La racine d'un arbre est un nœud spécial qui n'est pas connecté à une section amont, mais qui reçoit l'eau d'une installation d'alimentation, identifiée par un numéro, et d'un type tel que "captage", "réservoir", "station d'épuration", etc. Chaque installation d'alimentation alimente un arbre du réseau. Les feuilles d'un arbre n'alimentent pas de sections aval. Chaque nœud est repéré par son adresse; il est caractérisé par sa profondeur. Une section ne fait pas l'objet de plus d'une réparation par jour.

Chaque client (nom, adresse) possède un et un seul compteur identifié par son numéro, et branché sur une section. La position de ce branchement est indiquée par la distance à partir du nœud amont. Pour chaque client, on enregistre les consommations annuelles.

10.14 Construire le schéma conceptuel décrivant des compétitions sportives.

Chaque année est organisée une saison sportive, sous la présidence d'une personnalité de renom. Une saison est constituée d'une série de championnats, couvrant une certaine période, chacun consacré à une discipline sportive (nom identifiant et responsable). Il y a un seul championnat par saison et par discipline. Durant un championnat sont organisées, à des dates différentes pour un même championnat, des épreuves,

localisées chacune dans une ville. Une ville porte un nom; on y parle une langue principale. Une ville est située dans un pays (nom, code). Les villes d'un pays ont des noms distincts, mais rien n'interdit que deux pays aient des villes de même nom. Chaque pays a une capitale, qui est une ville de ce pays. Des sportifs (matricule, nom, prénom, date de naissance) représentent des pays. Durant une période déterminée, un sportif représente un seul pays. Il peut alors participer à une ou plusieurs épreuves, au terme de chacune desquelles il obtient un résultat.

On vérifiera que le schéma permet de répondre à des questions telles que les suivantes : Qui a obtenu la médaille de bronze en 110 m haies en 2001 ? Quel pays n'a obtenu aucune médaille en 1999 ? Quels sont les sportifs qui ont remporté une médaille d'or dans la capitale du pays qu'ils représentaient ?

- 10.15 Etudier soigneusement les principales pages d'un site de commerce électronique tel que www.amazon.fr, www.fnac.com ou www.ebay.fr. Il apparaît clairement que les informations qu'elles contiennent sont extraites d'une base de données (voir section 7.4.4).

Identifier dans le site choisi une partie réduite, telle que *Musique / Meilleures ventes* chez Amazon, et proposer un schéma conceptuel des concepts et des informations qui y sont représentés.

- 10.16 La partie *achat* du site commercial mentionné à la question précédente comporte aussi des informations relatives aux clients, aux paiements et aux conditions de livraison. Il est évident que la société conserve des informations sur les achats précédents des clients, de manière à personnaliser les pages qui sont présentées à un visiteur particulier.

Proposer une extension du schéma de la question précédente qui décrive ces nouvelles informations.

Remarque. Les exercices 10.15 et 10.16 illustrent une problématique qui prend actuellement de plus en plus d'importance : comprendre la structure sémantique de sites étrangers afin d'en extraire de manière intelligente des données pertinentes. Cet exercice relève de la *rétro-ingénierie des site web*.

- 10.17 Normaliser le schéma de la figure 10.27. Parmi les attributs de **DEPARTEMENT**, on observe les dépendances suivantes :

Entreprise —→ Adresse, Responsable
Responsable —→ Téléphone

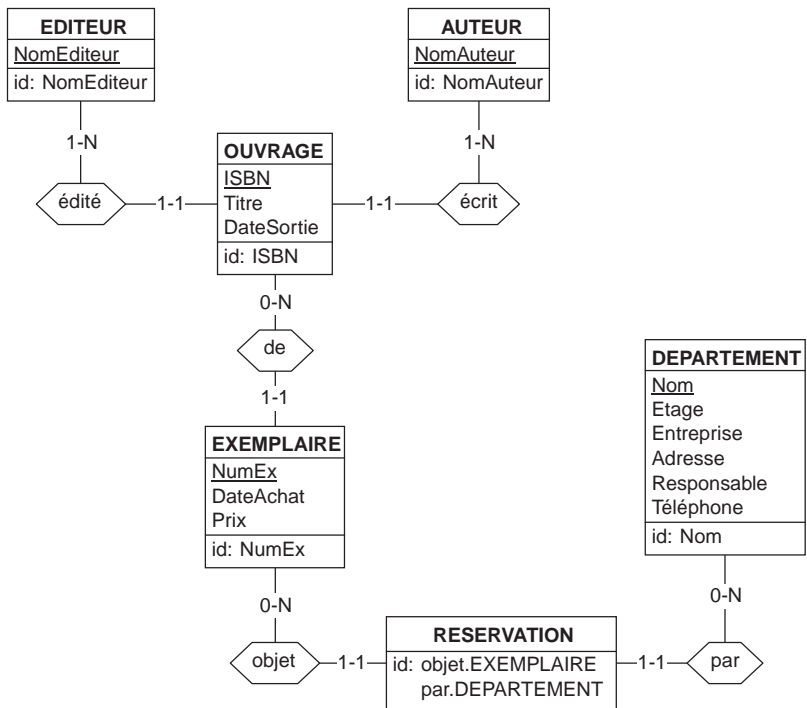


Figure 10.27 - Un schéma à normaliser

10.1 Montrer que les deux schémas de la figure 10.28 sont strictement équivalents. On adoptera d'abord une approche intuitive, basée sur un graphe de populations représentatif (section 9.6). On se servira ensuite des transformations présentées aux figures 10.5 et 10.7.

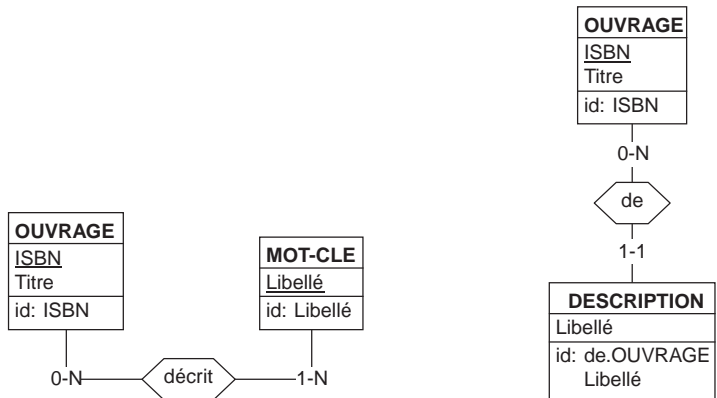


Figure 10.28 - Equivalence de schémas

Chapitre 11

Production du schéma de la base de données

La phase finale de production d'une base de données consiste à traduire le schéma conceptuel dans le langage de définition de données d'un SGBD (ici SQL). Pratiquement, on traduira chaque composant du schéma Entité-association en structures de tables, colonnes, identifiants et clés étrangères. Ces structures seront alors exprimées dans le langage SQL. À cette occasion, on étudiera également le processus de redocumentation d'une base de données existante, aussi appelé rétro-ingénierie.

11.1 INTRODUCTION

Le schéma conceptuel exprime clairement les structures d'information à représenter, mais il n'est pas accepté tel quel par l'ordinateur. Il est donc nécessaire de traduire ce schéma en structures techniques de tables et de colonnes, c'est-à-dire en concepts compréhensibles et gérables par des outils disponibles : les SGBD relationnels.

Nous proposerons des règles de traduction de chaque type de composants Entité-association en structures de bases de données telles qu'elles ont été décrites au chapitre 3. Les structures ainsi obtenues seront alors exprimées aisément en SQL. Nous proposerons une procédure simplifiée qui conviendra pour traiter les problèmes abordés dans cet ouvrage. En particulier, nous ne nous préoccuperons pas de critères de performance, que nous laisserons aux professionnels.

11.2 REPRÉSENTATION DES TYPES D'ENTITÉS

On représente chaque type d'entités par une table à laquelle on donne le nom de ce type d'entités (figure 11.1). Chaque entité de ce type sera décrite par une ligne de cette table.

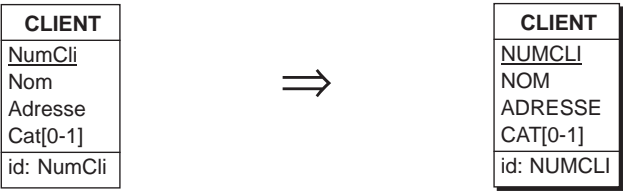


Figure 11.1 - Représentation d'un type d'entités par une table. A chaque attribut correspond une colonne

11.3 REPRÉSENTATION DES ATTRIBUTS

Chaque attribut d'un type d'entités est représenté par une colonne de la table qui représente le type d'entités. On définit le type et la longueur des valeurs de la colonne en fonction du domaine de valeurs de l'attribut. Si le type d'entités n'est pas doté d'attributs, la table n'a pas (encore) de colonnes. La colonne est obligatoire ou facultative selon que l'attribut qu'elle représente est lui-même obligatoire ou facultatif (figure 11.1).

Le nom d'une colonne doit se conformer à la syntaxe imposée par le langage SQL. Nous en dirons quelques mots plus loin dans ce chapitre.

11.4 REPRÉSENTATION DES TYPES D'ASSOCIATIONS

Nous devons distinguer les trois classes fonctionnelles de types d'associations : *un-à-plusieurs*, *un-à-un* et *plusieurs-à-plusieurs*.

11.4.1 Types d'associations un-à-plusieurs

Soit R un type d'associations *un-à-plusieurs* entre A et B (plusieurs entités B pour chaque entité A, une seule entité A pour chaque entité B). L'attribut IA est l'identifiant primaire de A. A est représenté par la table TA et B par la table TB.

- On représente R par une colonne RA de même type que IA et ajoutée à la table TB, de telle sorte que la valeur de RA dans une ligne de B soit la référence d'une ligne de A. Cette colonne RA est déclarée clé étrangère de B vers A.
- Si R est obligatoire pour B, la colonne RA de la table B sera déclarée obligatoire. Si en revanche R est facultatif, alors RA sera déclarée facultative.

Considérons par exemple, le type d'associations occupe entre DEPARTEMENT et EMPLOYE (figure 11.2). Ces deux types d'entités sont représentés par les tables DEPARTEMENT et EMPLOYE. Sachant que DEPARTEMENT est identifié par NomDépart, on représente occupe par une nouvelle colonne NOMDEPART dans la table EMPLOYE de telle sorte que la valeur de NOMDEPART identifie le DEPARTEMENT dans lequel l'EMPLOYE est occupé. NOMDEPART est donc une clé étrangère d'EMPLOYE qui référence la table DEPARTEMENT. On rappelle que, dans un schéma de base de données, l'arc indique la présence d'une **contrainte référentielle** à laquelle est soumise la clé étrangère NOMDEPART.

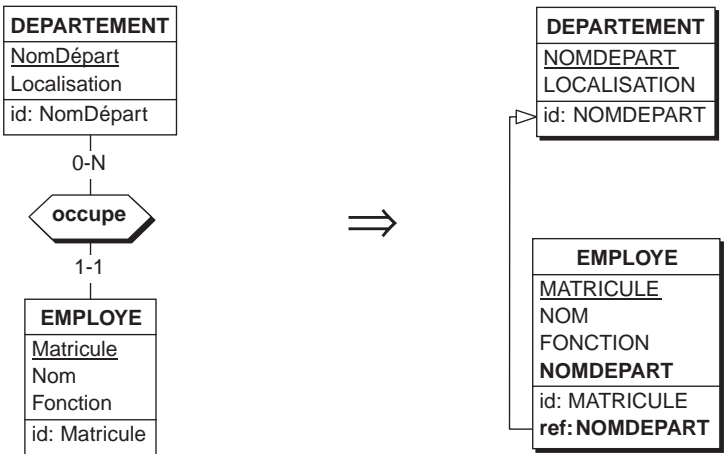


Figure 11.2 - Représentation d'un type d'associations *un-à-plusieurs* par une clé étrangère

Montrons que cette traduction respecte bien le schéma source.

Le type d'associations occupe nous donne deux informations, pas plus, pas moins : (1) à chaque entité EMPLOYE, il associe une et une seule [1-1] entité DEPARTEMENT et (2) à chaque entité DEPARTEMENT il associe un nombre quelconque [0-N] d'entités EMPLOYE.

Que décrit le schéma de droite ? D'une part, il nous dit que chaque ligne EMPLOYE possède une et une seule [1-1] valeur de NOMDEPART (colonne obligatoire) qui elle même désigne une ligne de DEPARTEMENT (clé étrangère). En outre, à chaque ligne de DEPARTEMENT correspond une valeur de NOMDEPART, qu'on va retrouver dans un nombre quelconque [0-N] de lignes EMPLOYE, puisque cette colonne ne constitue pas un identifiant. Les relations entre les lignes DEPARTEMENT et EMPLOYE sont donc l'image exacte des relations entre les entités DEPARTEMENT et EMPLOYE.

Les clés étrangères multi-composants

Lorsque l'identifiant de A est constitué de plusieurs attributs IA1, IA2, ..., on définit autant de nouvelles colonnes RA1, RA2, ... dans la table de B, qui forment ensemble la clé étrangère. Si R est facultatif pour B, alors les colonnes RA1, RA2, ... sont

facultatives. Dans ce cas cependant, il faut imposer une contrainte supplémentaire, selon laquelle tous les composants de la clé étrangère sont simultanément null ou simultanément non null. Il s'agit d'une contrainte de *coexistence* (notée *coex*, et illustrée à la figure 9.26). La figure 11.3 propose un exemple de ces deux situations.

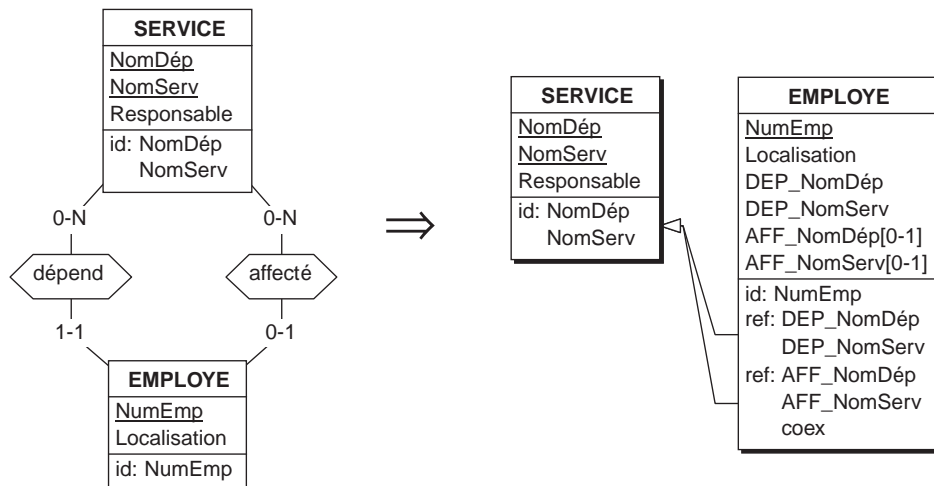


Figure 11.3 - Traduction sous forme d'une clé étrangère multi-composant

Les identifiants hybrides

Une question importante se pose ensuite : qu'en est-il des **identifiants hybrides**, qui comprennent un ou plusieurs types d'entités associés, lorsqu'on représente les types d'associations ? Considérons par exemple le schéma de la figure 9.22, dont un fragment est repris dans la figure 11.4. Dans ce schéma, le type d'entités **DEPARTEMENT** possède un identifiant hybride constitué de **DIRECTION** (via *de*) et **NomDépart**. La représentation du type d'associations *de*, qui entraîne l'ajout d'une colonne **NOMDIR** à la table **DEPARTEMENT**, doit conserver cet identifiant hybride, mais en l'adaptant à la structure de colonnes. Cette adaptation est obtenue par remplacement du composant **DIRECTION** par l'attribut **NOMDIR** de la clé étrangère qui vient d'être ajoutée. Dans notre cas, l'identifiant de la table **DEPARTEMENT** est constitué des colonnes **NOMDIR** et **NOMDEPART**.

Techniquement, le choix des noms des composants de la clé étrangère est indifférent, pourvu qu'ils soient distincts de ceux des autres colonnes de la table. On choisira cependant des noms qui évoquent explicitement le type d'entités référencé ou son rôle dans le type d'associations ou le nom de son identifiant primaire.

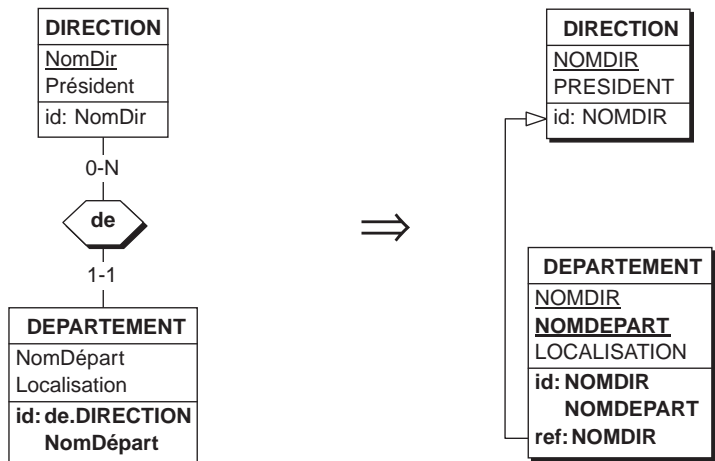


Figure 11.4 - Traduction d'un identifiant hybride : chaque composant *type d'entités* y est remplacé par l'identifiant de celui-ci

11.4.2 Types d'associations un-à-un

La traduction est similaire à celle du cas précédent, mais admet plusieurs variantes. Soit R un type d'associations *un-à-un* entre A et B (une seule entité B pour chaque entité A, une seule entité A pour chaque entité B). A est représenté par la table TA et B par la table TB. L'identifiant primaire de A est IA et/ou celui de B est IB.¹

a) Principe général

On représente R par l'ajout à la table d'un des types d'entités A ou B d'une colonne de même type que l'identifiant de l'autre table, et on déclare cette colonne clé étrangère vers l'autre table. En outre, cette colonne est déclarée identifiant.

Si l'identifiant de l'autre table est constitué de plusieurs attributs, la clé étrangère sera constituée d'autant d'attributs.

b) Plus précisément

- Si R est obligatoire pour A et facultatif pour B, on ajoutera à TA une nouvelle colonne RB, de même type que IB, et qu'on déclare clé étrangère vers TB.
- Si R est obligatoire pour B et facultatif pour A, on ajoutera à TB une nouvelle colonne RA, de même type que IA, et qu'on déclare clé étrangère vers TA.
- Si R est obligatoire pour A et B, le choix d'une des deux situations ci-dessus est indifférent².

1. L'un des types d'entités peut n'avoir qu'un identifiant implicite (voir section 9.4.3).

2. Ce cas peut se présenter, mais a été exclu du modèle présenté à la section 9.3. On ne pourra malheureusement pas traduire facilement le caractère obligatoire du rôle occupé par le type d'entités de la table référencée. Ce point sera repris à la section 11.10.2. Une solution possible serait de fusionner A et B.

- Dans ces trois cas, la nouvelle colonne est déclarée obligatoire.
- Si R est facultatif pour A et B, on procède comme ci-dessus : une clé étrangère est ajoutée indifféremment à TA ou TB. La nouvelle colonne sera déclarée facultative.
- Dans tous les cas, la nouvelle colonne (ou les nouvelles colonnes) constitue en outre un **identifiant** supplémentaire pour sa table.

La figure 11.5 illustre la représentation du type d'associations un-à-un dirige. On observe que la clé étrangère a été associée à la table **DEPARTEMENT** (du côté où dirige est obligatoire), qu'elle constitue un identifiant secondaire pour sa table et qu'on a donné à la nouvelle colonne le nom du rôle que joue l'employé par rapport au département, soit **DIRECTEUR**.

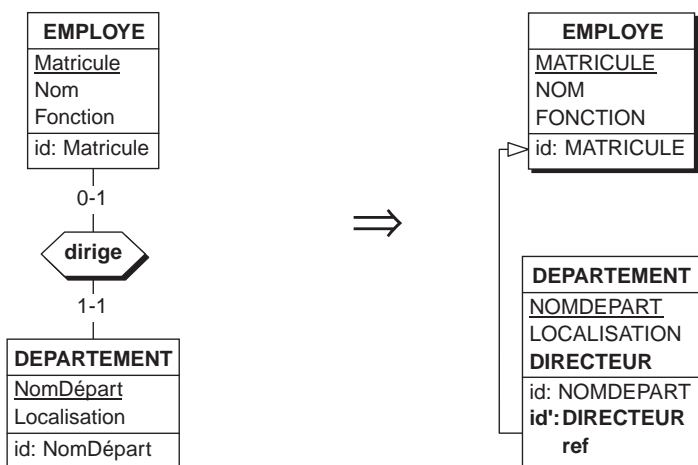


Figure 11.5 - Représentation d'un type d'associations *un-à-un* par une clé étrangère identifiante

Cet exemple nous permet de justifier les règles proposées ci-dessus. Supposons qu'on représente **dirige** par une colonne **DIRECTEUR_DE** ajoutée à la table **EMPLOIE**, déclarée clé étrangère vers **DEPARTEMENT**. Ce choix présente quelques inconvénients non négligeables par rapport à la solution standard :

- d'une part, nous introduisons une colonne facultative, dont on sait qu'elle sera plus délicate à manipuler (section 6.10);
- nous définissons aussi un identifiant facultatif, qui peut poser des problèmes de mise en oeuvre dans certains SGBD (section 6.10);
- enfin, cette clé étrangère doit être soumise à une contrainte très difficile à contrôler : à chaque ligne de **DEPARTEMENT** doit correspondre une ligne d'**EMPLOIE** dont les valeurs de **NOMDEPEART** et **DIRECTEUR_DE** sont égales. SQL est incapable de prendre en charge une telle contrainte, comme nous en discuterons à la section 11.10.2.

11.4.3 Types d'associations plusieurs-à-plusieurs

Soit R un type d'associations *plusieurs-à-plusieurs* entre A et B (plusieurs entités B pour chaque entité A, plusieurs entités A pour chaque entité B). On procède d'abord, comme indiqué à la figure 10.5, à la transformation de R en un type d'entités R' et deux types d'associations *un-à-plusieurs*. On poursuit ensuite comme décrit ci-dessus en représentant R' par une table et les deux types d'associations par des clés étrangères.

La figure 11.6 illustre la représentation du type d'associations *plusieurs-à-plusieurs* fabrique. On notera l'identifiant multicolonne de FABRICATION et les deux clés étrangères. Cette table, dite *associative*, ne contient que des clés étrangères, ce qui est conforme à son simple rôle d'*association* entre les tables USINE et PRODUIT

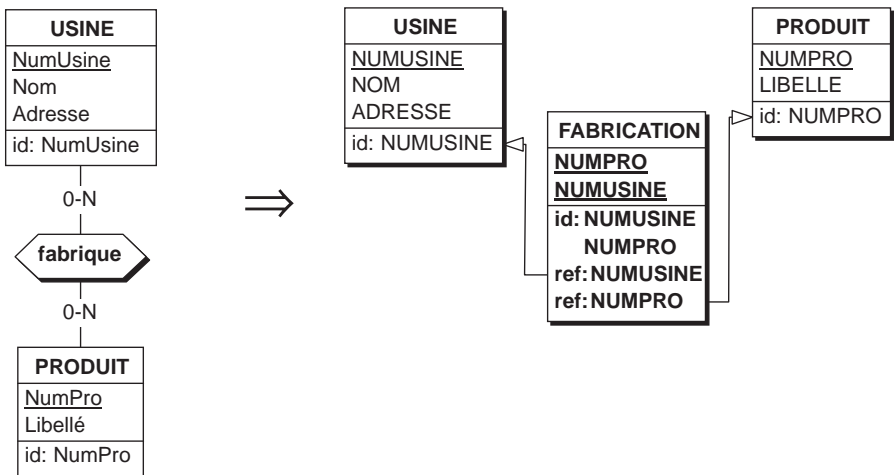


Figure 11.6 - Représentation d'un type d'associations *plusieurs-à-plusieurs* par une table associative munie de deux clés étrangères. Le procédé dérive de celui qui a été décrit à la figure 10.5.

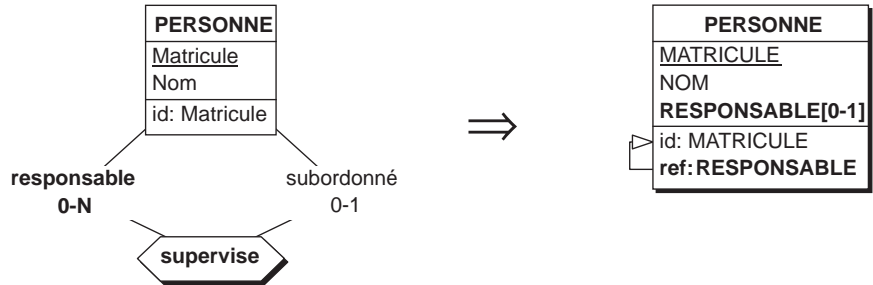


Figure 11.7 - La représentation d'un type d'association *un-à-plusieurs* cyclique obéit aux mêmes règles que celle des autres types d'associations

11.4.4 Types d'associations cycliques

On pourrait penser que les types d'associations cycliques posent des problèmes particuliers. Il n'en est rien. Les règles qui ont été proposées ci-dessus s'appliquent lorsque les deux membres d'un type d'associations sont un seul et même type d'entités. A titre d'exemple, on considérera la traduction du type d'associations *un-à-plusieurs* dirige de la figure 11.7, qui suit les règles énoncées en 11.4.1.

Le cas d'un type d'associations cyclique *un-à-un* s'en déduit aisément. La clé étrangère résultante est *cyclique* et *identifiante*. Si le type d'associations cyclique est *plusieurs-à-plusieurs*, alors les règles 11.4.3 sont d'application (figure 11.8).

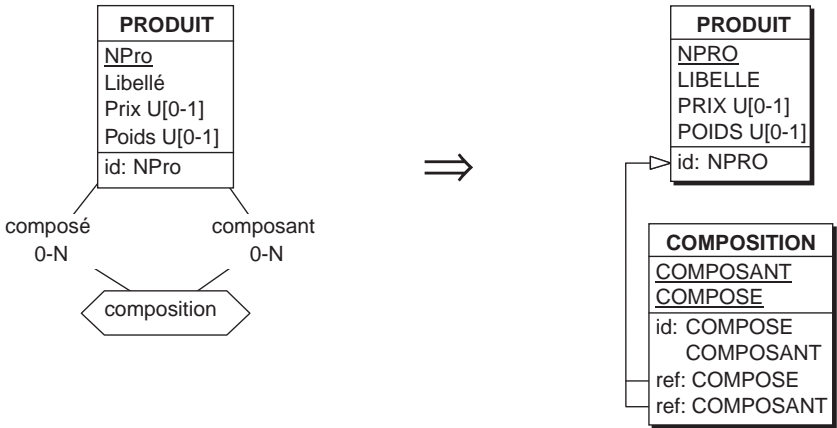


Figure 11.8 - La représentation d'un type d'associations cyclique *plusieurs-à-plusieurs* demande la définition d'une table associative

11.5 REPRÉSENTATION DES IDENTIFIANTS

La traduction des identifiants est immédiate lorsque le type d'entités possède un identifiant constitué exclusivement d'un ou plusieurs attributs. Dans le cas d'un identifiant hybride, comme nous l'avons déjà vu, chacun des composants *type d'entités* est remplacé par la clé étrangère qui représente le type d'associations.

Remarque pratique

On veillera à ne pas confondre deux situations qui pourraient sembler similaires tout en étant fort différentes³ :

- un identifiant constitué de deux (ou plusieurs) composants (figure 11.6 par exemple),
- une table possédant deux (ou plusieurs) identifiants (figure 11.5 par exemple).

Les conventions graphiques distinguent clairement ces deux cas.

3. De même qu'il ne faut pas confondre 1 véhicule à 4 roues avec 4 véhicules à 1 roue.

11.6 TRADUCTION DES NOMS

En principe on désire conserver les noms choisis dans le schéma conceptuel et les attribuer aux tables et colonnes comme suggéré jusqu'ici. Certains SGBD imposent cependant des contraintes sur les noms acceptables. La longueur des noms peut être limitée à 18, 20 ou 32 caractères selon les SGBD. Certains caractères sont interdits dans la composition des noms (le signe "-", les lettres accentuées, apostrophes et guillemets, signes de ponctuation et l'espace par exemple) et certaines combinaisons de caractères sont interdites (un caractère non alphabétique en première position par exemple). Enfin, certains mots sont dits **réservés**; ils ne peuvent être utilisés pour nommer une table ou un attribut. Ainsi en est-il des noms TABLE, COLUMN, DATE, INDEX, parmi d'autres.

Ces contraintes nous amèneront souvent à remplacer certains noms du schéma SQL par d'autres, parfois moins lisibles, mais conformes aux règles imposées par le SGBD. Comme nous l'avons fait dans nos exemples, on suggère que les noms soient transformés de la manière suivante : majuscules (bien que les SGBD SQL confondent majuscules et minuscules dans les noms apparaissant dans un schéma), élimination des accents, signes "-" et espaces remplacés par "_", modification des noms réservés.

Exemple

L'application de ces règles de traduction nous permet de construire le schéma des tables correspondant au schéma conceptuel de la figure 9.16. Les deux schémas sont repris à la figure 11.9. On observera que la table VEHICULE comporte deux colonnes, SIGNATAIRE et NUMCLIENT reférençant chacune une ligne de CLIENT. L'une d'elles n'est-elle pas redondante (auquel cas il faudrait la supprimer) ?

11.7 SYNTHÈSE DES RÈGLES DE TRADUCTION

Les tableaux ci-dessous constituent un récapitulatif des principales règles de traduction d'un schéma conceptuel en structures de tables.

Types d'entités et attributs										
	<table><tr><th>A</th></tr><tr><td>A1</td></tr><tr><td>A2</td></tr><tr><td>A3[0-1]</td></tr></table>	A	A1	A2	A3[0-1]	<table><tr><th>A</th></tr><tr><td>A1</td></tr><tr><td>A2</td></tr><tr><td>A3[0-1]</td></tr></table>	A	A1	A2	A3[0-1]
A										
A1										
A2										
A3[0-1]										
A										
A1										
A2										
A3[0-1]										

Types d'associations													
1 à N	<table><tr><th>A</th></tr><tr><td><u>IA</u></td></tr><tr><td>id: IA</td></tr></table> — 0-N — <table><tr><th>R</th></tr></table> — 1-1 — <table><tr><th>B</th></tr></table>	A	<u>IA</u>	id: IA	R	B	<table><tr><th>TA</th></tr><tr><td><u>IA</u></td></tr><tr><td>id: IA</td></tr></table> ← <table><tr><th>TB</th></tr><tr><td>RA</td></tr><tr><td>ref: RA</td></tr></table>	TA	<u>IA</u>	id: IA	TB	RA	ref: RA
A													
<u>IA</u>													
id: IA													
R													
B													
TA													
<u>IA</u>													
id: IA													
TB													
RA													
ref: RA													

1 à 1	<div><div><div>A</div><div><u>IA</u></div><div>id: IA</div></div><div>0-1</div><div>R</div><div>1-1</div><div>B</div></div>	<div><div><div>TA</div><div><u>IA</u></div><div>id: IA</div></div><div></div><div><div>TB</div><div>RA</div><div>id': RA</div><div>ref</div></div></div>
N à N	<div><div><div>A</div><div><u>IA</u></div><div>id: IA</div></div><div>0-N</div><div>R</div><div>0-N</div><div><div>B</div><div><u>IB</u></div><div>id: IB</div></div></div>	<div><div><div>A</div><div><u>IA</u></div><div>id: IA</div></div><div></div><div><div>TR</div><div><u>RB</u></div><div>RA</div><div>id: RB</div><div>RA</div><div>ref: RA</div><div>ref: RB</div></div><div></div><div><div>B</div><div><u>IB</u></div><div>id: IB</div></div></div>
Id multi-composant	<div><div><div>A</div><div><u>IA1</u></div><div><u>IA2</u></div><div>id: IA1</div><div>IA2</div></div><div>0-N</div><div>R</div><div>1-1</div><div>B</div></div>	<div><div><div>TA</div><div><u>IA1</u></div><div><u>IA2</u></div><div>id: IA1</div><div>IA2</div></div><div></div><div><div>TB</div><div>RA1</div><div>RA2</div><div>ref: RA1</div><div>RA2</div></div></div>

Identifiants

attributs	<div><div>A</div><div>A1</div><div>A2</div><div>id: A1</div></div>	<div><div>TC</div><div>C1</div><div>C2</div><div>id: C1</div></div>
mixte	<div><div><div>A</div><div><u>A1</u></div><div>id: A1</div></div><div>0-N</div><div>R</div><div>1-1</div><div><div>C</div><div>C1</div><div>id: R.A</div><div>C1</div></div></div>	<div><div><div>TA</div><div><u>A1</u></div><div>id: A1</div></div><div></div><div><div>TC</div><div><u>RA</u></div><div><u>C1</u></div><div>id: RA</div><div>C1</div><div>ref: RA</div></div></div>
rôles	<div><div><div>A</div><div><u>A1</u></div><div>id: A1</div></div><div>0-N</div><div>RA</div><div>1-1</div><div><div>C</div><div>C1</div><div>id: RB.B</div><div>RA.A</div></div><div>1-1</div><div>RB</div><div>0-N</div><div><div>B</div><div><u>B1</u></div><div>id: B1</div></div></div>	<div><div><div>A</div><div><u>A1</u></div><div>id: A1</div></div><div></div><div><div>C</div><div><u>RB</u></div><div><u>RA</u></div><div>C1</div><div>id: RB</div><div>RA</div><div>ref: RA</div><div>ref: RB</div></div><div></div><div><div>B</div><div><u>B1</u></div><div>id: B1</div></div></div>

11.8 LES STRUCTURES PHYSIQUES

Nous avons brièvement décrit dans le chapitre 3 des constructions physiques destinées à définir l’implantation des données dans une machine et en particulier à contrôler les performances (c’est-à-dire le temps d’exécution) des requêtes. Afin d’illustrer la définition de ces structures, nous ajouterons au schéma de tables de la figure 11.9 des index et des espaces de stockage. Dans le cadre limité de ce chapitre,

il est inutile de discuter de manière approfondie des questions de performances et d'optimisation.

Nous donnerons cependant deux règles de bon sens qui régissent la définition des index.

- 1. On définira un index pour chaque *identifiant*.
- 2. On définira un index pour chaque *clé étrangère*.

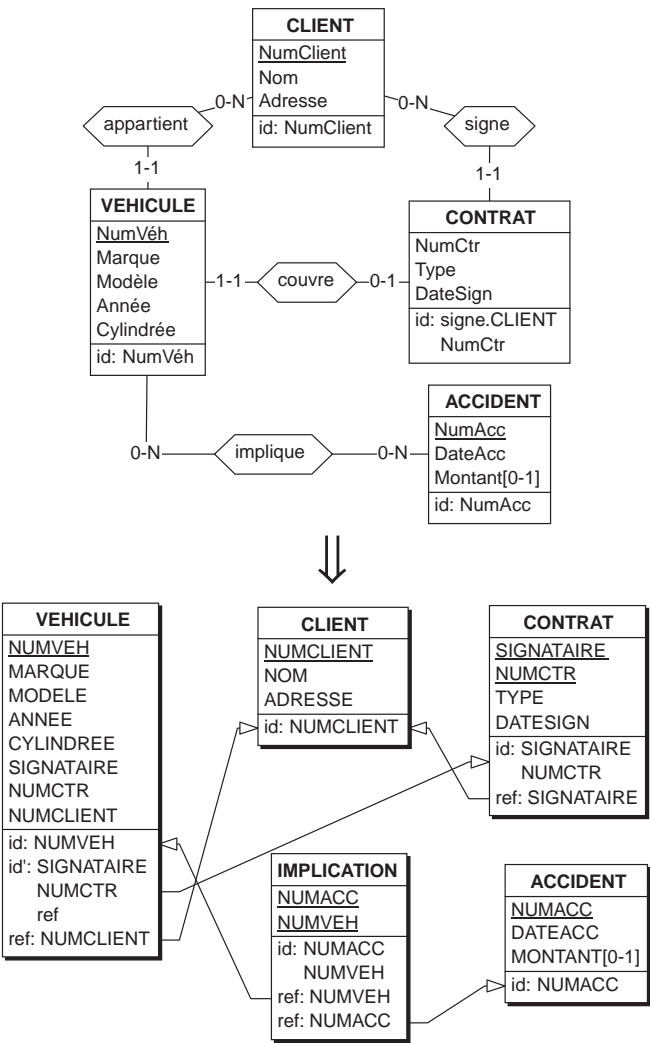


Figure 11.9 - Traduction d'un schéma conceptuel en structure de tables

Justification. Les identifiants et les clés étrangères sont souvent utilisés comme critères de sélection dans les requêtes `select`, `update` et `delete`. La présence d'index sur ces constructions permettra au SGBD d'effectuer des accès rapides aux

données. En outre, lors de l'insertion d'une ligne dans une table, le SGBD va d'abord vérifier que la valeur de chaque identifiant n'y est pas déjà présente. En l'absence d'un index sur les identifiants, cette vérification nécessiterait une lecture séquentielle de toutes les lignes de la table, dont le temps d'exécution serait prohibitif.

Le schéma de la figure 11.10 a été obtenu à partir du schéma précédent (11.9, bas), auquel les index suggérés ci-dessus ont été ajoutés, ainsi que deux espaces de stockage que nous mentionnerons sans justification. Le lecteur attentif remarquera que certaines clés étrangères ne font pas l'objet d'un index. Sous certaines conditions, dont la discussion dépasserait l'objectif de cet ouvrage, un index I2 dont les colonnes apparaissent en première position dans un autre index I1 peut être supprimé. Le SGBD est en effet capable d'utiliser l'index I1 lorsque la requête exigerait d'utiliser l'index I2.

Rappelons encore que la connaissance des index, des espaces de stockage et autres structures physiques est inutile, voire nuisible, pour rédiger des requêtes en SQL. C'est le rôle de l'optimiseur, composant essentiel du SGBD, que de traduire les requêtes en accès aux données sur le disque de manière à exploiter au mieux ces constructions.

11.9 TRADUCTION DES STRUCTURES EN SQL

La traduction de structures de tables en SQL est immédiate. Elle obéit aux règles énoncées dans la section 4.2, qui ne seront pas reprises ici, mais que nous illustrerons par un exemple concret, celui de l'expression du schéma de tables de la figure 11.10. Seuls deux index ont été traduits, à titre d'illustration.

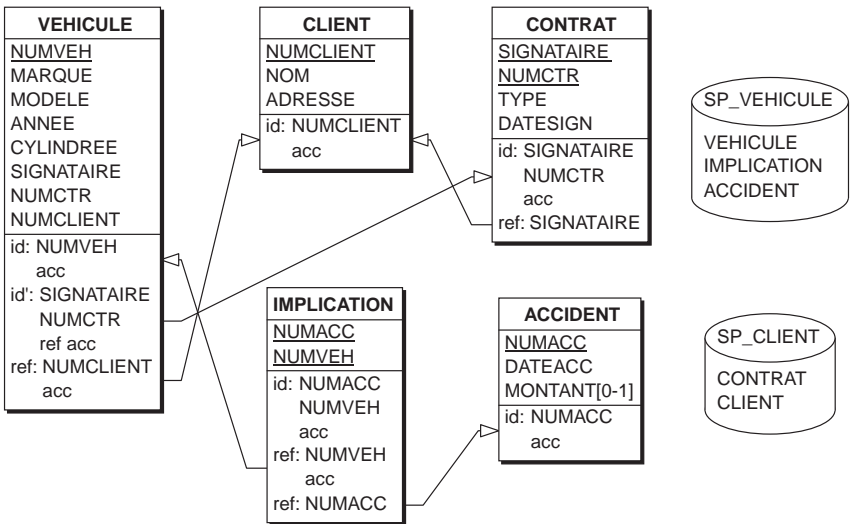


Figure 11.10 - Le schéma relationnel a été enrichi par l'adjonction d'index et d'espaces de stockage


```

create dbspace SP_VEHICULE;

create dbspace SP_CLIENT;

create table CLIENT ( NUMCLIENT char (12) not null,
                      NOM char (38) not null,
                      ADRESSE char (60) not null,
                      primary key (NUMCLIENT) )
in SP_CLIENT;

create table CONTRAT ( SIGNATAIRE char (12) not null,
                       NUMCTR char (8) not null,
                       TYPE char (12) not null,
                       DATESIGN date not null,
                       primary key (SIGNATAIRE,NUMCTR),
                       foreign key (SIGNATAIRE) references CLIENT )
in SP_CLIENT;

create table VEHICULE (NUMVEH char (16) not null,
                       MARQUE char (30) not null,
                       MODELE char (30) not null,
                       ANNEE decimal (4) not null,
                       CYLINDREE decimal (6) not null,
                       SIGNATAIRE char (12) not null,
                       NUMCTR decimal (8) not null,
                       NUMCLIENT char (12) not null,
                       primary key (NUMVEH),
                       unique (SIGNATAIRE,NUMCTR),
                       foreign key (NUMCLIENT) references CLIENT,
                       foreign key (SIGNATAIRE,NUMCTR) references CONTRAT)
in SP_VEHICULE;

create table ACCIDENT (NUMACC char (10) not null,
                       DATEACC date not null,
                       MONTANT decimal (6),
                       primary key (NUMACC))
in SP_VEHICULE;

create table IMPLICATION (NUMVEH char (16) not null,
                           NUMACC char (10) not null,
                           primary key (NUMVEH,NUMACC),
                           foreign key (NUMVEH) references VEHICULE,
                           foreign key (NUMACC) references ACCIDENT )
in SP_VEHICULE;

create unique index XCLI_NUMCLI
on CLIENT(NUMCLI);
...
create index XIMPL_NUMV
on IMPLICATION(NUMV);

```

11.10 COMPLÉMENTS

11.10.1 Les contraintes d'intégrité additionnelles

On sait que SQL ne prend en charge qu'un nombre restreint de contraintes d'intégrité, à savoir l'unicité (via les *primary keys* et le prédicat *unique*), l'intégrité référentielle (*foreign key*) et les colonnes obligatoires (*not null*). Les autres contraintes se coderont à l'aide de prédicats (section 6.4), de déclencheurs (section 6.6) ou de

procédures SQL (section 6.5). Nous illustrerons ce processus par la traduction de quelques contraintes reprises de la section 9.5.

a) Contraintes statiques

Les deux contraintes

- il existe quatre valeurs de catégories de clients : B1, B2, C1, C2;
- un client ne peut être de catégorie C2 que si son compte est non négatif;

s'expriment aisément par deux prédicats (on observe que la seconde contrainte est une *implication*, qui se traite comme décrit en 5.2.6) :

```
create table CLIENT( ...
    primary key(NUMCLI),
    check(CAT is null or CAT in ('B1','B2','C1','C2')),
    check(CAT<>'C2' or CAT is null4 or COMPTE >= 0));
```

Cependant, la contrainte

- toute commande doit avoir au moins un détail;

est plus complexe à traduire, car elle ne peut être prise en charge ni par des prédicats ni par des déclencheurs, mais par des procédures SQL. Elle correspond à la situation décrite ci-dessous (cardinalité 1-N, section 11.10.2). En outre, la procédure SUP_DETAIL de la section 6.5 gère l'opération de suppression d'un détail.

b) Contraintes dynamiques

Nous considérerons la contrainte :

- on ne peut ajouter de détails qui référencent un produit en rupture de stock.

Elle doit être validée lors de deux événements distincts : l'insertion d'une ligne de DETAIL et la modification de NPRO d'une ligne de DETAIL. Le mécanisme de traduction qui s'impose est un déclencheur :

```
create trigger INS_DETAIL_STK_0
before insert or update of NPRO on DETAIL
for each row
begin
    if (select QSTOCK from PRODUIT where NPRO=new.NPRO) <= 0
        then abort();
end;
```

Ce dispositif annule (abort) toute opération d'insertion ou de modification de NPRO qui entraînerait une violation de la contrainte.

4. Attention, en logique ternaire, not (CAT = 'C2') n'est pas (CAT <> 'C2') mais (CAT <> 'C2' or CAT is null).

11.10.2Au sujet des rôles de cardinalité 1-N

Nous avons explicitement interdit d’assigner à un rôle la cardinalité 1-N au motif que sa traduction dans un schéma de tables posait des problèmes qui dépassaient les objectifs de cet ouvrage. Il est certain en revanche qu’une telle cardinalité est parfois utile (le schéma de la figure 10.7 par exemple exigerait que le rôle de FOURNISSEUR soit obligatoire). Considérons le schéma de la figure 11.11 (haut), dans lequel on indique, via la cardinalité 1-N, le fait que *tout accident implique au moins un véhicule*.

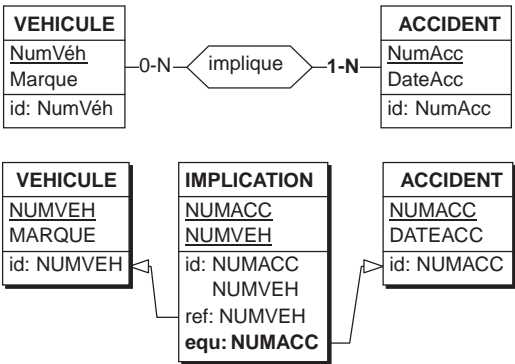


Figure 11.11 - Une cardinalité 1-N se traduit par une clé étrangère exprimant l’égalité de deux ensembles de valeurs plutôt que leur inclusion

Le type d’associations *implique* se traduit par la table IMPLICATION (figure 11.11, bas), incluant deux clés étrangères dont l’une, qui référence ACCIDENT, est un peu particulière. En effet, non seulement toute valeur de IMPLICATION.NUMACC⁵ doit se trouver parmi les valeurs de ACCIDENT.NUMACC, mais en outre, toute valeur de ACCIDENT.NUMACC doit aussi apparaître dans IMPLICATION.NUMACC. Dit plus clairement, les ensembles de valeurs ACCIDENT.NUMACC et IMPLICATION.NUMACC doivent être égaux, ce qu’indique le schéma via le symbole *equ*, en remplacement du symbole *ref*.

Une clé étrangère du type *equ* pose deux problèmes, qui expliquent pourquoi nous avons écarté les cardinalités 1-N. Le premier concerne la dynamique d’enregistrement d’un accident. Lors de l’insertion de la première ligne d’IMPLICATION, le SGBD vérifiera l’existence préalable de la ligne d’ACCIDENT cible, mais lors de l’insertion de la ligne d’ACCIDENT, il vérifiera aussi qu’il existe au moins une ligne d’IMPLICATION de même valeur de NUMACC, ce qui bien sûr est impossible. On pourrait donc conclure à l’impossibilité intrinsèque d’insérer des lignes d’ACCIDENT. Les tables ACCIDENT et IMPLICATION devraient donc rester irrémédiablement vides !

Le deuxième problème est qu’il est impossible de traduire complètement ce type de clé étrangère par un prédicat (check) ou par un déclencheur, notamment en ce qui

5. C’est-à-dire la valeur de la colonne NUMACC d’une ligne de la table ACCIDENT.

concerne l'enregistrement des données. En revanche, il est possible de construire une procédure SQL (section 6.5) qui reçoit en argument les données décrivant l'accident et les références des véhicules impliqués, puis crée la ligne d'ACCIDENT et une ou plusieurs lignes d'IMPLICATION à partir de ces données. Si celles-ci ne sont pas correctes, la procédure n'effectue aucune opération. Il faut encore obliger l'utilisateur à utiliser cette procédure lorsqu'il désire enregistrer un accident, et lui interdire d'agir directement sur les tables concernées (et de risquer d'enregistrer un accident sans véhicules), ce qui est du ressort du contrôle d'accès (section 6.1). Dans les programmes d'application, le programmeur définira une **transaction**, qui est une suite d'instructions qui doit être exécutée complètement ou pas du tout, quoi qu'il arrive. Au sein d'une transaction, des situations d'incohérence de données peuvent se produire (p. ex. on insère une ligne d'IMPLICATION sans ligne d'ACCIDENT cible, ou l'inverse), pour autant que tout rentre dans l'ordre lorsque la transaction se termine.

11.11 RÉTRO-INGÉNIERIE D'UNE BASE DE DONNÉES

Les matériaux étudiés dans ce chapitre nous permettent d'aborder un problème d'importance croissante : la redocumentation, ou plus généralement, la *rétro-ingénierie*, d'une base de données existante. Il s'agit en résumé de reconstruire le schéma de tables et le schéma conceptuel d'une base de données dont on a perdu toute documentation. En caricaturant, on pourrait dire qu'on inverse les techniques proposées dans ce chapitre, ce qui est en grande partie correct.

Les objectifs d'un tel processus sont variés : *maintenance* (correction des programmes utilisant la base de données), *évolution* (prise en compte de nouveaux concepts), *intégration* (d'applications ou de bases de données), *portage* dans un autre environnement (MS Access vers SQL Server par exemple), *évaluation* de la qualité d'une application (que peut valoir un programme si sa base de données a été mal conçue ?) ou tout simplement *remise en ordre* des composants d'une application (de la part du propriétaire des données avant son départ).

L'idée de base est qu'on ne peut utiliser une base de données que si une documentation correcte et complète est disponible, à défaut de quoi il est nécessaire de reconstruire celle-ci. A cela s'ajoute l'observation qu'on ne peut comprendre un programme existant (en vue de sa maintenance par exemple) que si on dispose d'une documentation de la base de données sur laquelle il travaille.

En pratique, la reconstruction des schémas s'appuie sur des sources d'information qui sont principalement le code DDL (ou le contenu des tables du catalogue), le code des programmes d'application, les écrans des programmes, les pages web générées (voir exercice 10.15) et les données elles-mêmes.

a) Première approche

Dans le cas des bases de données que nous avons rencontrées dans cet ouvrage, et qui résultent de l'application systématique de règles de production rigoureuses, la reconstruction des deux schémas ne pose guère de problèmes. Il suffit en effet

d'appliquer les règles inverses de celles que nous avons proposées dans ce chapitre. Ainsi, la figure 11.12 illustre la reconstruction du schéma de tables à partir du code SQL. Chaque clause SQL traduit un objet du schéma : table, colonne, domaine de valeurs, identifiant, clé étrangère. Il est donc aisé de redessiner le schéma qui est à l'origine de ce code.

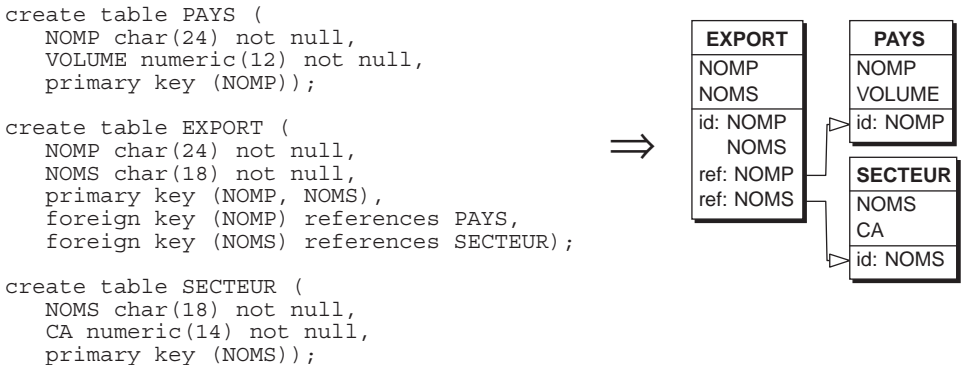


Figure 11.12 - Reconstruction du schéma de tables à partir du code SQL DDL

Ce schéma dérive lui-même d'un schéma conceptuel qu'on cherche à retrouver. La figure 11.13 illustre le processus inverse : chaque élément du schéma de tables est issu d'un objet conceptuel qu'il s'agit de *deviner* :

- à toute *table* correspond un type d'entités,
- à chaque *colonne* (non clé étrangère) correspond un attribut,
- aux colonnes de chaque *clé étrangère* correspond un type d'associations *un-à-plusieurs* ou *un-à-un*,
- à chaque *identifiant* de table correspond à un identifiant de type d'entités; un identifiant qui inclut une clé étrangère est l'image d'un identifiant hybride.

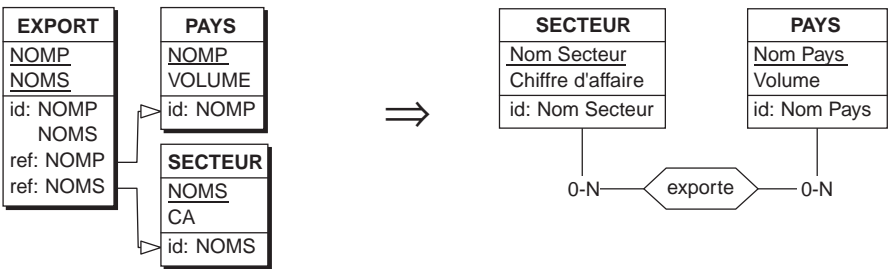


Figure 11.13 - Dérivation du schéma conceptuel à partir du schéma de tables

Le schéma résultant doit encore être *normalisé* au sens de la section 10.10. En particulier, un *type d'entités/association* (voir figure 10.24) doit être converti en type

d'associations. On retravaillera également les noms de manière à les rendre plus représentatifs des concepts qu'ils désignent.

Sur le terrain cependant, la rétro-ingénierie est loin de se présenter de manière aussi simpliste. Des contraintes techniques que nous avons ignorées jusqu'ici, ainsi que certaines pratiques de développement et de programmation viennent compliquer le processus. Nous allons évoquer brièvement ces problèmes ainsi que les solutions qui peuvent y être apportées.

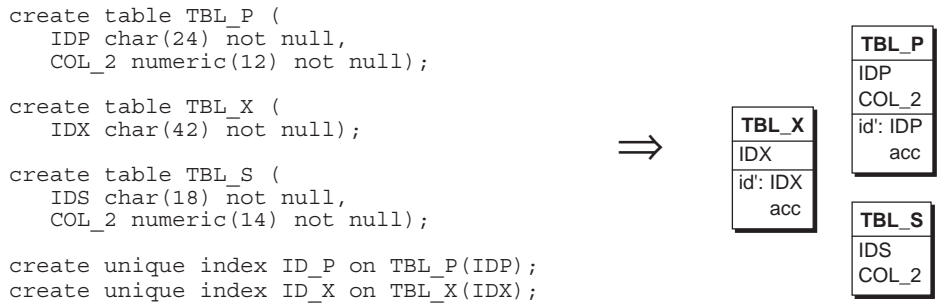


Figure 11.14 - La simple analyse du code DDL ne suffit pas toujours à retrouver le véritable schéma de tables

b) Le problème de l'extraction des structures de données

Souvent, le code DDL n'exprime qu'une partie seulement des objets du schéma de tables, et parfois d'une manière obscure. Le texte DDL de la figure 11.14 décrit une base de données de même contenu que celle du schéma 11.12. Cependant, le schéma de tables est à la fois plus pauvre et malaisé à interpréter, notamment en raison de conventions de dénomination peu expressives ("TBL_P" = "table PAYS", "IDX" = "identifiant de la table EXPORT", etc.). On observe que les identifiants n'ont pas été déclarés en tant que tels, mais via des index (unique). En outre, l'un d'eux, celui de TBL_S n'a pas été déclaré. Il en est de même des clés étrangères, dont le texte ne contient aucune trace. Enfin, les deux colonnes de EXPORT ont été fusionnées en une seule.

Il est probable que cette base de données est relativement ancienne, datant d'une époque où les concepts d'identifiants primaires et de clés étrangères n'étaient pas pris en charge par les SGBD. Il se peut aussi qu'elle résulte de la conversion d'anciens fichiers, soumis à des règles de dénomination drastiques (par exemple, noms des champs de 8 caractères maximum) et imposant qu'un index soit défini sur un seul champ (contrainte des fichiers COBOL). Il semble aussi que le programmeur ait cherché à optimiser le schéma, notamment en ne déclarant pas l'identifiant de TBL_S (= SECTEUR) qui comporte sans doute trop peu de lignes pour mériter un index, et donc un identifiant.

Retrouver le véritable schéma de la base de données, c'est-à-dire celui de la figure 11.12 (droite), est l'objectif du processus nommé *extraction des structures de données*. Le premier schéma issu de l'analyse du code DDL est progressivement enrichi des découvertes qu'on fera grâce à l'analyse des informations complé-

mentaires : examen du code des programmes d'application, analyse des données elles-mêmes (voir exercice 11.12), analyse des états imprimés et des écrans et pages web présentant les données, etc. Dans le cas de notre dernier exemple, le *code des programmes* nous apprendra que la colonne IDX est découpée en deux fragments, dont le premier est utilisé comme clé d'accès à la table TBL_P, suggérant ainsi un rôle de clé étrangère :

```
select IDX into :P from TBL_X where substring(IDX from 25 for 18)=:S  
select COL_2 from TBL_P where IDP=substring(:P from 1 for 24);
```

L'*examen des écrans* de saisie et d'interrogation des données nous en apprendra plus sur la signification de chaque table et chaque colonne, et nous suggérera des noms plus explicites. Enfin, une *analyse des données* de la table TBL_S vérifiera qu'il n'y existe pas de lignes se partageant la même valeur de IDS. La requête ci-dessous devrait renvoyer un résultat vide :

```
select IDS,count(*) from TBL_S group by IDS having count(*) > 1;
```

c) *Le problème de la conceptualisation des structures de données*

Si l'interprétation du schéma enrichi s'est avéré très simple dans le cas de la figure 11.13, il n'en va malheureusement pas de même dans la réalité. D'une part, les modèles utilisés sont plus riches que ceux que nous avons utilisés dans cet ouvrage (relire la section 9.9), et d'autre part, les développeurs appliquent des règles de production de structures de données beaucoup plus variées et complexes que celles que nous avons suggérées dans ce chapitre. Pour compliquer les choses, les bases de données réelles ont le plus souvent été fortement restructurées pour des raisons d'optimisation, et comportent des tables non normalisées (pour limiter la fragmentation des données résultant de la décomposition des structures, comme décrit dans les sections 3.9.2 et 10.10), des redondances (colonnes et tables dont le contenu est calculé à partir d'autres données de la base) et autres astuces techniques parfois douteuses qui peuvent compliquer la compréhension des structures de données et la reconstruction du schéma conceptuel. Enfin, le schéma d'une grande base de données comporte souvent des structures erronées et des zones mortes, laissées en place par manque de temps de la part des développeurs successifs.

A titre d'exemple, une propriété multiple (attribut multivalué) sera tout autant représentée par les techniques de la figure 10.11, qu'on déconseille, que par celles de la figure 10.12. Le processus de reconstruction du schéma conceptuel, dit de *conceptualisation*, repose sur une connaissance approfondie des différentes techniques de traduction de schémas conceptuels effectivement utilisées sur le terrain.

On conseillera au lecteur qui désire approfondir la question la référence [Hainaut, 2002] en guise de point de départ.

11.12 EXTENSIONS DE LA MÉTHODE

Les méthodes utilisées par les professionnels du développement d'applications informatiques, et particulièrement de bases de données, sont plus complexes que celle qui vient d'être décrite, mais n'en sont que des extensions naturelles si on tient compte des conditions dans lesquelles elles sont pratiquées.

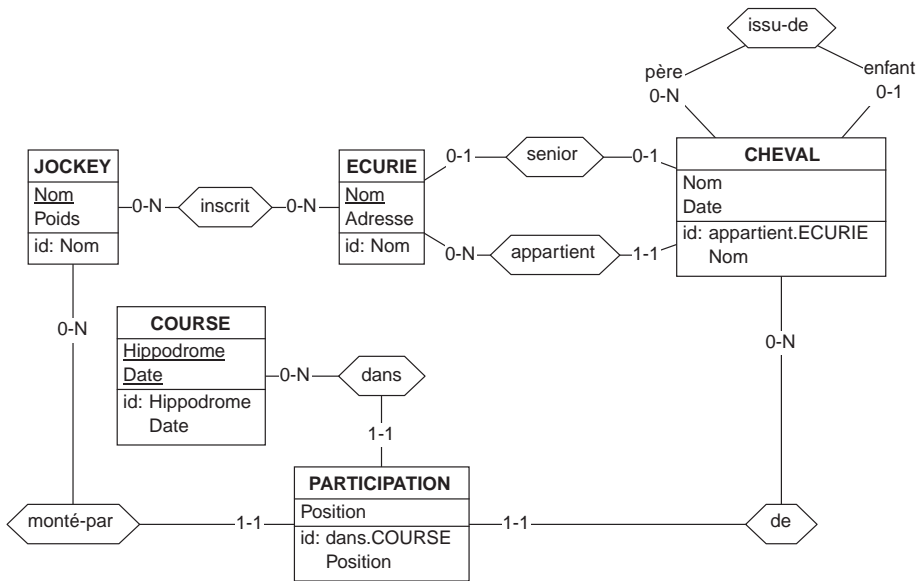
- Le modèle utilisé pour formuler les schémas conceptuels est plus riche, comme nous l'avons vu dans la section 9.9. La méthode doit donc tenir compte de la traduction des concepts additionnels.
- Les sources d'information sont plus variées et souvent moins structurées et cohérentes que celle qui est décrite dans cet ouvrage (chapitre 9) : interviews, observation des procédures de travail, analyse des informations existantes (formulaires, écrans, rapports, documents divers), applications informatiques existantes, etc.
- Plusieurs personnes peuvent procéder à l'analyse de parties différentes du domaine d'application. La fusion des schémas conceptuels partiels ainsi obtenus est un processus délicat qui réclame des techniques appropriées.
- La production d'une base de données réelle doit généralement respecter des contraintes que nous avons ignorées et dont la prise en compte peut être particulièrement complexe : répartition des données sur plusieurs machines, minimisation de l'espace occupé sur disque, optimisation des temps de réponse, optimisation des procédures d'exploitation, réutilisation de données existantes ou interconnexion avec des bases de données existantes par exemple.
- Le code SQL de grands schémas représente plusieurs centaines, voire plusieurs milliers de pages. Il comprend, outre les définitions illustrées dans ce chapitre, la définition des prédicats, des déclencheurs et des procédures SQL et les instructions de gestion des privilèges. Il est hors de question de rédiger manuellement un texte aussi complexe.

On trouvera notamment dans [Hainaut, 1986], [Batini, 1992], [Ceri, 1997], [Nancy, 1996], [Blaha, 1998], [Elmasri, 2000], [Hainaut, 2002b] l'exposé de méthodes plus complètes.

Confronté à des problèmes de cette nature, l'informaticien utilisera des outils spécialisés, les AGL ou ateliers de génie logiciel. Ces logiciels gèrent la saisie, la mémorisation et la modification des schémas conceptuels, vérifient leur cohérence, en permettent la normalisation, produisent des documents divers, dérivent des schémas de bases de données complets et génèrent automatiquement le code SQL (ou autre). Ils sont en outre capables de redocumenter des bases de données anciennes (*rétro-ingénierie*) et de les convertir selon d'autres technologies. Le lecteur intéressé trouvera sur le site Web de l'ouvrage la version pédagogique de l'outil DB-MAIN, un AGL supportant les principaux processus qui viennent d'être évoqués, et avec lequel tous les schémas relationnels et Entité-association de cet ouvrage ont été construits.

11.13 EXERCICES

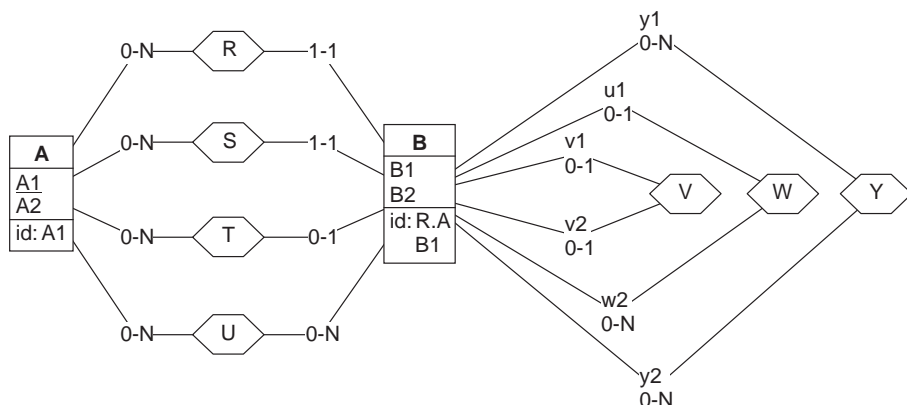
- 11.1 Construire le schéma des tables correspondant au schéma conceptuel d'une structure administrative de la figure 9.22.
- 11.2 Construire le schéma des tables correspondant au schéma conceptuel d'une gestion de bibliothèque de la figure 9.23.
- 11.3 Construire le schéma des tables correspondant au schéma conceptuel représentant des voyages en train de la figure 9.24.
- 11.4 Traduire en un schéma de tables le schéma conceptuel de l'exercice 10.3. Garnir les tables avec les données qui correspondent à l'exemple décrit.
- 11.5 Considérons le schéma étendu de la figure 9.25. Nous ne disposons évidemment pas des règles de traduction en structure de tables des concepts avancés utilisés dans ce schéma. Cependant, en réfléchissant à la signification des structures de ce schéma, il est possible de le réécrire dans le modèle simplifié présenté au chapitre 9. Traduire ensuite cette expression simplifiée en structures de tables.
- 11.6 Dériver un schéma de tables à partir du schéma conceptuel suivant.



On notera que le schéma ne dit rien sur les liens éventuels entre les types d'associations *senior* et *appartient*. Il conviendrait que la traduction demandée ne fasse pas non plus d'hypothèse sur ces liens.

11.7 Traduire le schéma conceptuel correspondant à l'exercice 10.8 en structure de tables. Ecrire le déclencheur SQL qui vérifie la contrainte sur le nombre maximum de participants à chaque session en recherchant les anomalies. Ecrire une requête SQL qui rassemble les données à inclure dans les factures du mois.

11.8 Traduire le schéma conceptuel ci-dessous en structure de tables.



11.9 Traduire en structure de tables le schéma conceptuel de l'exercice 10.11. Ecrire les requêtes qui aideront le médecin dans ses tâches d'analyse de données. Ses questions les plus fréquentes sont les suivantes :

- le nombre de cas dans chacune des classes d'affections;
- la corrélation entre la profession et chacune de ces classes d'affections;
- la corrélation entre la localité de résidence et chacune de ces classes d'affections;
- la corrélation entre l'âge et chacune de ces classes d'affections;
- la corrélation entre la localité de résidence et l'âge des patients;
- la répartition des âges de décès selon la classe d'affection;
- la répartition des âges de décès selon la profession.

11.10 Rédiger une procédure SQL qui enregistre dans la base de données de schéma 11.9, modifié selon la figure 11.11, les informations concernant un accident. Pour simplifier on admet qu'un accident n'implique pas plus de trois véhicules.

11.11 **Rétro-ingénierie.** On vient de retrouver le texte SQL-DDL ci-dessous, et on s'interroge sur sa signification. Dessiner le schéma de tables que ce texte exprime. Ensuite, construisez un schéma conceptuel dont ce premier schéma pourrait être la traduction exacte.

```
create table CERTIFICAT (NOM char(36) not null,
                        INTITULE char(36) not null,
```

```

        DATE_COURS date not null,
        NUMERO numeric(10) not null,
        NOTE numeric(3,1) not null,
primary key (NOM, INTITULE, DATE_COURS, NUMERO),
foreign key (NUMERO) references ETUDIANT,
foreign key (NOM,INTITULE,DATE_COURS) references COURS_DONNE);

create table COURS (NOM char(36) not null,
        INTITULE char(36) not null,
        DUREE numeric(3) not null,
primary key (NOM, INTITULE),
foreign key (NOM) references ECOLE,
foreign key (INTITULE) references MATIERE);

create table COURS_DONNE (NOM char(36) not null,
        INTITULE char(36) not null,
        DATE_COURS date not null,
primary key (NOM,INTITULE, DATE_COURS),
foreign key (NOM,INTITULE) references COURS);

create table ECOLE (NOM char(36) not null,
        MATRICULE char(12) not null,
        ADRESSE char(60) not null,
primary key (NOM),
unique (MATRICULE),
foreign key (MATRICULE) references PROFESSEUR);

create table ETUDIANT (NUMERO numeric(10) not null,
        NOM char(36) not null,
primary key (NUMERO));

create table MATIERE (INTITULE char(36) not null,
        TYPE char(6) not null,
primary key (INTITULE));

create table PAR (NOM char(36) not null,
        INTITULE char(36) not null,
        DATE_COURS date not null,
        MATRICULE char(12) not null,
primary key (MATRICULE, NOM, INTITULE, DATE_COURS),
foreign key (MATRICULE) references PROFESSEUR,
foreign key (NOM,INTITULE,DATE_COURS) references COURS_DONNE);

create table PREREQUIS (EST_INTITULE char(36) not null,
        INTITULE char(36) not null,
primary key (EST_INTITULE, INTITULE),
foreign key (INTITULE) references MATIERE,
foreign key (EST_INTITULE) references MATIERE);

create table PROFESSEUR (MATRICULE char(12) not null,
        NOM char(32) not null,
primary key (MATRICULE));

```

- 11.12 Rétro-ingénierie.** Proposer un schéma de tables complet et un schéma conceptuel pour les quatres tables du catalogues décrites aux figures 6.1 et 6.2. Pour compléter le schéma de tables, on recherchera les identifiants et les clés étrangères à partir des noms des colonnes, puis on validera par l'examen des données.

Chapitre 12

Bases de données : études de cas

Ce chapitre présente deux domaines d'application pour lesquels on construit un schéma conceptuel ainsi qu'une structure de base de données.

12.1 INTRODUCTION

En guise d'illustration, nous proposons au lecteur une visite au zoo, suivie d'un voyage en avion. Pour chacune de ces escapades, on proposera un énoncé en langage courant, la construction du schéma conceptuel résultant de l'analyse de cet énoncé et sa traduction en structures de base de données.

Ces deux cas présentent la particularité d'offrir différents niveaux de description. Chacun d'eux est constitué de faits qui peuvent aisément être représentés dans une base de données, mais également d'autres faits qui suggèrent une représentation par des formules de calcul et qui seront plus adéquatement traités dans la deuxième partie de l'ouvrage. L'intégration de ces deux représentations a fait l'objet d'une étude que le lecteur trouvera dans [Hainaut, 1994] ainsi que sur le site Web de l'ouvrage.

12.2 LES ANIMAUX DU ZOO

Le domaine d'application est inspiré du problème décrit dans [Geoffrion, 1987]. Dans cet article, l'auteur se limite à l'expression d'un seul régime pour un seul animal et réduit le problème aux concepts d'ingrédient, de nutriment, de teneur, de composition et de besoin. On y a donc ajouté les concepts de régime, d'animal, de soins et d'espèce.

12.2.1 Énoncé

Les animaux d'un zoo suivent chacun un régime alimentaire. Un régime est constitué d'un mélange d'ingrédients, chacun en quantité déterminée. Le régime d'un animal peut varier d'un jour à l'autre. Chaque animal est caractérisé, en fonction de son espèce, par ses besoins minima et maxima en nutriments (calcium, protéines, etc.), exprimés en mg par unité de poids de l'animal. Ces besoins sont fonction de l'espèce de l'animal. On connaît la teneur de chaque ingrédient en nutriments, exprimée en mg par kg d'ingrédient. Chaque ingrédient a un coût unitaire. Chaque animal requiert des soins qui sont évalués en francs par jour. Ces soins peuvent varier d'un jour à l'autre.

12.2.2 Construction du schéma conceptuel

L'énoncé est décomposé, et à l'occasion reformulé, en propositions élémentaires. On évalue la pertinence de chaque proposition, son originalité (le fait qu'elle exprime n'est-il pas déjà connu?), sa non-contradiction avec le schéma déjà élaboré. On enrichit ensuite ce schéma des concepts nouveaux qu'exprime cette proposition. On fait l'hypothèse que l'énoncé a déjà été lu dans son intégralité, ce qui nous permet de faire référence à certains faits non encore analysés.

1. un zoo a des animaux

Le zoo est le domaine d'application, qu'on ne représente pas explicitement pour l'instant. On définit en revanche un type d'entités ANIMAL.

2. un animal suit un régime

Un type d'entités REGIME est défini et relié à ANIMAL par le type d'associations suit. Ce type d'associations peut être *un-à-plusieurs* ou *un-à-un*. On choisit la dernière solution, car il semble plus réaliste de considérer qu'un régime est propre à un animal, même si deux régimes peuvent avoir la même composition.

3. un régime est constitué d'ingrédients

On définit un type d'entités INGREDIENT, relié à REGIME par un type d'associations composé de. Ce type d'associations est *plusieurs-à-plusieurs*, un ingrédient pouvant entrer dans la composition de plusieurs régimes.

4. un ingrédient entre dans la composition d'un régime en une quantité déterminée

Cette composition est caractérisée par une quantité. Elle ne peut donc être représentée sous la forme d'un type d'associations. Ce dernier est remplacé par un type d'entités COMPOSITION, qui recueille l'attribut Quantité. COMPOSITION est identifié par les types d'entités REGIME et INGREDIENT.

5. le régime que reçoit un animal dépend du jour

Un animal peut donc avoir plusieurs régimes. Le type d'associations suit est modifié pour devenir *un-à-plusieurs*. Un attribut Date régime est assigné au type d'entités REGIME. Dépendant de l'animal et de la date, le régime est identifié par ces deux éléments. On définit donc l'identifiant de REGIME.

6. un animal est d'une espèce

On affecte à ANIMAL un attribut Espèce.

7. une espèce a des besoins en nutriments

Le concept de nutriment est représenté par un type d'entités NUTRIMENT. Ce dernier devrait être relié à Espèce par un type d'associations, ce qui n'est possible que si on transforme cet attribut en type d'entités ESPECE, relié à ANIMAL par le type d'associations *un-à-plusieurs* origine. Le type d'association qui relie ESPECE et NUTRIMENT est du type *plusieurs-à-plusieurs*; on lui donne le nom a besoin de.

8. calcium, protéines sont des exemples de nutriments

Ces exemples suggèrent un attribut Nom nutriment identifiant de NUTRIMENT.

9. chaque besoin d'un animal en un ingrédient est caractérisé par une quantité minimale ...

Les besoins d'un animal sont ceux de son espèce. Il n'est donc pas nécessaire d'établir un type d'associations spécifique entre ANIMAL et NUTRIMENT. Le concept de besoin est caractérisé par une quantité minimale, qui se traduit idéalement par un attribut. Il faut donc d'abord transformer le type d'associations a besoin de en un type d'entités, de nom BESOINS. On affecte à ce dernier un attribut Min ...

- 10.... et une quantité maximale

... et un attribut Max. La valeur de ce dernier n'est pas inférieure à celle de Min pour toute entité BESOINS.

- 11.un animal a un poids

ANIMAL reçoit un attribut Poids.

- 12.ces besoins dépendent de l'espèce de l'animal

Cette propriété est déjà exprimée par le fait que chaque entité BESOINS est attachée à une entité ESPECE.

13.un ingrédient contient des nutriments

Un type d'associations contient est défini entre INGREDIENT et NUTRIMENT. Un nutriment pouvant intervenir dans plusieurs ingrédients, ce type d'associations est *plusieurs-à-plusieurs*.

14.... chacun en une teneur déterminée

Une teneur (c'est-à-dire une quantité contenue) caractérise un nutriment en tant que composant d'un ingrédient. Un attribut Quantité est défini, mais ne peut être affecté à NUTRIMENT (il y a autant de teneurs qu'il y a d'ingrédients où un nutriment apparaît), pas plus qu'à INGREDIENT (il y a autant de teneurs qu'il y a de nutriments qui composent l'ingrédient). Il doit être affecté au lien qui unit ces derniers, le type d'associations contient. Ce dernier doit donc avant tout être transformé en un type d'entités, qu'on nommera TENEUR.

15.un ingrédient a un coût unitaire

On affecte un attribut Coût unitaire à INGREDIENT.

16.un animal requiert des soins

Chaque prestation de soins à un animal est représentée par une entité SOINS. Un type d'associations objet de attache chaque entité SOINS à l'entité ANIMAL représentant l'animal qui en bénéficie. Ce type d'associations est *un-à-plusieurs*.

17.les soins d'un animal ont un coût

Le type d'entités SOINS reçoit un attribut Coût.

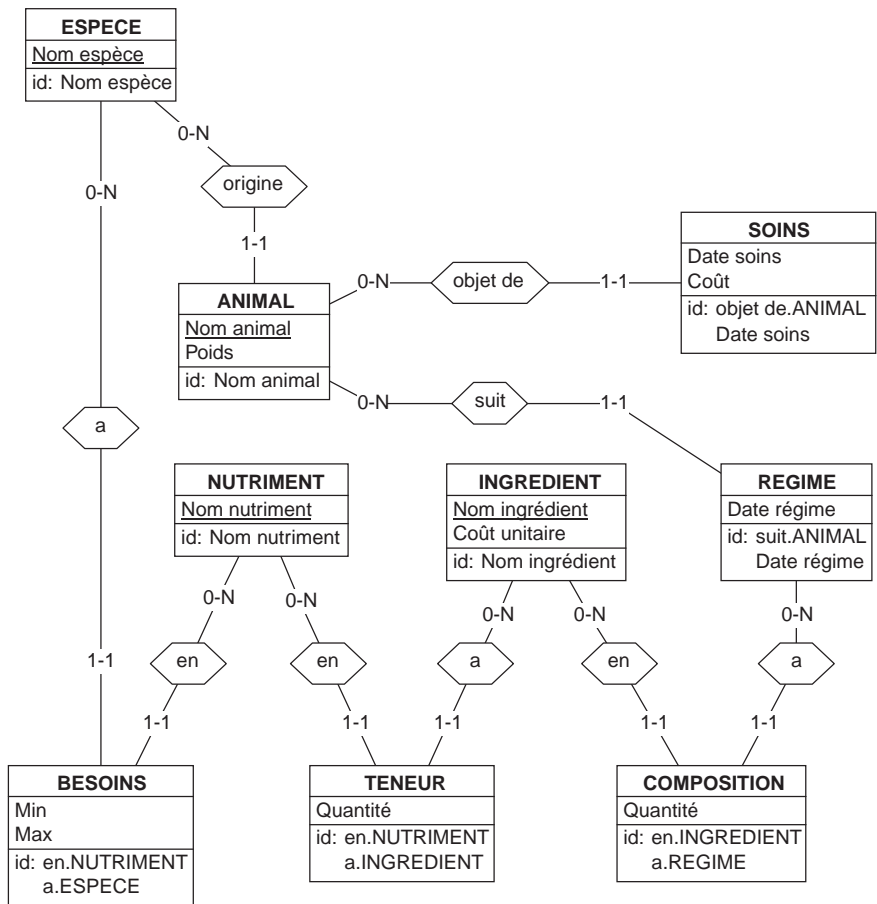
18.les soins d'un animal dépendent du jour

Le type d'entités SOINS reçoit un attribut Date soins. Étant donné l'unité de coût (francs par jour), une entité SOINS représente l'ensemble des prestations destinées à un animal, un jour déterminé. On traduit ce fait par l'identifiant de SOINS.

L'évaluation de la complétude du schéma montre qu'ANIMAL n'a pas d'identifiant. On ajoute un attribut Nom animal dont on fait l'identifiant de ce type d'entités. Le schéma de la figure 10.1 synthétise les résultats de cette analyse.

12.2.3 Production du schéma de tables

La traduction de ce schéma sous forme de structures de bases de données relationnelle ne pose pas de problèmes particuliers (schéma 10.2). On veillera à rendre les noms de tables et de colonnes conformes aux règles syntaxiques du langage SQL. En particulier, les espaces ont été remplacés par le symbole "_". Pour simplifier l'exercice, on a laissé la définition des index à l'initiative du lecteur.



Contrainte : Pour toute entité BESOINS b, b.Max >= b.Min

Figure 12.1 - Schéma conceptuel du domaine d'application ZOO

12.2.4 Production du code SQL

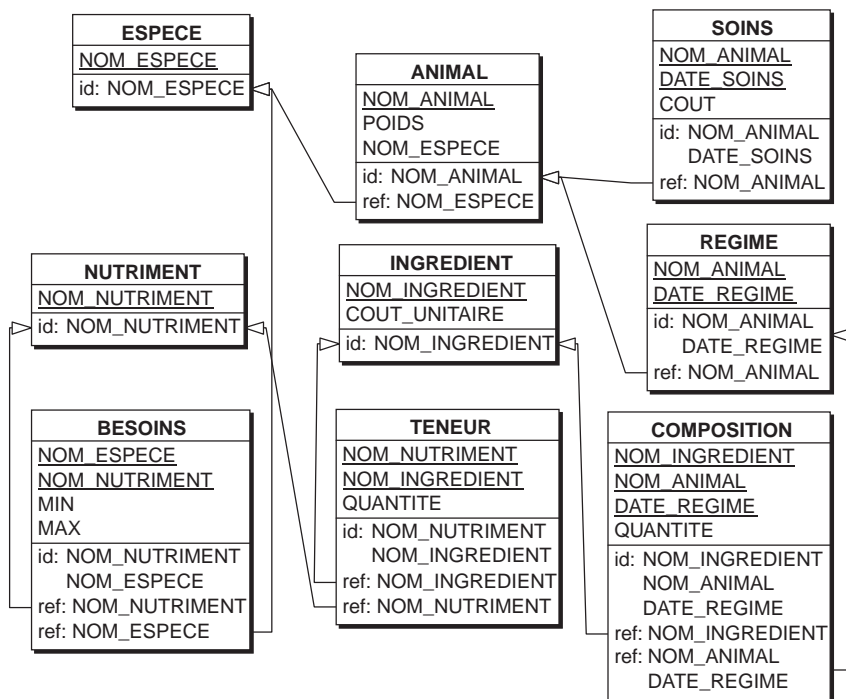
L'expression de ce schéma en structures de tables s'effectue également sans difficulté.

```

create schema ZOO;

create table ESPECE (
    NOM_ESPECE char(32) not null,
    primary key (NOM_ESPECE));

create table ANIMAL (
    NOM_ANIMAL char(32) not null,
    POIDS numeric(8,4) not null,
    NOM_ESPECE char(32) not null,
    primary key (NOM_ANIMAL),
    foreign key (NOM_ESPECE) references ESPECE);
    
```



Contrainte : Pour toute ligne b de BESOINS, $b.Max \geq b.Min$

Figure 12.2 - Le schéma des tables correspondant au schéma conceptuel 10.1

```

create table INGREDIENT (
    NOM_INGREDIENT char(32) not null,
    COUT_UNITAIRE numeric(8,2) not null,
    primary key (NOM_INGREDIENT));

create table NUTRIMENT (
    NOM_NUTRIMENT char(32) not null,
    primary key (NOM_NUTRIMENT));

create table REGIME (
    NOM_ANIMAL char(32) not null,
    DATE_REGIME numeric(6) not null,
    primary key (NOM_ANIMAL, DATE_REGIME),
    foreign key (NOM_ANIMAL) references ANIMAL);

create table SOINS (
    NOM_ANIMAL char(32) not null,
    DATE_SOINS numeric(6) not null,
    COUT numeric(6,2) not null,
    primary key (NOM_ANIMAL, DATE_SOINS),
    foreign key (NOM_ANIMAL) references ANIMAL);

create table BESOINS (
    NOM_ESPECE char(32) not null,
    NOM_NUTRIMENT char(32) not null,
    MIN numeric(8,4) not null,
    MAX numeric(8,4) not null,

```

```

primary key (NOM_NUTRIMENT, NOM_ESPECE),
foreign key (NOM_ESPECE) references ESPECE,
foreign key (NOM_NUTRIMENT) references NUTRIMENT),
check (MAX >= MIN));

create table COMPOSITION (
  NOM_INGREDIENT char(32) not null,
  NOM_ANIMAL char(32) not null,
  DATE_REGIME numeric(6) not null,
  QUANTITE numeric(8,2) not null,
  primary key (NOM_INGREDIENT, NOM_ANIMAL, DATE_REGIME),
  foreign key (NOM_INGREDIENT) references INGREDIENT,
  foreign key (NOM_ANIMAL, DATE_REGIME) references REGIME);

create table TENEUR (
  NOM_NUTRIMENT char(32) not null,
  NOM_INGREDIENT char(32) not null,
  QUANTITE numeric(8,4) not null,
  primary key (NOM_NUTRIMENT, NOM_INGREDIENT),
  foreign key (NOM_NUTRIMENT) references NUTRIMENT,
  foreign key (NOM_INGREDIENT) references INGREDIENT);

```

12.3 VOYAGES AÉRIENS

Ce cas est une extension d'un exemple de modèle utilisé lors d'un séminaire de formation à la bureautique et à l'aide à la décision organisé pour le personnel d'UTA dans les années 80 et repris dans [Hainaut, 90]. Il est évident que les problèmes qui se posent dans le domaine visé par ce cas sont d'une tout autre complexité. Tout comme le précédent, mais plus explicitement encore, cet énoncé suggère un modèle de calcul des coûts, modèle qui sera développé dans la deuxième partie de l'ouvrage.

12.3.1 Énoncé

Le domaine d'application concerne des vols organisés par une compagnie aérienne, et dont on veut déterminer la structure du coût. On considère donc qu'un vol relie deux aéroports en passant par un certain nombre d'escales, qui sont également, du moins quand les choses se passent normalement, des aéroports. Un aéroport porte un nom, mais sera le plus souvent désigné par un code standard propre aux compagnies aériennes. Un même vol peut être effectué à des dates différentes par des appareils différents. Un appareil, désigné par son modèle, est caractérisé par la capacité de ses réservoirs (en kilos de carburant) ainsi que la consommation à vide, équipage compris (en kilos de carburant par km). On connaît aussi sa charge utile maximale et sa consommation supplémentaire par kilo de charge (en kilos de carburant par kilo de charge et par km). Il est à noter cependant que la consommation à vide n'inclut pas le transport du carburant lui-même. On admet que la charge utile (fret et passagers) est constante pour toute la durée du vol, mais qu'elle peut varier d'une date à l'autre. On connaît la longueur de chaque tronçon du vol, c'est-à-dire la distance entre deux aéroports, ou escales, consécutifs de ce vol. On supposera que la consommation en vol est une fonction linéaire de la charge emportée (charge utile + carburant).

A chaque escale, l'appareil est ravitaillé en carburant. Celui-ci est acheté au tarif local (en dollars par kilo). Le tarif local dépend de la date.

Lorsque l'appareil atterrit à la fin de son vol, ainsi qu'à l'aéroport de départ, ses réservoirs peuvent contenir une quantité résiduelle non consommée lors du parcours du tronçon précédent. Pour effectuer le tronçon suivant, il est généralement nécessaire d'ajouter au réservoir une quantité qui permet d'atteindre l'escale ou l'aéroport suivant. Il est cependant possible d'emporter une quantité supérieure à ce qui est strictement nécessaire. Ce supplément peut être intéressant si le tarif local est particulièrement bas et si le tronçon suivant n'est pas trop long. On fera l'hypothèse que la valeur financière d'une quantité résiduelle est à calculer au tarif de l'endroit où cette quantité est observée, donc à l'atterrissage (= valeur de revente). On notera que l'appareil est présumé avoir consommé la quantité résiduelle à l'aéroport de départ et qu'il faut donc la lui imputer, mais qu'il n'a pas consommé celle qui subsiste après l'atterrissage final, et qu'il ne faut donc pas la lui imputer puisqu'elle n'aura pas servi au vol.

12.3.2 Construction du schéma conceptuel

L'objectif de calcul du coût des vols transparait clairement dans ce texte. Il se traduit par de nombreuses règles d'imputation et de ventilation des composants de ce coût. Nous limitant à la construction de la base de données, il nous faudra repérer les parties de l'énoncé qui décrivent les données, et donc détecter et écarter les concepts et les règles qui correspondent à des faits et propriétés dérivables. Nous les abandonnerons d'autant plus volontiers que leur analyse sera précisément l'objectif de la deuxième partie de cet ouvrage.

Afin de ne pas lasser le lecteur, nous ne détaillerons pas l'analyse de l'énoncé et la construction progressive du schéma conceptuel. Nous nous contenterons de proposer un exemple de solution (figure 12.3), accompagné d'une brève description des types d'entités.

Les types d'entités **TYPE-APPAREIL**, **TYPE-VOL**, **TYPE-TRONCON** et **AEROPORT** définissent les propriétés communes à tous les vols effectués. Ils représentent des concepts indépendants du temps, du moins à court terme. Les types d'entités **VOL**, **TRONCON** et **ESCALE-VOL** représentent au contraire les faits journaliers concernant les vols effectués et les tarifs locaux du carburant. Ils constituent en quelque sorte la *matérialisation* temporelle respectivement de **TYPE-VOL**, **TYPE-TRONCON** et **AEROPORT**, auxquels ils sont associés.

Une entité **TYPE-APPAREIL** représente un modèle d'appareil. Elle est caractérisée par la dénomination du modèle (Modèle), la capacité maximale des réservoirs (Cap-réservoir), la consommation en vol d'un appareil à vide, équipage compris, mais sans passagers, ni fret, ni carburant, exprimée en kilos de carburant par km (Consom-vide), la consommation en vol nécessaire au transport d'un kilo de charge sur un km (Consom-charge), la charge maximale qu'un appareil peut emporter (Charge-ut-max). Remarquons qu'on ne représente pas ici l'appareil qui effectue un

vol déterminé, mais seulement son modèle. Par exemple, il n'existera qu'une entité **APPAREIL** dont `Modèle` = « AIRBUS A330 ».

Une entité **TYPE-VOL** représente un type de vol caractérisé par un code identifiant (`ID-Tvol`) et par la suite des types de tronçons dont il est constitué (**TYPE-TRONCON**).

Une entité **AEROPORT** représente un aéroport. Elle est caractérisée par le nom en clair de l'aéroport (`Nom`) et code de l'aéroport (`Code-aéroport`).

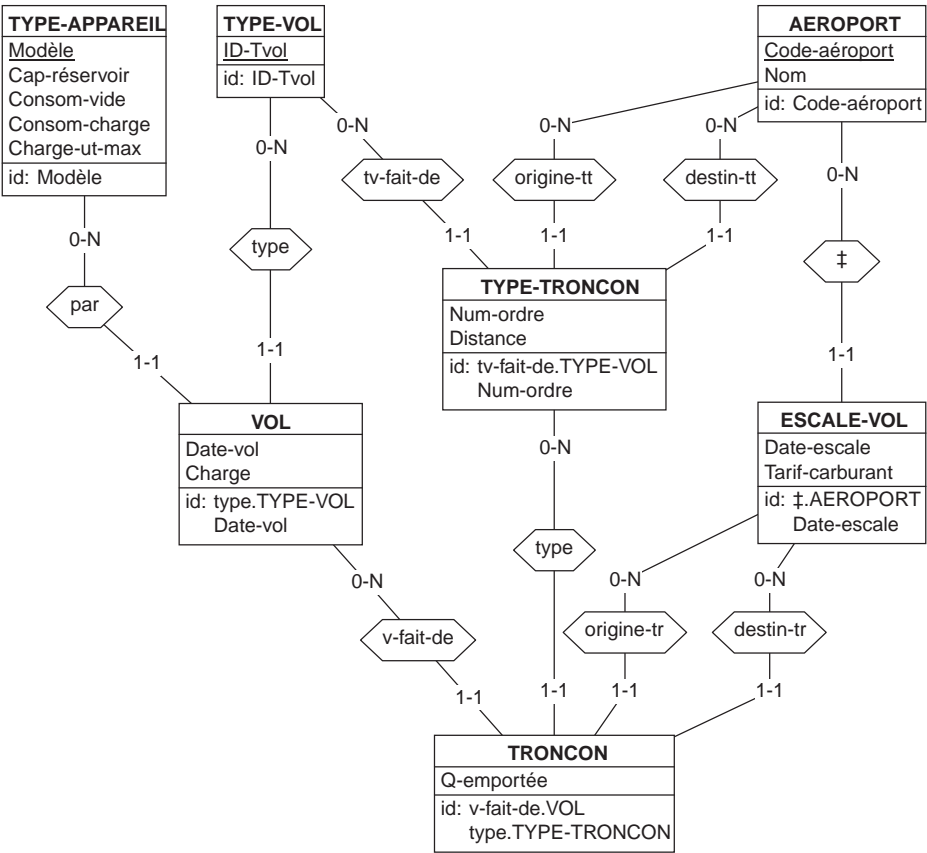


Figure 12.3 - Schéma conceptuel de la base de données des vols

Une entité **TYPE-TRONCON** représente une section ininterrompue d'un type de vol. Elle est attachée à une entité **TYPE-VOL**, et porte un numéro d'ordre parmi les types de tronçons du type de vol (`Num-ordre`), la longueur du tronçon en km (`Distance`), l'aéroport d'origine (**AEROPORT** *via* `origine-tt`) et l'aéroport de destination (**AEROPORT** *via* `destin-tt`).

Une entité **VOL** représente un vol réel effectué à une date déterminée. Elle correspond à une entité **TYPE-VOL** qui en définit les propriétés stables. Elle est caracté-

12.3.3 Production du schéma de tables

L'application des règles de traduction en structures relationnelles conduit au schéma de la figure 12.4.

La définition des structures physiques et la traduction en code SQL sont laissées à l'initiative du lecteur.

12.4 EXERCICE

12.1 L'étude de cas concernant les voyages aériens a laissé de côté certaines propriétés qui auraient dû être ajoutées au schéma conceptuel. On peut en particulier retenir les trois propriétés suivantes :

1. le TYPE-VOL du VOL d'un TRONCON est le TYPE-VOL du TYPE-TRONCON de ce TRONCON;
2. l'AEROPORT correspondant à l'ESCALE-VOL de laquelle part un TRONCON est l'AEROPORT d'origine du TYPE-TRONCON de ce TRONCON;
3. l'AEROPORT correspondant à l'ESCALE-VOL à laquelle aboutit un TRONCON est l'AEROPORT de destination du TYPE-TRONCON de ce TRONCON.

Rédiger les requêtes SQL qui recherchent dans la base de données proposée en 12.4 les données qui ne vérifient pas ces propriétés.

Exploiter ensuite ces propriétés pour simplifier la structure de la table TRONCON, en particulier en éliminant les colonnes redondantes et donc inutiles.

PARTIE 2

LES MODÈLES DE CALCUL

Chapitre 13

Introduction

Ce chapitre initie le lecteur au concept de tableur et de modèle de calcul, et discute de la nécessité d'une approche méthodologique adaptée à l'élaboration de modèles. Il décrit la structure de cette deuxième partie.

13.1 LE TABLEUR

C'est en 1979 qu'est apparu le premier tableur sur ordinateur personnel, VisiCalc de VisiCorp. Développé pour l'Apple II, il est réputé avoir donné à cette machine le statut d'ordinateur de gestion. Son principal successeur, 1-2-3 de Lotus, fut conçu en 1983 pour le PC d'IBM, pour lequel il a également constitué un argument de vente décisif.

Ces logiciels apparaissent comme les premiers outils de développement destinés à l'utilisateur. En effet, ils proposaient, pour les problèmes qui peuvent s'exprimer sous la forme de résultats à calculer par des formules à partir de données, une implantation directe de ces résultats, de ces données et de ces formules dans une grille de cellules visibles à l'écran. Par une programmation directe et visuelle, les objets du problème devenaient les objets du programme. Le concept premier était celui de **feuille de calcul**, simulation assez naturelle d'une feuille quadrillée sur laquelle l'utilisateur écrit les objets de son problème, et dont la principale propriété est d'évaluer toute formule rangée dans une cellule. Il suffisait donc de repérer les variables du problème, de décider de leur localisation dans la feuille de calcul, puis de ranger les formules de calcul dans les cellules ainsi choisies. Le problème était à la fois défini et résolu.

Les tableurs actuels, beaucoup plus puissants que leurs ancêtres, partagent cependant toujours avec eux le concept original de feuille de calcul et son mode de fonctionnement. Ils constituent encore aujourd'hui une part majeure des ventes de logiciels de bureautique.

13.2 LE CONCEPT DE MODÈLE

L'implication de cette technologie sur les rapports entre l'utilisateur et l'informatique est plus profonde qu'il n'y paraît. Pour la première fois en effet, il est possible de définir la solution informatique d'un problème comme une description naturelle de celui-ci. Il n'est donc plus nécessaire de traduire cette description en structures de données et en algorithmes, puis ces derniers en programmes, pour obtenir la solution cherchée. La principale difficulté reste alors celle de la définition du problème : quelles informations cherche-t-on à obtenir, de quelles données dispose-t-on, comment définit-on ces informations à partir de ces données ?

On a coutume de désigner par le terme de **modèle** cette définition du problème. En toute généralité, un modèle d'un phénomène ou d'une situation est une abstraction construite pour décrire, comprendre, expliquer, prévoir, voire gérer ou piloter ce phénomène. Le modèle est une image fidèle du phénomène en ce sens que les informations qu'il nous fournit le décrivent correctement. Bien sûr, un modèle n'est qu'une simplification et une réduction du phénomène et, en cela, n'est valable que dans des limites qui doivent être précisées au départ (voir par exemple [Courbon, 1993] pour une introduction à la notion de modèle).

Les tableurs s'adressent principalement à des phénomènes, à des situations, et plus généralement à des *domaines d'application* qui peuvent être décrits par des grandeurs quantitatives et par des relations de calcul entre ces grandeurs. On parlera donc volontiers de **modèles de calcul** à leur sujet.

13.3 CONSTRUCTION D'UN MODÈLE DE CALCUL

L'introduction de cet ouvrage a montré que les tableurs ne résolvaient pas tous les problèmes liés au développement d'applications par l'utilisateur. L'examen des erreurs typiques montre clairement que l'essentiel des difficultés réside dans l'analyse du problème plus que dans son implantation dans une feuille de calcul. Malheureusement il existe peu de propositions concrètes en cette matière, et en tout cas rien n'a été présenté à ce jour qui puisse recueillir un consensus similaire à celui qui concerne les bases de données : on attend toujours un modèle Entité-association et une méthode MERISE qui seraient destinés aux modèles de calcul.

Les raisons tiennent sans doute au public auquel ces logiciels s'adressent en priorité : les utilisateurs. Leur approche des problèmes est essentiellement pragmatique. Le raisonnement adopte les concepts proposés par l'outil pour décrire le problème à résoudre, et n'éprouve nullement le besoin d'un niveau de description plus abstrait, du moins pour des problèmes de faible complexité. Les concepts de

qualité d'une part, de démarche méthodologique structurée, dotée de modèles spécifiques, de processus et d'outils, d'autre part, qui sont familiers aux informaticiens de profession, seront généralement perçus comme encombrants, voire redondants et inutiles par l'utilisateur moyen.

13.4 DESCRIPTION DE LA DEUXIÈME PARTIE

La structure de cette deuxième partie est identique à celle de la première; elle est donc constituée de quatre blocs.

- *Les concepts* : le chapitre 14 décrit le concept de modèle pris en charge par les tableurs courants, ainsi que les principales notions proposées par ceux-ci.
- *Les outils* : le tableur le plus populaire, EXCEL, est très brièvement décrit dans le chapitre 15.
- *Les méthodes de conception* : les chapitres 16 à 19 proposent des éléments pour la construction systématique de modèles de calcul destinés aux tableurs. Le chapitre 17 décrit un mode de formulation abstraite de modèles qui est indépendant des tableurs, le chapitre 18 propose une démarche de définition et de validation de modèles abstraits et le chapitre 19 montre quelques techniques simples de traduction d'un modèle abstrait dans une feuille de calcul.
- Les deux *études de cas* du chapitre 12 sont réexaminées dans le chapitre 20 dans la perspective des modèles de calcul.

13.5 POUR EN SAVOIR PLUS

La littérature consacrée aux tableurs, et plus spécifiquement à chacun d'entre eux, est particulièrement abondante : chaque éditeur couvrant le marché de l'informatique personnelle pratique se doit d'offrir des ouvrages consacrés aux principaux tableurs, à leur apprentissage et même à leur utilisation avancée. Citons, parmi une centaine de références possibles, [Blattner, 1999]. En revanche, les aspects méthodologiques y sont singulièrement absents, si on excepte quelques recommandations générales et pour l'essentiel évidentes, même pour l'utilisateur débutant.

[Courbon, 1993] décrit, *via* plusieurs exemples, l'usage des tableurs comme outils d'aide à la décision. [Holsapple, 1987a] étudie également le tableur comme composant de systèmes experts dans les organisations, et formule des recommandations concernant le processus de développement de modèles.

Dans le domaine de la formulation de modèles abstraits, on peut considérer LUCID [Ashcroft, 1977] comme l'un des premiers langages non procéduraux incluant la notion de dimension (section 17.9) notamment comme support de l'expression de formules de récurrence. La modélisation équationnelle est également étudiée dans [Geoffrion, 1987], [Krishnan, 1992] et dans [Prywes, 1989]. Ce dernier, ainsi que [Prywes, 1983], aborde aussi certains des problèmes qui se posent lors de la traduction de modèles abstraits en programmes. Le lecteur trouvera dans

[Becker, 1993] un cadre de modélisation plus large, incorporant la formulation équationnelle dans une approche orientée objet.

La représentation du couplage entre bases de données et modèles de calcul est abordée dans les propositions de Geoffrion, décrites dans [Geoffrion, 1987] et développées dans les documents plus techniques [Hainaut, 1990], [Geoffrion, 1991a], [Geoffrion, 1991b] et [Hainaut, 1994]¹. Certains aspects plus pratiques sont décrits dans [Holsapple, 1987a]. [Lazimi, 1988] et [Lazimi, 1990] présentent ce couplage comme une extension déductive d'un schéma Entité-Association : les grandeurs dérivées sont considérées comme des attributs virtuels de types d'entités existants.

Les aspects méthodologiques sont peu abordés dans la littérature, du moins en ce qui concerne les modèles de calcul au sens strict. On citera [Blanning, 1987], qui propose une extension de la théorie relationnelle aux fonctions de calcul, les travaux de Geoffrion [Geoffrion, 1987], une étude des problèmes de cohérence dans les modèles abstraits couplés aux bases de données [Hick, 1991], ainsi que [Konopasek, 1984] qui étudie de manière très concrète et largement illustrée la formulation des modèles non directionnels (section 17.2).

Citons encore [Brown, 1987], [Ronen, 1989] et [Teo, 1999] comme études sur le comportement des concepteurs de modèles.

Alors que les références méthodologiques accessibles sont nombreuses dans le domaine des bases de données (on pense aux ouvrages sur MERISE et UML par exemple), le lecteur désireux de s'informer sur ce thème dans le domaine des modèles de calcul visitera plus les bibliothèques que les librairies. La seconde partie du présent ouvrage est une tentative de réponse à ce problème.

1. Les chapitres de cette référence relatifs à cette question sont aussi disponibles sur le site Web de l'ouvrage.

Chapitre 14

Concepts des modèles de calcul

Un modèle est l'expression d'un problème sous la forme d'un ensemble de grandeurs numériques et/ou logiques, ainsi que de relations qui permettent de calculer certaines de ces grandeurs (les résultats) à partir des autres (les données).

Il existe des processeurs, tels que les tableurs, qui évaluent les résultats d'un modèle en fonction de ses données. Un tableur gère et traite un modèle sous la forme d'un tableau implanté dans une feuille de calcul.

14.1 MODÈLES ET PROCESSEURS DE MODÈLES

Les concepts et les outils dont nous allons parler et que nous allons utiliser couvrent un champ de problèmes où le traitement de séries de données numériques constitue le mode privilégié de résolution.

Typiquement, l'expression d'un problème consiste en un ensemble de données de départ, un ensemble de résultats désirés, ainsi qu'un ensemble de relations mathématiques permettant de définir les résultats en fonction des données. Cette expression constitue le modèle d'un domaine d'application tel qu'un suivi de budget, une gestion comptable, un planning, le choix d'investissements, un circuit électrique, une construction mécanique ou un biorythme.

Evaluer ou exécuter un modèle consiste à rechercher les résultats, à partir d'un jeu déterminé de données, grâce aux relations entre ces résultats et ces données. Un

processeur de modèles est un programme général qui est capable d'exécuter un modèle dont on lui donne les composants. Il existe des processeurs spécialisés dans certaines classes de problèmes, tels que les processeurs de modèles financiers. Il existe également des processeurs généraux, tels que TK!Solver et EUREKA (tous deux aujourd'hui disparus), MACSYMA, MATHEMATICA ou MathCAD. Ceux-ci exploitent des techniques particulières, comme la résolution symbolique (les solutions sont exprimées sous la forme de formules) ou numérique (les solutions sont exprimées sous forme numérique) comme la programmation linéaire ou la programmation dynamique.

Afin d'illustrer ces concepts, nous ferons cependant appel à ces processeurs plus simples¹ mais d'usage universel que sont les **tableurs**. Leur principale caractéristique est d'offrir une interface visuelle simple et intuitive.

14.2 MODÈLES ET TABLEAUX

Le tableau est une forme élémentaire de modèle dans laquelle les informations sont présentées sous la forme de lignes et de colonnes. Cette organisation tabulaire permet la mise en évidence de relations entre les données. Par exemple, telle colonne représente les informations relatives à une même année, ou telle ligne contient les totaux des lignes qui précèdent, etc. La figure 14.1 représente un tableau financier reprenant, par année, les entrées, les sorties et les soldes. On y indique également le total des entrées, celui des sorties et celui des soldes.

	2004	2005	2006	TOTAUX
ENTREES	123	142	168	433
SORTIES	112	130	174	416
SOLDES	11	12	-6	17

Figure 14.1 - Un tableau financier typique

La structure d'accueil dans laquelle nous allons écrire notre tableau s'appelle une feuille électronique ou **feuille de calcul** (figure 14.2). Celle-ci se présente comme la simulation dans la mémoire de l'ordinateur d'une grande feuille de papier. Un quadrillage en lignes et colonnes y délimite des cases ou **cellules**. En désignant chaque colonne par une lettre (ou deux au-delà de la 26^e) et chaque ligne par un numéro, nous pouvons assigner à chaque cellule une **adresse**, qui est constituée de ses coordonnées. On parlera par exemple de la cellule B5 (colonne B, ligne 5). On peut également désigner un ensemble de cellules, qu'on appellera une **plage**. La

1. Encore que les tableurs offrent en général des fonctions de résolution appartenant traditionnellement aux processeurs plus puissants : résolveurs d'équations linéaires ou non linéaires, *goal-seeking* (recherche de la donnée qui fournirait un résultat qu'on impose).

plage D3:F4 est constituée des cellules inscrites dans le rectangle de diagonale D3 à F4, c'est-à-dire des cellules D3, D4, E3, E4, F3 et F4.

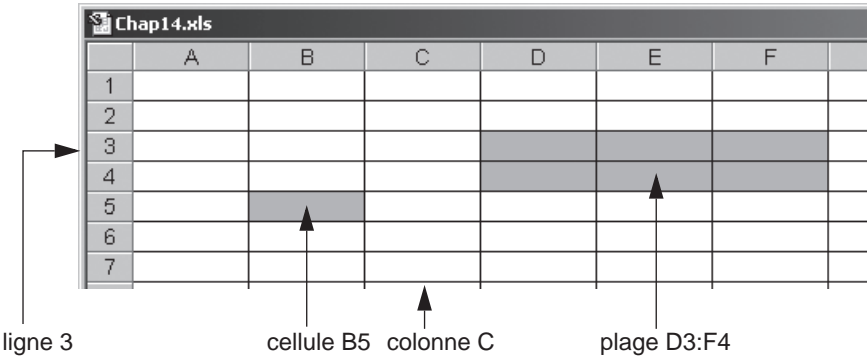


Figure 14.2 - Une feuille de calcul (vide)

Il est possible de ranger une information dans chaque cellule. Si on range le tableau financier de la figure 14.1 dans la feuille de calcul de la figure 14.2, celle-ci prend l'apparence (qu'il faut distinguer du contenu, comme on va le voir) de la figure 14.3.

	A	B	C	D	E	F
1						
2			2004	2005	2006	TOTAUX
3		ENTREES	123	142	168	433
4		SORTIES	112	130	174	416
5		SOLDES	11	12	-6	17
6						

Figure 14.3 - Le tableau financier implanté dans une feuille de calcul

Nous adoptons ici une acception un peu particulière du terme de *feuille de calcul*. Selon les manuels de certains tableurs, ce terme désigne un tableau, ou en tout cas un modèle représenté dans un tableau. Or rien n'interdit d'implanter plusieurs modèles dans une même feuille, ni d'utiliser plusieurs feuilles pour représenter un modèle.

14.3 REPRÉSENTATION D'UN MODÈLE DANS UNE FEUILLE DE CALCUL

En analysant plus avant le tableau financier, on observe qu'il est composé de trois types d'éléments (figure 14.4).

Le premier type consiste en des informations dont le seul but est de permettre une interprétation aisée des données et des résultats. Elles ne font l'objet d'aucun calcul

et servent uniquement à améliorer la lisibilité du tableau pour ses utilisateurs. Nous appellerons ces informations de présentation des **libellés**.

Un deuxième type regroupe les **données**, c'est-à-dire des informations à partir desquelles les résultats vont être calculés, mais qui ne découlent elles-mêmes d'aucun calcul.

Enfin, les éléments qui constituent des **résultats calculés** forment le troisième groupe.

	A	B	C	D	E	F
1						
2			2004	2005	2006	TOTAUX
3		ENTREES	123	142	168	433
4		SORTIES	112	130	174	416
5		SOLDES	11	12	-6	17
6						

libellés données résultats

Figure 14.4 - Les types de contenu des cellules d'une feuille de calcul

Les éléments du type libellé ou donnée doivent le plus souvent être inscrits à la main (au clavier, ou par tout moyen externe au tableur, tel qu'une requête de base de données) dans les cellules qui leur correspondent. En revanche, les résultats ne seront pas introduits tels qu'ils apparaissent dans la figure 14.4. On inscrira dans les cellules correspondantes la **formule de calcul** de chaque résultat. C'est au tableur d'évaluer ces formules et d'inscrire lui-même le résultat obtenu. Ainsi, lorsqu'une donnée est modifiée, le tableur recalcule les formules concernées et affiche les nouveaux résultats sans qu'un ordre explicite doive lui en être donné.

Nous examinerons à présent la manière dont s'exprime une formule de calcul. Considérons par exemple le cas du solde de 2004, qui doit être inscrit en C5. Cette valeur est obtenue par une expression du type :

entrées de 2004 - sorties de 2004

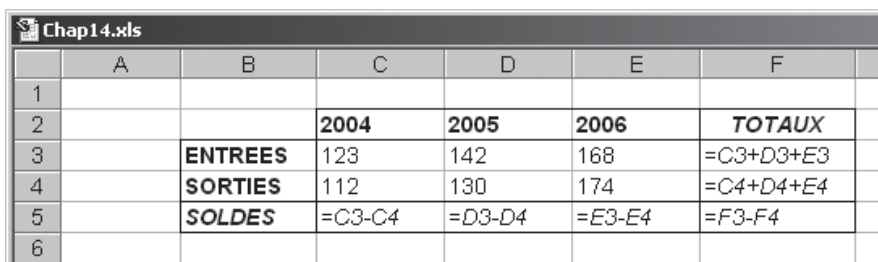
Pour traduire cette formule en termes accessibles au tableur, on observe que les entrées de 2004 se trouvent en C3, et les sorties de 2004 en C4. Par conséquent, on inscrit en C5 la formule :

= C3 - C4

De même, le total des entrées s'écrit en F3 sous la forme suivante :

= C3 + D3 + E3

La figure 14.5 montre le contenu réel du tableau. Ce sont ces informations qu'il faudra introduire dans le feuille de calcul afin de faire évaluer le modèle par le tableur.



	A	B	C	D	E	F
1						
2			2004	2005	2006	TOTAUX
3		ENTREES	123	142	168	=C3+D3+E3
4		SORTIES	112	130	174	=C4+D4+E4
5		SOLDES	=C3-C4	=D3-D4	=E3-E4	=F3-F4
6						

Figure 14.5 - Contenu réel du tableau

14.4 LE MARCHÉ DES TABLEURS

Les tableurs sont disponibles sur tous les types d'ordinateur personnel, sur les stations de travail, et même sur certaines grosses machines. Le premier des tableurs tels qu'on les connaît actuellement fut VisiCalc (*the Visible Calculator*), opérationnel sur les micro-ordinateurs APPLE II à la fin des années 70. Il fut suivi de 1-2-3 de Lotus, sur PC d'IBM. Après une période très riche en offre logicielle de tableurs, il reste actuellement un petit nombre de produits concurrents : essentiellement Excel et Works de Microsoft, Works de Claris, StarOffice de Sun, Quattro Pro de Corel et 1-2-3 de Lotus.

Les tableurs modernes incorporent également d'autres fonctions liées à l'activité présumée de leur utilisateur type : traitement de texte, gestion de bases de données, communications (transmission de données à distance), présentation graphique notamment. Pour les détails, le lecteur est renvoyé à la littérature spécialisée.

Chapitre 15

Un tableur type : EXCEL

Le tableur EXCEL, développé et distribué par Microsoft, à l'origine comme concurrent de 1-2-3 de Lotus, est l'un des tableurs les plus puissants et les plus complets actuellement disponibles sur le marché. On en décrit les principales fonctions et règles d'utilisation.

15.1 PRÉSENTATION D'EXCEL

Ce choix d'EXCEL est arbitraire. Des logiciels tels que 1-2-3 (Lotus), QUATTRO PRO (Corel) ou StarOffice (Sun), pour n'en citer que quelques-uns, auraient tout aussi bien pu remplir le rôle que nous recherchons, c'est-à-dire celui d'un véritable outil de développement d'applications professionnelles de taille moyenne.

Excel permet de développer, gérer et évaluer des feuilles de calcul complexes, mais offre également des fonctions de gestion de données, de traitement de texte, de dessin, de présentation graphique, de communication et de développement de procédures et d'interfaces utilisateur. En bref, Excel se présente, comme Access d'ailleurs¹, comme un environnement d'application raisonnablement complet pour des problèmes de taille moyenne.

Son calculateur offre les opérateurs arithmétiques, des opérateurs de manipulation de chaînes de caractères, des fonctions scientifiques (trigonométriques, exponentielle, logarithme, etc.), des fonctions financières, des fonctions statistiques, des fonctions temporelles (arithmétique de date) ainsi que des opérateurs logiques (et,

1. À tel point qu'une même application pourrait être développée en Access ou en Excel.

ou, non), pour ne citer que les familles principales. Tout comme le serait un texte, une feuille de calcul est en principe en mémoire centrale. Il est possible de sauvegarder sur disque le contenu d'une feuille de calcul sous forme d'un fichier et de ranger en mémoire une feuille rangée dans un fichier. Il est enfin possible d'imprimer des parties d'une feuille de calcul.

Il est hors de question de décrire ici les fonctions d'un logiciel dont la documentation technique approche les deux mille pages. Nous évoquerons seulement quelques outils, caractéristiques et fonctions qui auront un impact direct sur nos préoccupations méthodologiques et qui, en particulier, nous permettront de traduire simplement et efficacement la spécification abstraite d'un modèle.

15.2 LA FEUILLE DE CALCUL

L'écran montre une plage de la feuille de calcul (le reste de la feuille est invisible). La partie supérieure et la partie gauche du cadre identifient les lignes et les colonnes visibles. Parmi les cellules visibles, il en est une qui est mise en évidence, généralement en vidéo inverse. Il s'agit de la cellule courante, ou sélectionnée, c'est-à-dire celle qui fera l'objet de la prochaine opération demandée. On peut aussi considérer qu'il existe un curseur de cellule et que celui-ci est en général sur la cellule courante. Si plusieurs cellules sont sélectionnées, elles constituent la plage sur laquelle la prochaine opération sera effectuée.

Toute chaîne de caractères introduite au clavier est stockée dans la cellule courante, qu'il s'agisse d'une donnée numérique, d'un libellé ou d'une formule.

Il est possible de découper l'affichage en deux ou quatre, verticalement et horizontalement. Les deux ou quatre fragments agissent comme des caméras indépendantes dirigées vers des régions différentes de la même feuille de calcul.

15.3 ORGANISATION DES FEUILLES DE CALCUL ET DES MODÈLES

Les documents construits avec EXCEL, c'est-à-dire les feuilles de calcul, mais également les feuilles de graphique et les feuilles de macros (composants procéduraux), peuvent être rangés dans des **classeurs**. Une même feuille peut appartenir à plusieurs classeurs.

Une cellule ou plage d'une feuille peut être référencée dans une formule qui se trouve dans une autre feuille. Son adresse (ou son nom) est préfixée par le nom de la feuille à laquelle elle appartient. Il est donc possible de construire un modèle sur plusieurs feuilles et de définir des modèles différents qui se partagent des feuilles communes. La documentation suggère quelques applications du concept de feuilles multiples :

- une feuille est définie comme fusion, consolidation ou agrégation de plusieurs feuilles de données; par exemple, chaque département possède une feuille

décrivant l'état de ses comptes, et une feuille de synthèse intègre ces données en un seul tableau ;

- une feuille contient un modèle dans tous ses détails ; chaque feuille annexée reprend les résultats selon une présentation personnalisée ;
- un modèle complexe a été décomposé en modèles plus simples, répartis entre plusieurs feuilles spécifiques.

Un modèle d'une certaine complexité, ou du moins certaines de ses parties, est naturellement structuré de manière hiérarchique. Partant par exemple des ventes par produit et par région, on y définit également les totaux par produit, les totaux par région puis le total général. En outre, disposant de ces valeurs par jour, on les calculera par mois et par année. On définit ainsi le concept de vente ventilée selon trois dimensions : région, produit, date, ainsi que différents niveaux d'agrégation selon chacune de ces dimensions ou selon plusieurs d'entre elles. Afin de présenter plus lisiblement ces informations, il serait utile de masquer les niveaux hiérarchiques les plus détaillés. Par exemple, il est inutile de montrer les ventes détaillées lorsqu'on n'est intéressé que par les résultats mensuels par région. Le concept de **plan d'une feuille** permet cette visualisation condensée. Le concepteur définit les groupes hiérarchiques verticaux et horizontaux de son tableau (il indique par exemple les ventes puis leur total). Il peut alors demander de n'en voir que le *plan*, c'est-à-dire de masquer les niveaux inférieurs à un niveau déterminé.

15.4 LES COMPOSANTS D'UN MODÈLE

15.4.1 Désignation de cellules

Une cellule est désignée par son adresse. Une plage le sera par les adresses d'une diagonale : B3:D5. Toute cellule ou plage peut recevoir un nom par lequel elle pourra être désignée. L'usage de ce nom permet la construction de formules plus lisibles.

15.4.2 Le contenu des cellules

Tout caractère affichable est introduit dans la cellule courante. La touche ↵ ou un déplacement du curseur termine l'introduction. Une suite de caractères qui commence par un chiffre, éventuellement précédé d'un + ou d'un -, est interprétée comme une valeur numérique. Une suite de caractères commençant par le symbole = est interprétée comme une formule à évaluer. Une valeur commençant par tout autre caractère est considérée comme un libellé. Un libellé est une valeur qui peut être manipulée par des fonctions de traitement de chaînes. On notera que le symbole ' (apostrophe) en première position n'est pas considéré comme un caractère, mais comme l'indication d'un libellé : si les caractères qui suivent sont des chiffres, ils ne seront pas considérés comme définissant une valeur numérique, mais un simple libellé. Par convention, une cellule qui contient un libellé est considérée comme vide pour les fonctions agrégatives (somme, moyenne, nb, etc.) mais comme non valide pour les fonctions arithmétiques.

15.4.3 Les formules

Une formule est une expression dont l'évaluation fournit une valeur qui sera, selon le cas, un nombre, une chaîne de caractères, un booléen, une date, etc. Une formule numérique est construite à l'aide des opérateurs arithmétiques (+, -, /, *, ^) et fait intervenir des constantes et le contenu de cellules :

```
= C2-2*B2
= -A4
= 18-B8^2
```

Des parenthèses permettent de modifier les priorités des opérateurs :

```
= 2*(G1-3*B5)
= 1/(1-D4)
```

Une expression peut aussi utiliser des fonctions mathématiques :

= log (A2-D4)	(logarithme en base 10)
= 1-sin (2*B3)	(sinus)
= pi () *B2^2	(valeur de π)
= racine (1-cos (A3) ^2)	(racine carrée)
= ent (B2+0.5)	(partie entière)

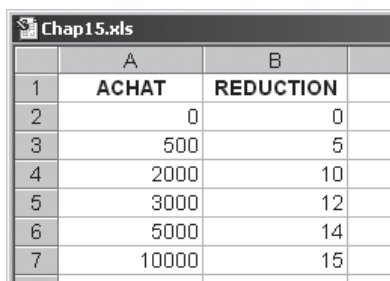
Il existe un jeu de **fonctions agrégatives** dont l'argument est une liste² d'expressions et/ou de désignations de plages (une valeur non effective correspond à une cellule vide) et qui renvoient une valeur élémentaire :

nb (B3 :H3)	(nombre de valeurs effectives)
somme (A2 ;D1 :D17)	(somme des valeurs effectives)
moyenne (D1 ;E2 ;F3)	(moyenne des valeurs effectives)
max (B3 :C10 ;A1-3)	(valeur maximum)
ecartype (E1 :E17)	(écart-type)

Il existe encore d'autres fonctions mathématiques et statistiques, des fonctions financières, des fonctions de date et des fonctions de **consultation de table**. Ces dernières méritent une brève description.

Considérons deux ou plusieurs suites verticales de cellules voisines, formant autant de colonnes de valeurs en correspondance. La colonne numéro 1 représente, comme le montre le tableau 15.1, des montants d'achat et la colonne numéro 2 les réductions accordées (il pourrait y avoir des colonnes 3, 4, etc.). Ce tableau indique qu'un achat de 0 à 499 euros ne donne lieu à aucune réduction (0 %), un achat de 500 à 1999 euros entraîne une réduction de 5 %, et ainsi de suite. Il est impossible de calculer au sens propre cette correspondance, sinon par une structure complexe de fonctions **si** emboîtées, comme on le verra ci-après.

2. Les éléments d'une liste sont séparés par le symbole ";" si la machine est configurée pour la langue française. De même, les nombres s'expriment à l'aide d'une virgule décimale. Pour la langue anglaise, ces symboles ont respectivement "," et ".".



	A	B
1	ACHAT	REDUCTION
2	0	0
3	500	5
4	2000	10
5	3000	12
6	5000	14
7	10000	15

Figure 15.1 - Une table de réduction

La fonction `RECHERCHEV(arg;table;nb;mode)` permet d'exprimer la consultation de telles tables :

- `arg` représente l'argument de recherche : constante, adresse de cellule, expression de calcul,
- `table` désigne la table des valeurs, sous la forme d'une plage,
- `nb` désigne la colonne dont on extrait la valeur (de 1 à n),
- et `mode` la manière dont la valeur de `arg` est recherchée dans la colonne 1 : `arg = faux` indique qu'on recherche une valeur égale (dans ce cas la table n'est pas nécessairement triée sur la colonne 1) et `arg = vrai` indique qu'on recherche la plus grande valeur qui soit inférieure ou égale à `arg` (la table doit alors être triée sur la colonne 1).

Par exemple, `RECHERCHEV(2350;A2:B7;2;vrai)` renvoie la valeur de la réduction correspondant, dans la table A2:B7, à la plus grande valeur de montant d'achat de la table qui soit inférieure ou égale à 2350, soit ici 10 %. Une valeur de `arg` qui serait inférieure à la plus petite des valeurs de la première colonne provoquerait une erreur. Autres exemples :

`RECHERCHEV(400;A2:B7;2;vrai)` renvoie 0,
`RECHERCHEV(3000;A2:B7;2;vrai)` renvoie 12,
`RECHERCHEV(14000;A2:B7;2;vrai)` renvoie 15.
`RECHERCHEV(4500;A2:B7;1;vrai)` renvoie 3000.
`RECHERCHEV(4500;A2:B7;2;faux)` provoque une erreur.

Il est possible également de définir des **expressions conditionnelles**, grâce à la fonction `si`. Les arguments de cette fonction forment une liste de trois expressions. La première est une condition, la deuxième est la valeur que renvoie la fonction lorsque la condition est vraie, et la troisième partie est la valeur de la fonction lorsque la condition est fausse. Ces deux valeurs se présentent comme des expressions quelconques. Quelques exemples :

= `si (B2=C2;1;-1)` si la condition B2=C2 est vraie, la valeur est 1, sinon elle est -1.
 = `si (B2+1>2*C2+C3;somme (A2:J2) ;somme (A3:J3))`

si la condition est vraie, la valeur correspond à la première somme, sinon à la seconde.

= si (A1<B1;1;si (A1=B1;2;3)) vaut 1, 2, 3 selon que A1 est <, =, > B1.

Une condition possède une valeur logique VRAI ou FAUX. Il est d'ailleurs possible de manipuler explicitement des valeurs logiques.

Ainsi, si la cellule B2 contient la formule = (B3 < B4), on pourra écrire dans toute autre cellule :

= si (B2;B3;B4)

qui s'interprète comme suit :

Si B2 est VRAI, c'est-à-dire si B3<B4, alors cette formule renvoie la valeur de B3, sinon elle renvoie celle de B4.

Des expressions logiques complexes seront construites à l'aide des opérateurs logiques et, ou et non, qui s'expriment sous la forme de fonctions :

et (B2>0;B3>0)	≡ B2>0 et B3>0
ou (B2>0;B3>0)	≡ B2>0 ou B3>0
non (B2>0)	≡ non B2>0

Ce format nous permet d'écrire des formules complexes :

non (ou (B2>0;et (B3>0;non (B4=0))))

Remarque

Une valeur numérique peut être traitée comme une valeur logique : 0 est interprété comme faux et toute valeur non nulle comme vrai. En revanche, une valeur logique n'a pas d'équivalent numérique.

15.5 MODIFICATIONS ÉLÉMENTAIRES D'UN MODÈLE

Étant donné un tableau, ou plus généralement un modèle, déjà constitué, on peut être amené à en modifier la structure et l'apparence afin de l'étendre ou d'en améliorer la lisibilité. En résumé, il est possible de :

- spécifier le format d'affichage des nombres, des dates, des libellés d'un ensemble de cellules;
- modifier la largeur d'une ou plusieurs colonnes ou la hauteur de lignes;
- insérer une nouvelle colonne (ou ligne) vide entre deux colonnes (lignes) existantes (les formules des cellules déplacées sont ajustées en conséquence);
- enlever une colonne (ligne) existante;
- effacer le contenu de cellules.

15.6 DÉPLACEMENT ET COPIE DE FRAGMENTS DE MODÈLES

L'opération de *déplacement* permet de restructurer un tableau en en déplaçant des plages. Si une formule contient des adresses relatives ou absolues (voir ci-dessous) désignant des cellules de la plage déplacée, elle sera automatiquement ajustée. D'autre part, les formules de la plage déplacée qui contiennent des adresses relatives sont également ajustées. Pour exécuter cette commande, il faut sélectionner la plage à transférer puis tirer celle-ci jusqu'à l'endroit de destination (ou encore par couper/coller dans le cas général).

L'opération de *copie* effectue une copie du contenu d'une plage vers un autre endroit de la feuille. La plage d'origine peut se réduire à une seule cellule, ce qui sera le cas le plus fréquent. On sélectionnera la plage à copier puis on étendra (en tirant la petite poignée en bas à gauche) jusqu'à remplir l'espace de destination (ou encore par copier/coller dans le cas général).

Il existe des possibilités de copie avec variantes : copier non pas les formules, mais le résultat de leur évaluation, copier en additionnant ou soustrayant les valeurs copiées aux valeurs anciennes plutôt qu'en remplaçant celles-ci (**Copie spéciale**).

15.6.1 Adresses relatives et adresses absolues

Bien qu'il soit étranger au fonctionnement du tableau (il n'intervient que lors de l'implantation du tableau dans une feuille de calcul), le concept d'**adresse relative** pose quelques problèmes au modélisateur débutant et mérite qu'on s'y arrête un instant.

Considérons le tableau défini à la figure 15.2 qui donne l'évolution de différents montants (par exemple 1000, 2000, 5000, etc.), placés à un taux déterminé (par exemple 5.25 %), durant un certain nombre d'années (par exemple 1, 2, 3).

Chap15.xls				
	A	B	C	D
1	Taux :	0,0525		
2				
3	Montant :	1000	2000	5000
4	1 an	=B3*(1+B1)	=C3*(1+B1)	=D3*(1+B1)
5	2 ans	=B4*(1+B1)	=C4*(1+B1)	=D4*(1+B1)
6	3 ans	=B5*(1+B1)	=C5*(1+B1)	=D5*(1+B1)

Figure 15.2 - Évolution de montants placés à un taux déterminé

La formule de calcul est identique, *mutatis mutandis*, dans toutes les cellules B4:D6; elle exprime en effet que le montant en fin d'année est égal à celui de l'année précédente augmenté des intérêts. On serait donc tenté de définir une seule formule, par exemple celle qui se trouve en B4 (figure 15.3), et de la recopier dans toutes les autres cellules similaires, en espérant que le tableur se charge d'ajuster les adresses de manière que les formules représentent bien le comportement des montants.

Chap15.xls				
	A	B	C	D
1	Taux :	0,0525		
2				
3	Montant :	1000	2000	5000
4	1 an	=B3*(1+B1)		
5	2 ans			
6	3 ans			

Figure 15.3 - Définition d'une formule type

C'est effectivement ce qui va se passer. En copiant la formule type « =B3 * (1+B1) » de la cellule B4 dans les cellules voisines de la plage B4:D6, EXCEL va convertir les adresses qui y apparaissent comme suit :

L'adresse B3 qui apparaît dans la formule de la cellule B4 est interprétée, non pas comme la désignation de la cellule B3, mais plutôt comme celle de la cellule qui se trouve, par rapport à B4, dans la même colonne (B) et une position vers le haut (3-4=-1). Par conséquent, lors de la copie de cette formule dans la cellule C4, l'adresse B3 sera adaptée en tenant compte de cette interprétation, et donc traduite en C3 (même colonne, numéro de ligne -1), qui désigne bien la cellule qui se trouve au-dessus de C4. Il en sera de même lorsque cette formule sera recopiée en B5, où B3 deviendra B4.

B3 est donc considérée comme une *adresse relative* par rapport à B4. Après recopie de cette formule-type qui se trouve en B4 dans les cellules voisines, on obtient le tableau 15.4.

Chap15.xls				
	A	B	C	D
1	Taux :	0,0525		
2				
3	Montant :	1000	2000	5000
4	1 an	=B3*(1+B1)	=C3*(1+C1)	=D3*(1+D1)
5	2 ans	=B4*(1+B2)	=C4*(1+C2)	=D4*(1+D2)
6	3 ans	=B5*(1+B3)	=C5*(1+C3)	=D5*(1+D3)

Figure 15.4 - État du tableau après recopie de la formule-type B4 dans les cellules voisines

Le résultat n'est évidemment pas celui qu'on escomptait : si l'adresse du *montant* a bien été ajustée comme espéré, le tableur en a fait de même pour l'adresse du *taux*, B1, qui est devenue respectivement B2, B3, C1, C2, etc. En fait, ce taux est identique pour toutes les formules, et il aurait fallu que le tableur considère B2, non comme une adresse relative, mais bien comme une **adresse absolue** : B1 doit rester B1 quelle que soit la formule dans laquelle elle apparaît.

On peut inhiber l'ajustement automatique d'une adresse, et donc la rendre absolue, lors d'une recopie en préfixant un ou les deux composants de cette adresse

du signe \$. Ainsi, l'adresse B1 écrite sous la forme \$B\$1 ne sera jamais ajustée lors d'une recopie et sera donc préservée; l'adresse \$B1 gardera intact son code de colonne, alors que son numéro de ligne sera ajusté et B\$1 gardera intact son numéro de ligne, alors que son code de colonne sera ajusté.

La formule-type à ranger en B4 doit donc s'écrire sous la forme, « =B3*(1+\$B\$1) » de manière à produire le modèle complet de la figure 15.5. Notons encore une fois que ce mécanisme n'a pour but que de permettre la copie de formules, et ne change rien au comportement du tableau lors de l'évaluation des formules. Sur ce dernier point, les tableaux 15.2 et 15.5 sont strictement équivalents, le calculateur d'EXCEL ignorant les signes \$ dans les adresses.

Chap15.xls				
	A	B	C	D
1	Taux :	0,0525		
2				
3	Montant :	1000	2000	5000
4	1 an	=B3*(1+\$B\$1)	=C3*(1+\$B\$1)	=D3*(1+\$B\$1)
5	2 ans	=B4*(1+\$B\$1)	=C4*(1+\$B\$1)	=D4*(1+\$B\$1)
6	3 ans	=B5*(1+\$B\$1)	=C5*(1+\$B\$1)	=D5*(1+\$B\$1)

Figure 15.5 - Modèle correct obtenu par copie de la formule-type en B4

15.7 LES RÉFÉRENCES CIRCULAIRES

Ce terme désigne la situation particulière d'une formule qui dépend directement ou indirectement de la cellule dans laquelle elle est rangée. Les deux fragments suivants contiennent chacun une référence circulaire :

Exemple 1 : dans C4 : = (C1 - C4) / 2 la formule contient l'adresse de sa propre cellule;

Exemple 2 : dans C2 : =2*C3 le contenu de C2 dépend de celui de C3,
 dans C3 : =C4+C5 . . . qui dépend de celui de C4,
 dans C4 : =C2/2 . . . qui dépend de celui de C2.

Comme tous les tableurs, EXCEL détecte l'existence de telles boucles de cellules (ici {C4} et {C2, C3, C4}), dénommées **références circulaires**, et attire l'attention de l'utilisateur sur ce qui est à considérer *a priori* comme une anomalie. En effet, cette circularité correspond à une boucle de calcul sans fin, ce qui n'est pas un mode de fonctionnement normal pour un tableur. Si l'utilisateur confirme cette situation, EXCEL propose de traiter cet ensemble de formules d'une manière itérative, soit un certain nombre de fois N, soit jusqu'à ce que les valeurs successives d'une cellule de la boucle convergent, c'est-à-dire ne diffèrent plus que d'une quantité minimale ε. Les valeurs N et ε sont fixées par l'utilisateur.

3. Très opportunément choisi comme symbole universellement accepté de stabilité...

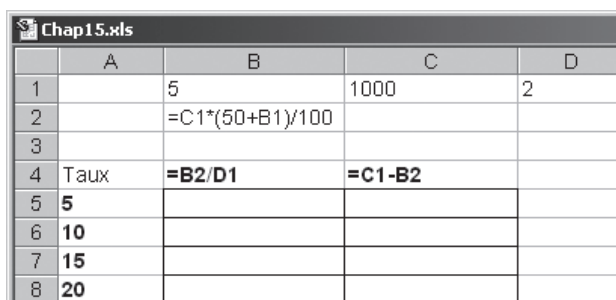
15.8 FONCTIONS DE BASES DE DONNÉES

EXCEL offre également des fonctions dites de *bases de données*. Elles permettent de considérer une plage comme une base de données (en fait une table au sens de la première partie), de gérer et interroger, de manière très élémentaire d'ailleurs, le contenu de cette plage.

Ces concepts et ces fonctions seront décrits dans la troisième partie de [Hainaut, 1994], également disponible sur le site Web de l'ouvrage.

15.9 LES TABLES DE DONNÉES

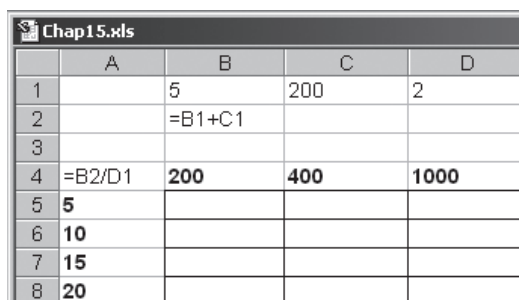
Une **table de données à simple entrée** est une liste de cellules contiguës, placées verticalement (ou horizontalement), et qui contiennent les valeurs successives à attribuer à une donnée d'un modèle. A la droite de la table sont rangées des formules dépendant, directement ou non, de cette donnée. EXCEL exécute le modèle pour chaque valeur de la table et place le résultat de l'évaluation des formules à l'intersection de la ligne de chaque valeur de donnée et de la colonne de chaque formule. Dans l'exemple de la figure 15.6, la plage A5:A8 est la table de données liée à la cellule B1, et B4:C4 contient les formules à évaluer pour chacune des valeurs de B1. EXCEL range dans B5:B8 les valeurs de la formule B4 pour chaque valeur de B1 et dans C5:C8 celles de la formule C4 :



	A	B	C	D
1		5	1000	2
2		=C1*(50+B1)/100		
3				
4	Taux	=B2/D1	=C1-B2	
5	5			
6	10			
7	15			
8	20			

Figure 15.6 - La plage A5:A8 est une table de données à simple entrée pour la cellule B1

EXCEL offre également la possibilité de définir une liste de valeurs pour deux données d'un modèle : les **tables de données à double entrée**. Ici, on ne peut calculer les valeurs correspondantes que d'une seule formule. Un modèle basé sur une table de données à double entrée se présente comme suit (figure 15.7). Les valeurs A5:A8 sont associées à la cellule B1, les valeurs B4:D4 sont associées à la cellule C1, et la formule à évaluer pour tous les couples <B1,C1> est placée à l'intersection de ces deux rangées de données, en A4. Les résultats apparaissent dans la plage B5:D8 définie par les deux jeux de valeurs.



	A	B	C	D
1		5	200	2
2		=B1+C1		
3				
4	=B2/D1	200	400	1000
5	5			
6	10			
7	15			
8	20			

Figure 15.7 - Les plages A5:A8 et B4:D4 constituent une table de données à double entrée pour les cellules B1 et C1

Ces structures sont présentées comme outil de simulation. Elles sont cependant d'un usage plus général, comme on le suggérera plus tard.

15.10 LES SCÉNARIOS

Étant donné une liste de cellules, un **scénario** est un jeu de valeurs à assigner à chacune de ces cellules. Ce jeu de valeurs est enregistré sous un nom déterminé. On peut enregistrer un nombre quelconque de scénarios pour une même liste de cellules. Nous référant à la figure 14.3, nous pourrions examiner ce tableau selon diverses hypothèses (scénarios) quant aux dépenses (cellules B3:D3). Nous pourrions alors définir quelques jeux de valeurs pour ces cellules, tels que {100,130,170}, {110,130,172}, {112,130,174}, {112,130,180}, constituant chacun un scénario. Ces quatre jeux seront définis et enregistrés sous les noms DEPENSES_1, DEPENSES_2, ..., DEPENSES_4 par exemple.

En principe, un scénario est destiné à représenter une situation particulière qu'on désire comparer à d'autres, ou à laquelle on désire revenir plus tard. Il est généralement présenté comme un outil de simulation, mais cet usage n'est pas exclusif.

15.11 MACROS ET FONCTIONS PERSONNALISÉES

Une **macro** est au départ une suite d'instructions enregistrées (dans une **feuille de macros**) qui correspondent à des actions que l'utilisateur pourrait commander à l'aide des menus et d'activation d'outils. Elle permet une économie de manipulation car l'exécution de cette suite d'actions sera lancée par une seule autre action : exécuter la macro. Sans entrer dans le détail, les instructions d'une macro⁴ sont introduites dans les cellules successives d'une colonne de la feuille de macro. A

4. Le terme *macro* est emprunté au domaine des langages de programmation, où certains mécanismes existent qui permettent de définir une suite d'instructions comme une nouvelle instruction : une *macro-instruction*.

chaque action que l'utilisateur peut demander correspond une instruction de format déterminé. Le langage de macros a cependant été enrichi d'instructions et de constructions qui en font un langage de programmation procédural complet : boucles, alternatives, variables locales, passage d'arguments, manipulation d'adresses de cellules, interaction avec l'utilisateur, communications avec l'environnement, etc⁵.

Une macro peut prendre deux formes. La première, ou **macro** proprement dite, définit une commande qui s'ajoute à celles déjà proposées par EXCEL. La seconde, appelée **fonction macro**, ou fonction personnalisée, définit une nouvelle fonction qui s'ajoute à celles offertes par EXCEL et peut à ce titre être utilisée dans des formules.

15.12 LES RÉSOLVEURS AVANCÉS

Ces nouveaux processeurs enrichissent considérablement la palette d'outils de résolution offerte par les tableurs traditionnels. Ils permettent en effet d'aborder des problèmes qu'il serait malaisé d'exprimer à l'aide du schéma *données, résultats, formules de calcul* utilisé jusqu'ici. Nous en citerons deux, la fonction de la **valeur cible** et le **solveur**.

15.12.1 La valeur cible⁶

Considérons une cellule C2 contenant une formule qui référence, directement ou non, la cellule C1 qui contient elle-même une constante. En fonctionnement normal, C1 reçoit une valeur (une donnée) et le tableur calcule, entre autres choses, la valeur de C2 (le résultat). Ce comportement répond aux questions du type suivant : quelle est la valeur de C2 étant donné la valeur de C1 ? La fonction EXCEL de valeur cible permet de poser le problème inverse : quelle devrait être la valeur de C1 pour que C2 ait une valeur déterminée ? En se reportant au tableau 14.3, il serait possible de demander à EXCEL la valeur des sorties de 2006 (D3) qui rendrait le total des soldes (E5) égal à 20.

15.12.2 Le solveur

Le solveur d'EXCEL est un résolveur d'équations, ou plus précisément un optimiseur exploitant des techniques de programmation linéaire et non linéaire.

Très schématiquement, il fonctionne comme suit. On considère un modèle de calcul dans lequel on repère une cellule résultat R, dite cellule cible, et des cellules de données Di, dites cellules variables. Le solveur résout les problèmes du type suivant :

5. Ces fonctions peuvent aussi être écrites en Visual Basic. A partir d'Office 2000, Microsoft généralise l'usage d'une version unique de *VB for Application* pour le développement de composants procéduraux dans ses logiciels bureautique.

6. Ou *goal-seeking* en anglais.

- trouver des valeurs des D_i qui conduisent à $R = v$, où v est une constante fixée par l'utilisateur;
- déterminer les valeurs des D_i qui rendent minimum (maximum) la valeur de R ;
- déterminer les valeurs des D_i qui rendent minimum (maximum) la valeur de R , de telle manière que des contraintes (conditions⁷) spécifiées soient respectées;
- trouver les valeurs des D_i qui satisfont les contraintes spécifiées.

Lorsque le solveur a terminé, les cellules R et C_i ont reçu des valeurs qui résolvent le problème posé, pour autant que celui-ci admette au moins une solution, et que le solveur l'ait trouvée. L'utilisation du solveur réclame quelques connaissances sur les méthodes numériques d'optimisation et de résolution de systèmes d'équations. On consultera par exemple [Dion, 1999].

7. Aux conditions habituelles telles que « $B3 > B7 + B8$ », qu'on trouve par ailleurs dans la fonction `si()`, s'ajoute la contrainte importante « $B7 = \text{entier}$ », permettant de poser des problèmes en nombres entiers. On peut par exemple imposer que le nombre de véhicules soit un nombre entier.

Chapitre 16

Construction d'un modèle de calcul

Ce chapitre pose les principes de l'élaboration d'un modèle implanté dans une feuille de calcul comme solution à un problème posé. On y dresse un parallèle avec la démarche d'élaboration d'une base de données. Les chapitres 17 à 19 vont en développer les éléments méthodologiques.

Les chapitres qui suivent sont consacrés au problème de la construction et de l'implantation d'un modèle de calcul dans une feuille de calcul afin qu'on puisse l'exécuter.

Rappelons brièvement qu'un **modèle** est constitué d'un ensemble de **données**, d'un ensemble de **résultats** attendus et d'un ensemble de **relations** (logiques ou de calcul) exprimant les résultats en fonction des données. En outre, il existe une classe de logiciels qui permettent de représenter un modèle sous la forme d'un **tableau** enregistré dans une feuille de calcul et de l'exécuter, c'est-à-dire de calculer les résultats correspondant aux données qu'on lui fournit.

L'expression d'un petit modèle simple ne requiert pas d'habileté particulière : la connaissance des principes de fonctionnement du tableur choisi et un minimum de soin permettent d'exprimer rapidement le modèle sous la forme d'un tableau. Il n'en va cependant pas de même lorsque le modèle est plus complexe. L'expression directe d'un tel modèle dans une feuille électronique devient laborieuse, procède par ajustements et remaniements successifs, et conduit à un résultat peu lisible. L'examen du modèle (des règles notamment) est difficile et n'offre pas une vue structurée et logique. L'écran montre en effet le résultat de l'évaluation des règles,

mais rend difficile la lecture des règles, même en mode *formule*. Le résultat est en définitive peu fiable et difficile à modifier. La nécessité d'un mode d'expression du modèle qui soit plus global, plus simple, plus intuitif et indépendant de l'outil apparaît bien vite.

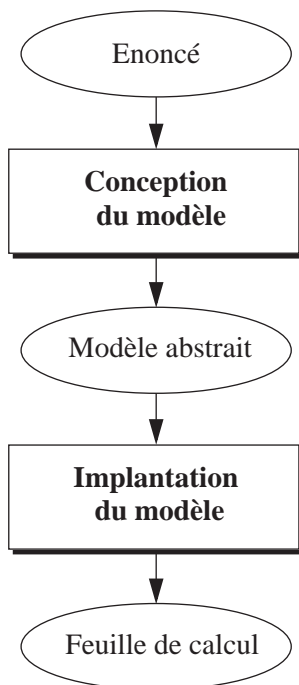


Figure 16.1 - Structure générale de la méthode de construction d'un modèle de calcul à l'aide d'un tableur

Le problème de la construction d'un modèle présente donc des similitudes étroites avec celui de la construction d'un programme ou d'une base de données ; en particulier, il se décompose en deux sous-problèmes indépendants. Le premier est de nature conceptuelle : il s'agit de l'élaboration d'un modèle qui décrive avec précision le domaine d'application indépendamment du tableur. Le second est de nature technique, et consiste à représenter le modèle ainsi élaboré à l'aide d'un outil tel qu'un tableur.

L'élaboration d'un modèle indépendant de tout outil de réalisation doit se baser sur un formalisme d'expression à la fois intuitif et abstrait¹. Le chapitre 17 est consacré à la présentation des concepts permettant de définir un modèle d'une manière abstraite, et des propriétés de ces concepts. Le chapitre 18 aborde le

1. Le terme *abstrait* indique que les expressions qu'on construit à l'aide de ce formalisme sont indépendantes des détails techniques de l'outil de mise en œuvre. Une expression mathématique est *abstraite* par rapport aux instructions en FORTRAN, Pascal ou EXCEL qui en calculent la valeur.

problème de l'élaboration de l'expression abstraite d'un modèle tandis que le chapitre 19 étudie la traduction de l'expression abstraite d'un modèle en une forme exécutable par un tableur tel que EXCEL. Le chapitre 20 propose deux études de cas qui illustrent certains de ces concepts.

Les concepts et les raisonnements présentés dans cette deuxième partie sont parallèles à ceux qui ont été développés dans la première, ainsi que le montre le tableau ci-dessous.

	Bases de données	Modèles de calcul
Formalisme conceptuel	Type d'entités, d'associations, attribut, identifiant, etc.	Grandeur, règle, contrainte
Spécification conceptuelle	Schéma conceptuel	Modèle abstrait
Formalisme concret	Table, colonne, identifiant, clé étrangère	Feuille de calcul, cellule, formule, macro
Spécification concrète	Schéma de tables SQL	Feuille de calcul EXCEL
Outils	ORACLE, SQL Server, DB2	EXCEL, 1-2-3, WORKS

Les concepts qui vont être développés concernent une classe très large de problèmes. Il existe des domaines plus spécifiques qui utilisent leurs propres concepts et leurs propres méthodes. Nous citerons par exemple la recherche opérationnelle et l'économétrie. Les problèmes relevant de ces domaines seront bien entendu plus adéquatement traités selon ces approches spécifiques.

Chapitre 17

Expression abstraite d'un modèle

On propose un formalisme d'expression de modèles de calcul indépendant des outils de résolution. On y développe les notions de modèle, de grandeur, de règle de définition. On décrit les grandeurs à définition multiple, les grandeurs et modèles simples et dimensionnés, les fonctions agrégatives et les modèles basés sur la récurrence. On présente la notion de graphe de dépendance et celle de modularisation.

17.1 INTRODUCTION

Alors que le domaine des bases de données a fait depuis plus de trois décennies l'objet de développements importants sur le plan méthodologique, rien, ou presque, n'existe en ce qui concerne les modèles de calcul, si ce n'est sur le plan des publications scientifiques. On pourrait citer les formalismes d'expression abstraite proposés dans [Geoffrion, 1987], [Blanning, 1987] ou [Lazimi, 1988], mais ces propositions n'ont pas acquis la popularité du modèle Entité-Association par exemple.

Les concepts que nous allons développer reprennent, en les dégageant du contexte des outils qui les mettent en œuvre, en les simplifiant et en les généralisant, les principaux objets et structures disponibles dans les tableurs, notamment dans leurs versions les plus récentes. Qu'on nous comprenne bien cependant, il ne s'agit pas de proposer des notions qui couvrent dans les moindres détails toutes les fonctionnalités de tous les tableurs présents et à venir. Il en résulterait un formalisme complexe et très certainement incohérent et redondant, c'est-à-dire incompatible avec nos objectifs de simplicité et de généralité.

Les concepts de base sont réduits à l'extrême : un **modèle** est constitué de **grandeurs**, dont certaines sont des **données** et d'autres des **résultats**, et de **règles** exprimant les résultats en fonction des données. Certains modèles sont dits **simples**, lorsque chacune de leurs grandeurs ne peut prendre qu'une valeur à la fois. Lorsqu'une ou plusieurs de ses grandeurs peuvent en revanche prendre non pas une valeur, mais une liste de valeurs, le modèle est dit **dimensionné**.

Un formalisme de modélisation n'a de sens que s'il peut servir de support à une démarche systématique et intuitive de construction de modèles. Dans cette perspective, le formalisme sera enrichi de concepts et de propriétés qui seront utiles pour la construction et la validation des solutions qu'il permet d'exprimer. C'est ainsi que nous décrirons des notions telles que les **graphes de dépendance** et la **modularisation** d'un modèle.

17.2 GRANDEURS ET RÈGLES

L'élaboration d'un modèle est un processus délicat de conceptualisation et d'abstraction. Du point de vue intellectuel, il présente de fortes analogies avec la construction d'un schéma conceptuel de données (chapitre 10). Étant donné une situation qui constitue le **domaine d'application** ou le **système** à représenter, il s'agit d'y repérer les **concepts** fondamentaux qui paraissent représentatifs de ce domaine. Ces concepts ou **grandeurs** sont ici **mesurables**, c'est-à-dire qu'on peut leur associer une **valeur** à tout instant (plus généralement, un concept a des propriétés mesurables, mais nous ne ferons pas de distinction pour l'instant). Cette valeur sera le plus souvent **numérique** (montant des exportations, quantité en stock, distance), mais aussi parfois **logique** (ou booléenne : vrai ou faux, grand ou petit, actif ou inactif), ou tout simplement **qualitative**, et représentée par un libellé pris dans un ensemble déterminé (nom d'un département, marque d'une voiture). Si on affecte une valeur à chaque grandeur, l'ensemble de ces valeurs représente un état déterminé du domaine.

Les concepts ne sont évidemment pas indépendants. Certaines grandeurs dépendent d'autres grandeurs. Il existe donc des **relations** ou des **lois** entre les grandeurs, de la même manière qu'il existe des types d'associations entre les types d'entités au niveau des données. Les relations sont ici très précisément spécifiées, par exemple sous la forme d'une expression mathématique ou logique. Grâce aux relations, il est possible d'obtenir la valeur d'une grandeur à partir des valeurs d'autres grandeurs.

Supposons que D0, D1, D2 soient trois grandeurs quelconques et qu'on puisse calculer D0 à partir de D1 et D2 selon la formule¹ 17.0.

$$D0 = 2 * D1 * (1 - D2) + 1 \quad (\text{expr. 17.0})$$

1. Le symbole « * » représente l'opérateur de multiplication et le symbole « / » celui de division.

Cette même formule, pour autant qu'on l'interprète comme une relation entre ces grandeurs, aurait pu s'écrire de l'une des manières suivantes, choisies parmi d'autres formulations :

$$\begin{aligned} D1 &= (D0 - 1) / (2 * (1 - D2)) \\ D2 &= 1 - (D0 - 1) / (2 * D1) \\ D0 - 2 * D1 * (1 - D2) + 1 &= 0 \end{aligned}$$

D'une manière plus générale, et en ne retenant que l'existence des grandeurs impliquées et de leurs relations, on peut aussi écrire ces relations sous les formes suivantes :

$$\begin{aligned} D0 &= f(D1, D2) && \text{(expr. 17.1)} \\ D1 &= g(D0, D2) && \text{(expr. 17.2)} \\ D2 &= h(D0, D1) && \text{(expr. 17.3)} \\ k(D0, D1, D2) &= 0 && \text{(expr. 17.4)} \end{aligned}$$

Dans certains cas cependant, toutes les expressions ne sont pas à retenir; par exemple $x = y^2$ est univoque, tandis que son inverse ne l'est pas. Il en est de même de la fonction *abs* (valeur absolue). D'autres fonctions n'ont tout simplement pas d'inverse, telles que *ent* (partie entière), ainsi que les fonctions de définition multiple que nous verrons plus loin.

Il est donc important de déterminer parmi les grandeurs celles pour lesquelles on dispose de valeurs (les données), et celles dont on recherche les valeurs (les résultats). Ainsi, à partir de l'expression la plus générale d'une relation (expr. 17.4), on peut déduire chacune des formes particulières (expr. 17.1, 17.2, 17.3) en fixant le rôle des grandeurs. Dans ces dernières expressions, la grandeur résultat est respectivement D0, D1 et D2.

La discussion ci-dessus est justifiée dans la mesure où les logiciels d'exécution de modèles se distinguent quant au moment où le rôle des grandeurs doit être défini. Les **processeurs d'équations** permettent généralement qu'on définisse au départ les grandeurs et les relations, puis qu'*a posteriori* on choisisse parmi les grandeurs celles qui seront les données et celles qui seront les résultats. On peut d'ailleurs changer ce rôle à n'importe quel moment. Pour ces logiciels, la forme de la relation — ou **directionnalité** — est indifférente, c'est-à-dire que les expressions 17.1 à 17.4 sont équivalentes. En revanche, les **tableurs** n'admettent que des modèles dans lesquels on a choisi *a priori* et définitivement les données et les résultats, comme suggéré dans les expressions 17.1 à 17.3.

Appliquons ces principes à un problème concret. Considérons par exemple les grandeurs et la relation suivantes, décrivant un domaine d'application bien connu :

$$\begin{aligned} V &: \text{vitesse du véhicule} \\ D &: \text{distance à parcourir} \\ T &: \text{temps du parcours} \\ T &= D / V \end{aligned} \quad \text{(expr. 17.5)}$$

Dans un processeur d'équations, ce modèle permettra de calculer n'importe quelle grandeur à partir des deux autres. Dans un tableur, en revanche, ce modèle permet

seulement de calculer T en fonction de D et V , supposés connus. On devra, pour obtenir la généralité de comportement d'un processeur d'équations, définir *trois modèles distincts*, contenant les mêmes grandeurs, ainsi que l'une des relations 17.6 à 17.8.

$$T = D / V \quad (\text{expr. 17.6})$$

$$D = T * V \quad (\text{expr. 17.7})$$

$$V = D / T \quad (\text{expr. 17.8})$$

On en conclut que si on se dispose à utiliser un tableur, il est impératif de poser au départ ce qu'on cherche et ce dont on dispose, c'est-à-dire le rôle des grandeurs. Il en serait de même si le modèle devait être mis en œuvre sous la forme d'un programme rédigé dans un langage traditionnel comme Pascal, BASIC, C ou Java. Les modèles exécutables par un tableur constituent donc une classe restreinte de modèles. C'est de cette classe que nous discuterons.

Nous appellerons désormais **règle de définition** une relation directionnelle, c'est-à-dire une expression de définition d'une grandeur. Une règle est constituée d'une partie gauche, le nom de la grandeur définie, et d'une partie droite, l'expression de définition de cette grandeur.

Une grandeur qui représente un phénomène mesurable n'a de sens que si on en connaît le domaine de valeurs, ainsi que l'unité de mesure. En outre, il est essentiel de décrire la signification de chaque grandeur par une courte phrase.

Le domaine de valeurs peut être spécifié comme un type standard, tel que les *entiers*, les *réels*, les *booléens* ou les *dates*. Il peut également être défini comme un intervalle : $[1-10]$, $[1999-2010]$, ou comme un ensemble de valeurs explicitement énumérées : $\{0;1;2;3\}$, $\{5;19,5;25\}$, $\{\text{lundi;mardi;...;dimanche}\}$. Reprenons les expressions précédentes en les complétant de la spécification des domaines de valeurs et des unités.

V : réel (m/s) ; vitesse du véhicule
 D : entier (m) ; distance à parcourir
 T : réel (s) ; temps du parcours
 $T = D / V$

Dans la suite de ce chapitre, qui sera consacrée aux concepts fondamentaux des modèles et à leur structure, nous omettrons la spécification des domaines de valeurs, des unités et même des descriptions textuelles des grandeurs, afin de ne pas surcharger les exemples.

17.3 NOTION DE MODÈLE

Revenons à l'expression 17.1, dans laquelle nous décidons que $D0$ est le résultat cherché, et que $D1$, $D2$ constituent les données. En désignant par \mathbf{D} la liste $\{D1, D2\}$, on peut aussi écrire la forme générale :

$$D0 = f(\mathbf{D}) \quad (\text{expr. 17.9})$$

Cette notation vectorielle peut s'étendre aux résultats et aux règles. Si \mathbf{R} désigne la liste des grandeurs à calculer, on peut symboliser le modèle de la manière suivante :

Données : \mathbf{D}
 Résultats : \mathbf{R}
 Règles : $\mathbf{R} = \mathbf{F}(\mathbf{D})$ (expr. 17.10)

où **D** représente la liste des données, **R** la liste des résultats et $\mathbf{R} = \mathbf{F}(\mathbf{D})$ la liste des règles de définition.

En ce qui concerne les expressions utiles, indiquons encore qu'une règle n'est pas nécessairement construite comme une formule mathématique. Par exemple, on peut admettre que la règle qui exprime le salaire brut en fonction de l'ancienneté et du niveau d'un salarié :

```
BRUT = br (ANCIENNETE, NIVEAU)
```

soit évaluée par consultation d'une table de barèmes, et non par une expression arithmétique.

Nous mettons encore en évidence une classe particulière de règles. Il s'agit des **contraintes** qui définissent les états valides des grandeurs, notamment celles qui jouent le rôle de données. Elles s'expriment sous la forme d'une condition qui doit être vraie pour tous les états cohérents du modèle. La règle

$$2 \leq \text{NIVEAU} \leq 12$$

exprime qu'à tout instant la valeur de la grandeur NIVEAU doit être comprise entre 2 et 12. Si NIVEAU est une donnée, cette règle correspond à une condition de validation contrôlant l'introduction des valeurs par l'utilisateur. Elle précise qu'une valeur qui viole cette condition doit être rejetée.

Données

ANCIENNETE, NIVEAU, PRIMES, INDEX

Résultats

BRUT, COTISATION_SOCIALE, NET_IMPOSABLE,
RETENUE-FISCALE, NET-PAYE

Règles

```
BRUT = (br(ANCIENNETE,NIVEAU) + PRIMES)*INDEX
COTISATION_SOCIALE = (0,1+NIVEAU/100)*BRUT
NET_IMPOSABLE = BRUT - COTISATION_SOCIALE
RETENUE_FISCALE = NET_IMPOSABLE*
(0,1+min(0,4;NET_IMPOSABLE/50000))
NET_PAYE = NET_IMPOSABLE - RETENUE_FISCALE
2 ≤ NIVEAU ≤ 12
```

Figure 17.1 - Un modèle de calcul du traitement d'un employé

Considérons à titre d'exemple le développement du modèle, fortement simplifié, de calcul du traitement d'un employé. Il pourrait se présenter comme dans la figure 17.1.

Tant les grandeurs que les règles sont présentées dans un ordre déterminé pour des raisons de convenance et non selon la séquence d'évaluation lors de l'exécution du modèle. Tout autre ordre aurait été adéquat. La liste des règles ne constitue donc pas une séquence d'instructions. Le texte de la figure 17.1 est de nature **déclarative**, et non **algorithmique**. En outre, une règle est à distinguer d'une instruction d'assignation, qui demande l'exécution d'une opération. Une règle définit une grandeur en fonction d'autres grandeurs.

Remarque

Nous aurons à représenter à la fois des nombre décimaux et des listes d'expressions, ce qui entraîne, selon les conventions françaises, des ambiguïtés. Par exemple, le minimum des valeurs 1, 2, 1, 7 et 0, 8 s'écrira $\min(1, 2, 1, 7, 0, 8)$, ce qui n'est pas particulièrement clair. D'autre part, l'usage du symbole ";" comme séparateur d'éléments de liste, s'il est acceptable dans le langage d'un tableur, est peu naturel dans une formulation abstraite. Nous conviendrons donc d'utiliser (1) la virgule décimale, (2) le point-virgule comme séparateur d'éléments dans la liste des arguments d'une fonction technique (min, max, etc.) et (3) la virgule comme séparateur d'éléments dans la liste des arguments d'un sous-modèle (tel que `br` ci-dessus).

17.4 DESCRIPTIONS EXTERNE ET INTERNE D'UN MODÈLE

Dans le modèle 17.1, toutes les grandeurs du modèle correspondent à des concepts explicites et visibles. Dans un modèle complexe, en revanche, il est pratique de considérer l'existence de concepts qui sont inconnus (c'est-à-dire dont l'existence n'est pas perçue) de l'utilisateur de ce modèle. Ces concepts (ou grandeurs) internes sont introduits de manière à faciliter l'expression, la construction et la compréhension du *détail* du modèle, mais sont inutiles à sa compréhension globale et à son utilisation.

Données

ANCIENNETE, NIVEAU, PRIMES, INDEX

Résultats

NET_PAYE

Règles

NET-PAYE = **np**(ANCIENNETE, NIVEAU, PRIMES, INDEX)
 $2 \leq \text{NIVEAU} \leq 12$

Figure 17.2 - Un modèle de calcul du traitement d'un employé - Description externe

Reprenons le modèle de la figure 17.1 en considérant qu'on désire seulement obtenir le traitement net payé à l'employé. Perçu de l'extérieur, le modèle se présente alors comme dans la figure 17.2, où **np** représente une fonction de calcul non précisée pour l'instant.

Si on détaille les règles, on est amené à définir les grandeurs internes que sont le *brut*, la *cotisation sociale*, le *net imposable* et la *retenue fiscale*. En tenant compte de cette distinction, nous pouvons réécrire le modèle 17.1 selon la figure 17.3.

Données

ANCIENNETE, NIVEAU, PRIMES, INDEX

Résultats

NET_PAYE

Grandeurs internes

BRUT, COTISATION_SOCIALE, NET_IMPOSABLE, RETENUE_FISCALE,

Règles

```
BRUT = (br(ANCIENNETE, NIVEAU) + PRIMES) * INDEX
COTISATION_SOCIALE = (0,1 + NIVEAU/100) * BRUT
NET_IMPOSABLE = BRUT - COTISATION_SOCIALE
RETENUE_FISCALE = NET_IMPOSABLE *
                    (0,1 + min(0,4; NET_IMPOSABLE/50000))
NET_PAYE = NET_IMPOSABLE - RETENUE_FISCALE
2 ≤ NIVEAU ≤ 12
```

Figure 17.3 - Un modèle de calcul du traitement d'un employé - Description complète

17.5 GRANDEURS À DÉFINITION MULTIPLE

Certaines grandeurs peuvent recevoir plusieurs expressions de définition selon le contexte dans lequel elles sont définies. Supposons à titre d'exemple que le concept d'allocation soit défini comme suit :

l'allocation est égale à 100 lorsque $NET_PAYE < 1000$

l'allocation est égale à $0,1 * NET_PAYE$ lorsque $1000 \leq NET_PAYE < 3000$

l'allocation est égale à 300 lorsque $NET_PAYE \geq 3000$.

On écrira la définition de ALLOCATION sous la forme suivante :

```
ALLOCATION = 100          si NET_PAYE < 1000
              0,1*NET_PAYE  si 1000 ≤ NET_PAYE < 3000
              300          si NET_PAYE ≥ 3000
```

Cette définition comporte trois branches, chacune d'elles constituant une définition qui s'applique dans le contexte décrit par la condition. La condition de la dernière branche peut être complémentaire de toutes les autres, et sera déclarée par la clause *sinon*, illustrée par la règle ci-dessous.

```

ALLOCATION =100                si NET_PAYE < 1000
                        0,1*NET_PAYE    si 1000 ≤ NET_PAYE < 3000
                        300              sinon

```

Le modèle de la figure 17.4 contiendra d'autres illustrations de ce concept. La définition d'une telle grandeur réclame une attention particulière. Il faut en effet respecter trois propriétés essentielles : la *complétude*, la *non-ambiguïté* et l'*absence de branches mortes*².

- La définition multiple doit être **complète** : la grandeur doit être définie dans tous les cas. L'usage d'une branche finale du type `sinon` garantit la complétude.
- La définition multiple doit être **non ambiguë** : il ne peut exister de situations telles que la grandeur soit définie plus d'une fois.
- Il ne peut exister de **branches mortes**, c'est-à-dire de conditions qui soient toujours fausses, quelles que soient les valeurs des données du modèle.

Ces propriétés seront étudiées plus en détail à la section 18.5.2.

17.6 GRANDEURS ET RÈGLES LOGIQUES

Les règles qui ont été proposées jusqu'ici définissent une grandeur numérique comme une expression arithmétique (ou mathématique) faisant intervenir d'autres grandeurs, ou constantes numériques. Les **grandeurs logiques** ou à valeurs booléennes constituent un concept des plus utiles pour l'expression naturelle de nombreux modèles tels que ceux qu'on destine à l'aide à la décision. Un modèle peut alors apparaître comme une **base de règles** dans l'acception traditionnelle du terme. Il est en effet admis qu'il soit possible de réaliser de petits **systèmes-experts** à l'aide de tableurs.

Nous illustrerons ce concept par le modèle de la figure 17.4 qui détermine la durée et le montant d'un prêt accordé à un client dont on connaît le salaire, l'âge, le montant du compte et la fiabilité (OUI/NON).

Remarquons que l'extension de la notion de règle à des grandeurs non numériques peut aussi couvrir d'autres types, tels que les types temporels et les chaînes de caractères, qui peuvent faire l'objet de fonctions de traitement spécifiques.

17.7 GRAPHE DE DÉPENDANCE

Il est intéressant, afin de mieux percevoir la structure d'un modèle, d'examiner d'une manière graphique les dépendances entre les grandeurs. On dira qu'une gran-

2. Ces contraintes peuvent être assouplies dans certains systèmes experts. Il n'en est cependant pas question lorsque le modèle doit être exécuté par un tableur, ou même par un processeur d'équations. En effet, sauf cas limités, ces outils ne sont pas conçus pour réagir adéquatement à des événements imprévus, et notamment aux situations d'ambiguïté.

deur A dépend directement d'une grandeur B s'il existe dans le modèle une relation du type

$$A = f(B, \dots)$$

Chaque grandeur du modèle est représentée par un nœud du graphe. Nous représenterons ensuite une dépendance telle que $A = f(B, \dots)$ par un arc orienté qui part du nœud B et qui aboutit au nœud A (on représente le fait que B influence A , ou que B intervient dans la définition de A).

Données

SALAIRE, AGE, COMPTE, FIABLE

Résultats

PRET_ACCORDE, DUREE, MONTANT

Grandeurs internes

CLIENT_JEUNE, CLIENT_MOYEN, CLIENT_AGE,
PRET_MODERE, PRET_ELEVE, BON_CLIENT, HAUT_SALAIRE

Règles

```
PRET_ACCORDE = FIABLE ou (HAUT_SALAIRE et COMPTE > 0)
DUREE = 0      si non PRET_ACCORDE
          10    si PRET_ACCORDE et CLIENT_JEUNE
          8     si PRET_ACCORDE et CLIENT_MOYEN
          6     si PRET_ACCORDE et CLIENT_AGE
MONTANT = 0    si non PRET_ACCORDE
          5 * SALAIRE si PRET_MODERE
          10 * SALAIRE si PRET_ELEVE
PRET_MODERE = PRET_ACCORDE et non BON_CLIENT
PRET_ELEVE  = PRET_ACCORDE et BON_CLIENT
BON_CLIENT  = FIABLE et HAUT_SALAIRE
HAUT_SALAIRE = SALAIRE > 1000000
CLIENT_JEUNE = AGE ≤ 30
CLIENT_MOYEN = AGE ≤ 60 et non CLIENT_JEUNE
CLIENT_AGE  = non (CLIENT_JEUNE ou CLIENT_MOYEN)
```

Figure 17.4 - Exemples de définitions multiples et de règles logiques

Le graphe de la figure 17.5 représente les dépendances directes présentes dans le modèle 17.3. Remarquons que si A dépend de B et si B dépend de C , alors A dépend aussi (indirectement) de C . On dira que la notion de dépendance est **transitive**. Le graphe de dépendance ne reprendra que les dépendances directes. On n'y inclura donc pas les dépendances transitives, qui le rendraient illisible.

Un tel graphe permet des observations sur la structure du modèle :

- une grandeur à laquelle n'aboutit aucun arc est nécessairement une donnée;
- une grandeur de laquelle ne part aucun arc est nécessairement un résultat;
- une grandeur origine et destination d'arcs est soit un résultat, soit une grandeur interne.

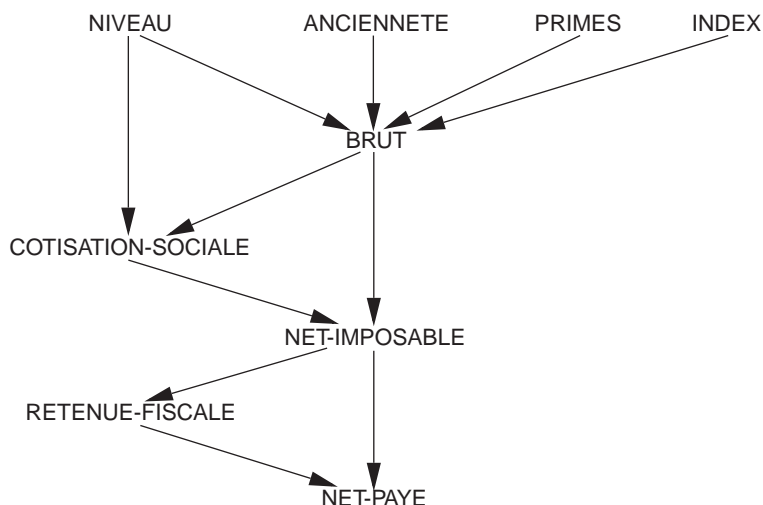


Figure 17.5 - Graphe de dépendance du modèle 17.3

Rien n'interdit qu'un résultat intervienne dans le calcul d'autres résultats (*cf.* modèle 17.1 par exemple). D'autre part, certains nœuds intermédiaires peuvent être considérés comme dignes d'intérêt, par exemple pour surveiller le comportement interne d'un modèle complexe (figure 17.6). Ces grandeurs, qui seront visibles dans la feuille de calcul, sont à considérer comme des résultats.

Données

A

Résultats

C, E

Grandeurs internes

B, D

Règles

$B = f(A)$

$C = g(B)$

$D = h(C)$

$E = k(D)$

Figure 17.6 - Les résultats ne sont pas toujours des nœuds terminaux

Comme on le verra dans la section 17.11, cette présentation synthétique des règles montre aussi clairement d'autres propriétés d'un modèle, telles que l'existence de relations récursives. Celles-ci définissent dans le graphe un **circuit**, c'est-à-dire une suite d'arcs tels qu'en les suivant dans le sens des flèches, il est possible de retrouver le point de départ. Un tel circuit indique que chacune de ses grandeurs est définie,

directement ou indirectement, à partir d'elle-même. Signalons aussi qu'on y repère immédiatement les grandeurs qui ne sont ni utilisées ni définies : elles ne sont concernées par aucun arc. Elles témoignent généralement de défauts de construction du modèle.

Notons enfin que le graphe de dépendance peut constituer un support de raisonnement *a priori*, préalablement à l'établissement des règles de définition³. On peut en effet le considérer comme l'ébauche de l'architecture du modèle, ébauche qui sera affinée par la détermination précise des règles.

Il est intéressant de savoir que les tableurs construisent pour chaque modèle introduit dans une feuille de calcul un graphe de dépendance qui leur permet d'évaluer rapidement les formules suite à la modification d'une donnée. Certains, tel EXCEL, peuvent même afficher les relations de dépendance entre cellules (figure 17.7) pour permettre à l'utilisateur de comprendre les liens entre cellules. Ce graphe devenant rapidement illisible, on en limitera l'usage à la visualisation des antécédents d'une cellule.

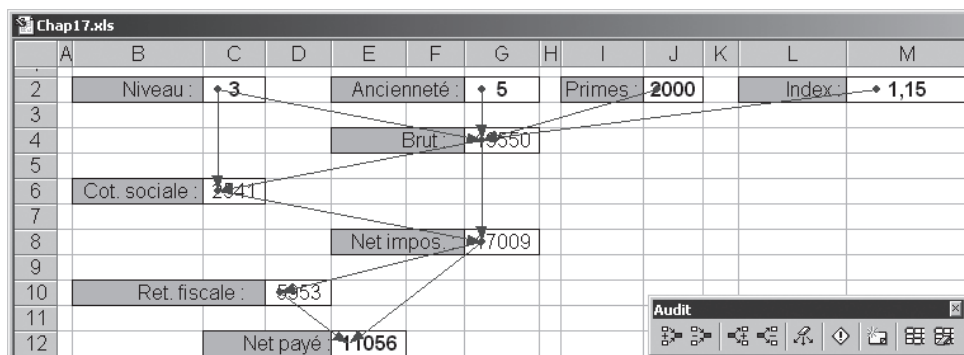


Figure 17.7 - Graphe de dépendance tel qu'affiché par EXCEL

17.8 LES VALEURS D'EXCEPTION

Nous n'avons considéré jusqu'ici que des modèles cohérents et dont les données sont disponibles et correctes. Il n'en sera pas toujours ainsi. En effet, comme toute construction humaine, un modèle peut comporter des erreurs⁴. D'autre part, certaines données peuvent être manquantes ou erronées. L'exécution d'un modèle peut donc entraîner deux problèmes :

- certaines grandeurs n'ont pas pu être calculées en raison de données manquantes,
- certaines grandeurs n'ont pas pu être calculées car l'évaluation de leur règle de définition entraîne une erreur.

3. Voir par exemple [Geoffrion, 1987].

4. Nous réexaminerons plus tard le problème de la correction d'un modèle (section 18.5).

La plupart des tableurs offrent (au moins) deux valeurs standard qui permettent de représenter chacune de ces situations. La première, que nous dénoterons par **absent**, représente le fait que la grandeur n'a pas de valeur parce que sa règle de définition n'a pu être évaluée faute de disposer des grandeurs nécessaires. La seconde, que nous appellerons **erreur**, représente le fait que la grandeur n'a pas reçu de valeur car l'évaluation de sa règle de définition a provoqué une erreur.

Ces valeurs, dites **valeurs d'exception**, vont se propager dans le modèle lors de son exécution selon les principes suivants⁵ :

- une grandeur B dont la règle de définition utilise une donnée A sans valeur prend la valeur *absent*,
- une grandeur B dont la règle de définition utilise une grandeur A à laquelle la valeur *absent* a été assignée prend également la valeur *absent*,
- une grandeur B dont la règle de définition utilise une grandeur A à laquelle la valeur *erreur* a été assignée prend également la valeur *erreur*,
- en cas d'ambiguïté, la valeur *erreur* a priorité sur la valeur *absent*.

Lorsque le modèle ne comporte pas de règles absorbantes (voir ci-dessous), la propagation de ces valeurs ainsi que des données manquantes est entièrement déterminée par le graphe de dépendance, ainsi que l'illustre l'exemple de la figure 17.8.

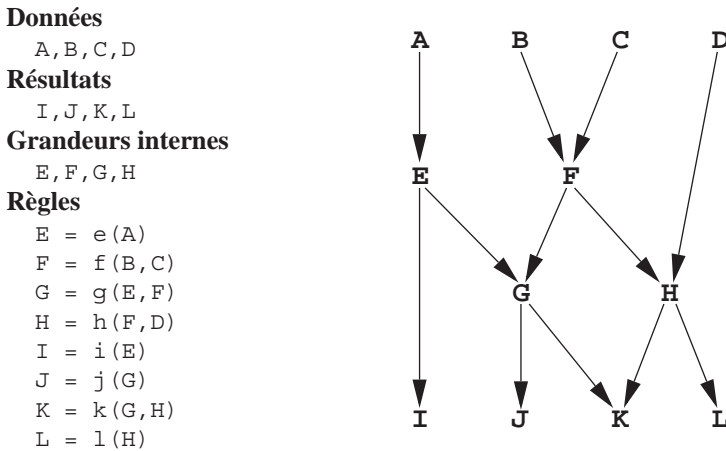


Figure 17.8 - Un modèle et son graphe de dépendance

On en déduit par exemple que

- si la donnée D est manquante, alors H, K et L prendront la valeur *absent*,
- si E a la valeur *erreur*, alors G, I, J et K prennent également la valeur *erreur*,
- si F a la valeur *erreur* et si la donnée D est manquante, H prend la valeur *erreur*,

5. On rapprochera ces règles de celles de la propagation de la valeur null en SQL (section 6.10).

- si K a la valeur *absent*, alors G ou H (ou les deux) a la valeur *absent*,
- si L a la valeur *erreur*, alors la fonction f ou la fonction h ont provoqué une erreur.

Ces principes de propagation des valeurs d'exception sont valables pour autant que les règles de définition des grandeurs concernées n'absorbent pas ces valeurs. En effet, la partie conditionnelle d'une définition multiple, ou la définition d'une grandeur logique, peuvent les transformer en valeurs normales, comme l'illustre la définition ci-dessous. Une telle règle **absorbe** l'éventuelle valeur d'exception *absent* qui lui serait communiquée *via* la grandeur DURÉE. Cette valeur d'exception ne sera pas transmise à PRIME, mais sera transformée en valeur normale, ici 0.

```
PRIME = 0 si DUREE = absent
        10 si DUREE < 5
        20 si DUREE ≥ 5
```

La propagation des valeurs d'exception dans les expressions logiques est régie par des lois particulières. En effet, les opérateurs *et*, *ou* et *non* doivent être définis, non plus sur un ensemble de deux valeurs {VRAI, FAUX}, mais sur un ensemble plus étendu {VRAI, FAUX, *absent*, *erreur*}. Ces raisonnements relèvent de ce qu'on appelle les logiques non standards⁶, et dépassent le cadre d'un ouvrage comme celui-ci.

17.9 GRANDEURS ET MODÈLES DIMENSIONNÉS

Il est possible de constituer la description d'un état déterminé du domaine d'application, par exemple le budget d'un département à une date déterminée, en assignant une valeur à chaque grandeur qui décrit une caractéristique du domaine. Dans ce cas, chaque grandeur possède une **valeur**. On peut aussi envisager de décrire des états successifs du domaine (ce même budget pendant 12 mois), ou d'un ensemble d'éléments de même nature (les budgets de tous les départements). Il existe alors une ou plusieurs grandeurs pour lesquelles nous observons non pas une valeur, mais une **suite de valeurs**.

Considérons par exemple le domaine des traitements du personnel dans une entreprise. Nous avons établi à la figure 17.1 un modèle qui décrit l'état d'un employé pour un mois déterminé. Fixer ou calculer une valeur pour chaque concept représente l'état du traitement d'une personne pour un mois déterminé.

Admettons qu'on désire décrire le domaine dans son évolution temporelle. En d'autres termes, et selon une échelle adéquate, on se propose de prendre en compte la dimension du temps, qu'on représentera par la grandeur T. Il serait aisé de généraliser l'approche suivie jusqu'à présent en traitant cette nouvelle grandeur comme les autres et en ajoutant de nouvelles règles telles que la suivante :

$$\text{INDEX} = \text{I}(\text{T})$$

6. Comme la logique ternaire de SQL (section 6.10).

où I est une fonction délivrant la valeur de l'index au mois T .

Cependant, la spécificité du traitement et surtout de l'interprétation de ce type de **grandeurs multivaluées** (une telle grandeur peut être considérée comme une **dimension** du problème) suggère une représentation qui leur soit propre. On proposera par conséquent la notation indicée illustrée dans la figure 17.9, dans laquelle on considère que T désigne les mois de t_1 à t_n , selon une numérotation quelconque. On y a considéré de nouvelles grandeurs qui sont le *total des cotisations de l'employé* (TOTAL_COTISATION_E) et le *total des retenues fiscales de l'employé* (TOTAL_RETENUE_E). Celles-ci ne dépendent plus de T .

Données

$T = t_1..t_n$
 ANCIENNETE_T, NIVEAU_T, PRIMEST, INDEX_T

Résultats

BRUT_T, COTISATION_SOCIALE_T, NET_IMPOSABLE_T,
 RETENUE_FISCALE_T, NET_PAYE_T
 TOTAL_RETENUES_E, TOTAL_COTISATIONS_E

Règles

$BRUT_T = (br(ANCIENNETE_T, NIVEAU_T) + PRIMEST) * INDEX_T$
 $COTISATION_SOCIALE_T = (0,1 + NIVEAU_T/100) * BRUT_T$
 $NET_IMPOSABLE_T = BRUT_T - COTISATION_SOCIALE_T$
 $RETENUE_FISCALE_T = NET_IMPOSABLE_T * (0,1 + \min(0,4; NET_IMPOSABLE_T/50.000))$
 $NET_PAYE_T = NET_IMPOSABLE_T - RETENUE_FISCALE_T$
 $TOTAL_RETENUE_E = \sum_T RETENUE_FISCALE_T$
 $TOTAL_COTISATION_E = \sum_T COTISATION_SOCIALE_T$
 $2 \leq NIVEAU_T \leq 12$

Figure 17.9 - Un modèle de calcul du traitement d'un employé selon le mois

Il faut bien comprendre que la notation synthétique

$$X_T = f(Y_T)$$

est mise pour

$$X_{t_i} = f(Y_{t_i}) \quad (i = 1..n)$$

et représente donc en réalité les n règles similaires

$$\begin{aligned} X_{t_1} &= f(Y_{t_1}) \\ X_{t_2} &= f(Y_{t_1}) \\ &\dots \\ X_{t_{n-1}} &= f(Y_{t_{n-1}}) \\ X_{t_n} &= f(Y_{t_n}) \end{aligned}$$

Dans un modèle tel que celui de la figure 17.9, l'expression

$$\text{TOTAL_RETENUE_E} = \sum_T \text{RETENUE_FISCALE}_T$$

est mise pour

$$\text{TOTAL_RETENUE_E} = \sum_{i=1..n} \text{RETENUE_FISCALE}_{t_i}$$

Afin de bien comprendre la structure de ce modèle, il faut se rendre compte qu'une grandeur telle que PRIMES_T représente en fait les n grandeurs PRIMES_{t_1} , ..., PRIMES_{t_n} , et que la règle

$$\text{NET_PAYE}_T = \text{NET_IMPOSABLE}_T - \text{RETENUE_FISCALE}_T$$

désigne en réalité les n règles

$$\begin{aligned} \text{NET_PAYE}_{t_1} &= \text{NET_IMPOSABLE}_{t_1} - \text{RETENUE_FISCALE}_{t_1} \\ &\dots \\ \text{NET_PAYE}_{t_n} &= \text{NET_IMPOSABLE}_{t_n} - \text{RETENUE_FISCALE}_{t_n} \end{aligned}$$

Considérons à présent que le domaine s'étend à *tous les employés* de l'entreprise. En traitant cet aspect non pas comme le simple ajout d'une grandeur, mais comme une dimension supplémentaire, on obtient un modèle tel que celui qui est présenté à la figure 17.10. La grandeur E représente les employés (ses valeurs sont par exemple des entiers dont chacun désigne un employé) et est traitée comme l'était T dans le modèle de la figure 17.9. Nous avons aussi considéré de nouvelles grandeurs qui sont le *total des cotisations par mois* ($\text{TOTAL_COTISATION_M}_T$), et le *total de toutes les cotisations* (TOTAL_COTISATION).

17.10 LES FONCTIONS AGRÉGATIVES

La fonction Σ rencontrée dans le modèle 17.10 est dite **agrégative** car elle renvoie une valeur *agrégée*, calculée sur un ensemble de valeurs de taille quelconque, généralement spécifié comme une grandeur multivaluée. Il en existe d'autres, définies pour toute grandeur (ou expression) G numérique, et pour toute dimension T non vide (contenant au moins une valeur). On peut considérer la liste suivante comme représentative :

- $\Sigma_T(G_T)$: renvoie la somme des G_T ; est également notée $\sum_T G_T$
- $\Pi_T(G_T)$: renvoie le produit des G_T ; est également notée $\prod_T G_T$
- $\min_T(G_T)$: renvoie la valeur minimale des G_T ;
- $\max_T(G_T)$: renvoie la valeur maximale des G_T ;
- $\text{moy}_T(G_T)$: renvoie la valeur moyenne des G_T ;

Si la grandeur (ou expression) G est booléenne, et pour toute dimension T non vide, on utilisera aussi les fonctions

- $\text{ou}_T(G_T)$: renvoie la somme logique (disjonction) des G_T ;
- $\text{et}_T(G_T)$: renvoie le produit logique (conjonction) des G_T .

Données

$T = t_1..t_n$
 $E = e_1..e_m$
 $ANCIENNETE_{T,E}, NIVEAU_{T,E}, PRIMES_{T,E}, INDEX_T$

Résultats

$BRUT_{T,E}, COTISATION_SOCIALE_{T,E}, NET_IMPOSABLE_{T,E}$
 $RETENUE_FISCALE_{T,E}, NET_PAYE_{T,E}$
 $TOTAL_RETENUE_E_E, TOTAL_COTISATION_E_E$
 $TOTAL_COTISATION_M_T, TOTAL_COTISATION$

Règles

$BRUT_{T,E} = (br(ANCIENNETE_{T,E}, NIVEAU_{T,E}) + PRIMES_{T,E}) * INDEX_T$
 $COTISATION_SOCIALE_{T,E} = (0, 1 + NIVEAU_{T,E}/100) * BRUT_{T,E}$
 $NET_IMPOSABLE_{T,E} = BRUT_{T,E} - COTISATION_SOCIALE_{T,E}$
 $RETENUE_FISCALE_{T,E} = NET_IMPOSABLE_{T,E}$
 $\quad * (0, 1 + \min(0, 4; NET_IMPOSABLE_{T,E}/50000))$
 $NET_PAYE_{T,E} = NET_IMPOSABLE_{T,E} - RETENUE_FISCALE_{T,E}$
 $TOTAL_RETENUE_E_E = \sum_T RETENUE_FISCALE_{T,E}$
 $TOTAL_COTISATION_E_E = \sum_T COTISATION_SOCIALE_{T,E}$
 $TOTAL_COTISATION_M_T = \sum_E COTISATION_SOCIALE_{T,E}$
 $TOTAL_COTISATION = \sum_T TOTAL_COTISATION_M_T$
 $2 \leq NIVEAU_{T,E} \leq 12$

Figure 17.10 - Un modèle de calcul du traitement des employés selon le mois

Enfin, la fonction suivante est définie pour toute grandeur T :

- $nombre(T)$: renvoie le nombre de valeurs de la dimension T .

On mentionnera encore certaines fonctions correspondant à des opérateurs vectoriels ou matriciels, généralement disponibles dans les tableurs.

Utilisées dans des règles de définition ou des contraintes, les fonctions agrégatives ont une propriété remarquable, qui est d'*absorber* la dimension sur laquelle elles sont définies. En effet, le résultat d'une telle fonction perd cette dimension. Par exemple, l'expression

$$\sum_E COTISATION_SOCIALE_{T,E}$$

est dimensionnée selon les grandeurs $\{T, E\} - \{E\}$, c'est-à-dire selon $\{T\}$. De la même manière, l'expression.

$$\sum_T TOTAL_COTISATION_M_T$$

est sans dimension⁷, c'est-à-dire monovaluée.

Cette règle d'absorption, comme nous le verrons en 18.5.5, permet de vérifier la cohérence des dimensions dans des expressions complexes.

7. Attention, cela ne veut pas dire qu'elle est sans unité.

La plupart de ces fonctions peuvent être définies par une formule de récurrence. Par exemple, la fonction $\text{ou}_T(G_T)$ est définie comme étant égale à résultat_{t_n} dans le fragment de modèle suivant, qui ne fait appel qu'à l'opérateur binaire ou :

```

résultatt1 = Gt1
résultatti = résultatti-1 ou Gti (i = 2..n)

```

Cette remarque prendra toute son importance lors de la traduction d'un modèle abstrait dans une feuille de calcul lorsque le tableur n'offre pas en standard la fonction agrégative utilisée.

17.11 RÈGLES DE RÉCURRENCE ET RÉCURSIVITÉ

Il existe une forme particulière d'expression appelée **règle de récurrence**. Une telle règle concerne une grandeur indicée, c'est-à-dire fonction d'une autre grandeur jouant le rôle de **dimension**. Cette grandeur indicée se retrouve à la fois en partie gauche et en partie droite de cette règle. Une règle de récurrence exprime que la valeur de la grandeur est fonction de sa valeur (ou ses valeurs) précédente(s), ou même suivante(s), selon la dimension considérée. Prenons l'exemple de la figure 17.11. On y décrit le suivi d'un poste budgétaire, qui dispose d'une somme initiale (BUDGET) et fait l'objet de dépenses mensuelles (DEPENSE_{t_i}). On désire observer l'évolution du montant disponible au début de chaque mois pour ce poste (BUDGET-DISP_{t_i}).

Données

```

T = t1..tn
BUDGET
DEPENSET

```

Résultats

```

BUDGET-DISPT
SOLDE

```

Règles

```

BUDGET-DISPt1 = BUDGET
BUDGET-DISPti = BUDGET-DISPti-1 - DEPENSEti-1 (i = 2..n)
SOLDE = BUDGET-DISPtn - DEPENSEtn

```

Figure 17.11 - Règles de récurrence - Contrôle d'un poste budgétaire

On détermine que le montant disponible au début d'un mois est le montant disponible au début du mois précédent duquel on retire le montant des dépenses du mois précédent. Cette règle n'est pas déterminée pour le premier mois. Il faut donc dans ce cas donner une formule spéciale, qui détermine la valeur initiale de la suite des valeurs. On ajoute la grandeur SOLDE qui représente le solde à la fin du dernier mois.

On résistera à la tentation de simplifier le modèle par l'introduction de grandeurs fictives telles que $BUDGET-DISP_{t_0}$ et $DEPENSE_{t_0}$, qui permettraient de traiter de façon uniforme toutes les valeurs réelles de $BUDGET-DISP_T$. Si la deuxième règle peut alors s'écrire de manière plus régulière,

$$BUDGET-DISP_{t_i} = BUDGET-DISP_{t_{i-1}} - DEPENSE_{t_{i-1}} \quad (i = 1..n)$$

il n'en reste pas moins que les grandeurs introduites n'ont aucune signification et rendent donc le modèle plus complexe et moins lisible. Cette pratique s'inspire d'ailleurs d'une technique, souvent douteuse, utilisée en programmation algorithmique. On l'évitera donc.

Une autre forme d'expression importante est celle des **règles récursives**. On entend par là une règle qui exprime qu'une grandeur dépend d'elle-même, directement ou non. Donnons un exemple concret.

Une entreprise fabrique un type d'appareils qu'elle désire vendre. Le prix de fabrication dépend de la quantité fabriquée. En outre, on est certain de vendre la quantité qu'on aura fabriquée. La quantité vendue dépend évidemment du prix de vente, lequel est fixé à 150 % du prix de fabrication.

D'où le modèle ci-dessous :

$$\begin{aligned} \text{PRIX-F} &= f(\text{QUANTITE-F}, \dots) \\ \text{QUANTITE-F} &= \text{QUANTITE-V} \\ \text{QUANTITE-V} &= g(\text{PRIX-V}, \dots) \\ \text{PRIX-V} &= 1,5 * \text{PRIX-F} \end{aligned}$$

On y observe que PRIX-F dépend de QUANTITE-F , qui dépend de QUANTITE-V , qui dépend de PRIX-V , qui dépend lui-même de PRIX-F ! Donc, chaque grandeur dépend indirectement d'elle-même. Ce modèle est dit **récursif**. La propriété de dépendance cyclique s'observe plus clairement encore dans le graphe de dépendance qui lui correspond (figure 17.12).

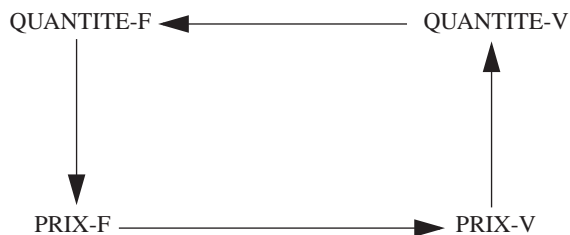


Figure 17.12 - Circuit dans un graphe de dépendance

On peut mieux mettre cette propriété en évidence en redéfinissant les fonctions f et g , de manière à n'expliciter que les grandeurs liées à la vente :

$$\begin{aligned} \text{PRIX-V} &= f'(\text{QUANTITE-V}, \dots) \\ \text{QUANTITE-V} &= g'(\text{PRIX-V}, \dots) \end{aligned}$$

L'exécution d'un tel modèle consiste généralement à rechercher (s'il en existe) des valeurs de PRIX et de QUANTITE qui valident les règles ci-dessus, c'est-à-dire une solution du système d'équations, éventuellement non linéaires, que constituent ces relations.

Remarque

On pourrait suggérer *subtilement* qu'il suffirait de réécrire une des règles $X = f(Y)$ sous la forme $Y = g(X)$ pour faire disparaître le circuit ! Il n'en est malheureusement rien, car alors la grandeur Y du modèle y serait définie deux fois, ce qui sera explicitement exclu au chapitre 18.

Il existe une autre catégorie de modèles qui n'admettent pas une (ou plusieurs) solution(s) comme nous venons de le voir, mais qui décrivent des **systèmes en évolution**. Ils font appel à des règles plus complexes, dans lesquelles on fait intervenir non seulement les valeurs des grandeurs, mais également leur **vitesse d'évolution**. Ces règles sont le plus souvent des équations différentielles, qu'on traduira sous une forme propre au calcul, et en particulier à la prise en charge par un tableur, celle d'*équations aux différences finies*.

La résolution d'un ensemble de règles récursives n'est pas chose aisée en général. Elle correspond à la résolution d'un système d'équations, problème qui n'est pas toujours à la portée immédiate des tableurs classiques. On peut cependant tirer profit du fonctionnement naturel des tableurs pour mettre en œuvre une variante simple des techniques de résolution itérative dites par approximations successives, ou par *relaxation*.

On considère d'abord que, du point de vue du calcul, des règles récursives peuvent être développées sous la forme d'un ensemble de règles de récurrence. Ainsi, les règles

$$\begin{aligned} A &= f(B) \\ B &= g(A) \end{aligned}$$

peuvent-elles être *dépliées* selon les relations suivantes

$$\begin{aligned} B_1 &= \text{valeur initiale arbitraire} \\ A_1 &= f(B_1) \\ B_2 &= g(A_1) \\ A_2 &= f(B_2) \\ B_3 &= g(A_2) \\ A_3 &= f(B_3) \\ &\vdots \\ A_{n-1} &= f(B_{n-1}) \\ B_n &= g(A_{n-1}) \\ A_n &= f(B_n) \end{aligned}$$

selon le motif général

$$\begin{aligned} B_i &= g(A_{i-1}) \\ A_i &= f(B_i) \end{aligned}$$

Le nombre **n** peut être fixé d'avance, comme dans un modèle décrivant une évolution qu'on veut suivre durant **n** périodes. Il peut aussi être défini par une **condition de convergence** qui indique qu'on a trouvé des valeurs validant les règles qui les définissent. Il s'agit donc d'un procédé de calcul par approximations successives. La *valeur initiale arbitraire* est une première valeur choisie comme une solution plausible. Le critère d'arrêt est la stabilisation des valeurs obtenues. En principe, on cherche une valeur de A telle que

$$A = f(g(A))$$

ou, pratiquement, telle que

$$A_i \approx A_{i-1}$$

ou encore, si ε est un nombre positif arbitrairement petit,

$$|A_i - A_{i-1}| \leq \varepsilon$$

Le mode opératoire des tableurs modernes permet cependant une expression condensée de ces règles de récurrence. La formulation semble récursive.

$$A = f(B)$$

$$B = g(A)$$

Ce modèle sera cependant traité de manière **itérative**, soit par activation manuelle (commande explicite de chaque itération), soit par contrôle automatique en mode de calcul *itératif*, étant donné une valeur de ε . Chaque itération consiste à calculer la valeur de A, puis la valeur de B, ce processus étant reconduit autant de fois que nécessaire.

Considérons un exemple élémentaire. Le montant net N correspondant à un revenu peut se calculer comme suit : la retenue fiscale R est calculée comme une proportion p, non pas du montant brut B, mais du net N :

$$R = p * N$$

$$N = B - R$$

Il est clair que ce modèle linéaire pourrait se simplifier par élimination de R, de sorte que les règles ne soient plus récursives. On obtiendrait immédiatement $N = 833,3333...$ et $R = 166,6666...$. Nous conserverons cependant le modèle dans cet état aux fins de la démonstration.

Supposons que R soit évaluée avant N et posons $p = 0,2$ et $B = 1000$. Avant évaluation, N n'a pas encore de valeur, ce qui peut être assimilé à 0, *valeur initiale arbitraire* typique. Après la première évaluation, R vaut 0 (puisque N vaut 0) puis N est évalué, ce qui lui donne la valeur 1000. Si on exécute à nouveau le modèle, R vaut alors 200 et N 800. En poursuivant les évaluations de manière itérative, on obtient les valeurs indiquées dans le tableau ci-dessous :

On observe que les résultats convergent rapidement vers les valeurs exactes. L'arrêt des calculs sera déterminé soit par un nombre fixe d'itérations (on pourrait considérer que 8 est un nombre suffisant), soit par une condition de convergence qui

serait par exemple que la différence entre deux valeurs successives d'une grandeur tombe au-dessous d'une *valeur de consigne* ($\epsilon = 0,01$ serait raisonnable ici).

Evaluation	$R = 0,2 * N$	$N = 1000 - R$
avant	-	0
1	0	1000
2	200	800
3	160	840
4	168	832
5	166,4	833,6
6	166,72	833,28
7	166,656	833,344
8	166,6688	833,3312

Remarque importante

Le mode de résolution itératif de règles récursives est induit naturellement par les fonctions des tableurs. Il doit cependant être considéré comme très rudimentaire par rapport aux techniques classiques de résolution numérique de systèmes d'équations. Il peut en effet présenter des défauts importants comme une convergence lente, ou même une absence de convergence : le calcul s'éloigne de toute solution, alors que le système en admet au moins une. Pour illustrer ce problème, il suffit de considérer à nouveau l'exemple ci-dessus, mais reformulé comme suit :

$$\begin{aligned} N &= R / p \\ R &= B - N \end{aligned}$$

soit ici

$$\begin{aligned} N &= 5 * R \\ R &= 1000 - N \end{aligned}$$

Il apparaît immédiatement que la résolution itérative diverge très rapidement et échoue à trouver la solution :

Evaluation	$R = 1000 - N$	$N = 5 * R$
avant	-	0
1	1000	5000
2	-4000	-20000
3	21000	105000
4	-104000	-520000
5	521000	...

Cette technique conduit à une solution pour certains problèmes, formulés d'une manière favorable, et pour certaines valeurs de démarrage. Il existe d'autres techniques plus sûres et plus efficaces, mais dont l'étude dépasserait le cadre de cet ouvrage. On citera par exemple les techniques de programmation linéaire destinées à la résolution de systèmes d'équations linéaires sous contraintes, les algorithmes de

Newton-Raphson, les algorithmes du gradient et leur dérivés (gradient conjugué, Davidon, etc.) pour les systèmes non linéaires. Certains tableurs offrent des processeurs de résolution basés sur de telles techniques (le *solveur* d'EXCEL par exemple). On pourra renvoyer le lecteur aux ouvrages spécialisés ([Dion, 1999] par exemple). Il était cependant important d'attirer l'attention du lecteur sur les limites du procédé évoqué.

17.12 SOUS-MODÈLES ET MODULARISATION

Un problème complexe peut souvent se décomposer en sous-problèmes plus simples à résoudre. La résolution du problème initial se simplifie dès que les sous-problèmes sont eux-mêmes résolus. Un sous-problème peut avoir été déjà résolu, auquel cas on renvoie simplement à sa solution. Sa résolution peut aussi être remise à plus tard, auquel cas on poursuit la résolution du problème initial comme si ce sous-problème était résolu.

Ces principes sont à la base de nombreux concepts de programmation tels que la *programmation modulaire*, les *types abstraits*, l'*architecture client-serveur*, l'*approche orientée objet* et les *composants réutilisables* en programmation procédurale. Parmi les motivations qui sont à la base de ces approches on peut citer les suivantes :

- la solution d'un sous-problème peut être élaborée indépendamment du problème global,
- cette solution doit pouvoir être réutilisée ailleurs et plus tard, pour d'autres problèmes globaux,
- il est possible de réviser la solution du sous-problème sans toucher à celle du problème global; cette dernière est donc aussi plus stable vis-à-vis de ces perturbations,
- il est possible d'autre part de réviser la solution du problème global sans toucher à celles des sous-problèmes,
- la solution du problème global (de même que celle du sous-problème) est plus simple à élaborer, à comprendre, à modifier; elle est donc aussi plus fiable⁸.

Par analogie, un modèle complexe peut renvoyer à des **sous-modèles**, qui sont des modèles déjà définis (ou à définir). Dans son utilisation comme composant d'un modèle, un sous-modèle ne nous intéresse que pour son effet externe. Ainsi, il serait intéressant de considérer la fonction suivante, extraite de la figure 17.1,

```
br (ANCIENNETE, NIVEAU)
```

comme le sous-modèle d'évaluation du traitement brut de base d'un employé en fonction de son ancienneté et de son niveau.

8. La probabilité qu'elle soit correcte est plus grande.

Modèle TRAITEMENT**Données**

ANCIENNETE, NIVEAU, PRIMES, INDEX

Résultats

BRUT_DE_BASE, BRUT, COTISATION_SOCIALE, NET_IMPOSABLE,
RETENUE-FISCALE, NET-PAYE

Règles

BRUT_DE_BASE = **calcul_BRUT**(ANCIENNETE, NIVEAU)
BRUT = (BRUT_DE_BASE + PRIMES) * INDEX
COTISATION_SOCIALE = **calcul_COTISATION**(NIVEAU, BRUT)
NET_IMPOSABLE = BRUT - COTISATION_SOCIALE
RETENUE_FISCALE = **calcul_RETENUE**(NET_IMPOSABLE)
NET_PAYE = NET_IMPOSABLE - RETENUE_FISCALE
 $2 \leq \text{NIVEAU} \leq 12$

Modèle calcul_BRUT**Données**

ANC, NIV

Résultats

BRUT_D_B

Règles

BRUT_D_B =

Modèle calcul_COTISATION**Données**

NIV, BRUT

Résultats

COT

Règles

$\text{COT} = (0,1 + \text{NIV}/100) * \text{BRUT}$

Modèle calcul_RETENUE**Données**

NET

Résultats

RET

Règles

$\text{RET} = \text{NET} * (0,1 + \min(0,4; \text{NET}/50000))$

Figure 17.13 - Structure modulaire d'un modèle

Le détail du procédé d'évaluation de ce traitement (consultation de tables, calcul, consultation d'une base de données, etc.) est supposé défini par ailleurs et seul nous intéresse ici le fait que ce modèle est disponible et que sa spécification⁹ est connue.

9. On appelle spécification d'une fonction tout ce qu'il faut connaître (mais pas plus) pour l'utiliser correctement. La spécification indique en particulier la classe de problèmes que la fonction résout, et dans quelles limites. Par exemple, les techniques spécifiques sur lesquelles cette solution est élaborée peuvent (le plus souvent) être ignorées.

Le mode de calcul peut d'ailleurs évoluer au gré des modifications apportées à la législation sociale et aux règles de gestion de l'organisation. Ces modifications n'ont aucun impact sur le modèle 17.1, puisqu'elles sont **cachées** dans le sous-modèle.

Modèle TRAITEMENT

Données

$T = t_1..t_n$
 ANCIENNETE_T, NIVEAU_T, PRIMES_T, INDEX_T

Résultats

BRUT_DE_BASE_T, BRUT_T, COTISATION_SOCIALE_T,
 NET_IMPOSABLE_T,
 RETENUE_FISCALE_T, NET_PAYE_T
 TOTAL_RETENUES_E, TOTAL_COTISATIONS_E

Règles

BRUT_DE_BASE_T = **calcul_BRUT**(ANCIENNETE_T, NIVEAU_T)
 BRUT_T = (BRUT_DE_BASE_T + PRIMES_T) * INDEX_T
 COTISATION_SOCIALE_T = **calcul_COTISATION**(NIVEAU_T, BRUT_T)
 NET_IMPOSABLE_T = BRUT_T - COTISATION_SOCIALE_T
 RETENUE_FISCALE_T = **calcul_RETENUE**(NET_IMPOSABLE_T)
 NET_PAYE_T = NET_IMPOSABLE_T - RETENUE_FISCALE_T
 TOTAL_RETENUE_E = \sum_T RETENUE_FISCALE_T
 TOTAL_COTISATION_E = \sum_T COTISATION_SOCIALE_T
 $2 \leq \text{NIVEAU}_T \leq 12$

Modèle calcul_BRUT

Données

ANC, NIV

Résultats

BRUT_D_B

Règles

BRUT_D_B =

Modèle calcul_COTISATION

Données

NIV, BRUT

Résultats

COT

Règles

COT = $(0, 1 + \text{NIV}/100) * \text{BRUT}$

Modèle calcul_RETENUE

Données

NET

Résultats

RET

Règles

RET = $\text{NET} * (0, 1 + \min(0, 4; \text{NET}/50000))$

Figure 17.14 - Structure modulaire d'un modèle dimensionné

Afin d'illustrer la structure de sous-modèle, on reprend le modèle de la figure 17.1,

mais en cachant non seulement le calcul du brut de base, mais aussi celui de la cotisation sociale et de la retenue fiscale, calculs qui sont susceptibles d'évoluer dans le temps. On peut alors proposer le modèle 17.13.

En guise d'illustration de la réutilisation des sous-modèles, on reformulera le modèle 17.9 sous la forme du modèle 17.14. On observe que les sous-modèles y sont intégralement repris, sans aucune modification. En particulier, la dimension temporelle du modèle, représentée par la grandeur T, apparaît dans le modèle principal TRAITEMENT, mais non dans les sous-modèles.

Un sous-modèle peut encore être utilisé pour cacher le détail d'un calcul technique sans rapport direct avec le domaine d'application. La figure 17.15 montre comment l'absence d'une fonction OU agrégative peut être palliée par la définition d'un sous-modèle OU.

Modèle PRINCIPAL

Données

$T = t_1..t_n$
 OK_T, \dots

Résultats

ACCEPTTE, ...

Règles

...
 $ACCEPTTE = OU(T, OK_T)$
 ...

Modèle OU

Données

$I = i_1..i_m$
 B_I

Grandeurs internes

resultat_I

Résultats

R

Règles

$R = resultat_{i_m}$
 $resultat_{i_1} = B_{i_1}$
 $resultat_{ij} = resultat_{ij-i} \text{ ou } B_{ij} \quad (j = 2..m)$

Figure 17.15 - Simulation de la fonction agrégative OU par un sous-modèle

La définition et l'invocation de sous-modèles, telles qu'elles ont été illustrées ci-dessus, méritent quelques précisions.

- La définition d'un sous-modèle comporte une section **Données**, cependant facultative, qui énumère la ou les grandeurs qui lui seront communiquées par le modèle invoquant. Le nom de ces grandeurs est local et indépendant des noms des grandeurs du modèle invoquant.
- Elle comporte une section **Résultat**, obligatoire, qui énumère la ou les grandeurs

- que le sous-modèle évalue pour le compte du modèle qui l'invoque. Le nom de ces grandeurs est local et indépendant des noms des grandeurs du modèle invoquant.
- Elle comporte une section **Grandeurs internes**, facultative, fonction des besoins internes du sous-modèle. Ici encore, les noms sont indépendants de ceux du modèle invoquant.

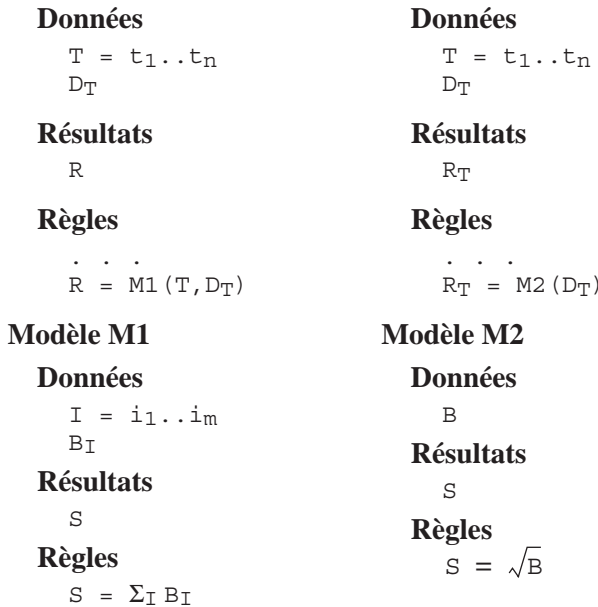


Figure 17.16 - Sous-modèles et grandeurs dimensionnées

- L'invocation d'un sous-modèle par le modèle invoquant est identique à l'appel d'une fonction ordinaire. Afin de simplifier le format des règles, l'invocation d'un sous-modèle de nom M prend l'aspect d'une règle, comme illustré ci-dessous :

$$R = M(D1, D2, \dots, Dn)$$

Les arguments (données) transmis sont cités dans l'ordre de leur déclaration dans la section **Données** du sous-modèle. Les résultats sont assignés à la grandeur R du modèle invoquant. Si plus d'un résultat est attendu, la liste en est spécifiée de la manière suivante :

$$\{R1, R2, \dots, Rm\} = M(D1, D2, \dots, Dn)$$

- L'ordre de citation des résultats en partie gauche de cette règle est celui de leur déclaration dans la section **Résultat** du sous-modèle.
- Un sous-modèle peut opérer sur (c'est-à-dire connaît) des grandeurs sans dimension, ou sur des grandeurs dimensionnées. Dans ce dernier cas, la (les) dimension doit être citée dans la section **Données** du sous-modèle. Les deux modèles de la figure 17.16 illustrent chacun une de ces situations.

Chapitre 18

Conception d'un modèle

On présente une démarche de construction d'un modèle abstrait. Cette démarche consiste à exprimer les résultats en fonction soit des données, soit de grandeurs internes pertinentes qu'on met en évidence, puis à définir ces dernières comme on l'a fait des résultats. On étudie les diverses règles de cohérence d'un modèle. On aborde enfin le problème des fragments de modèles qui ne s'expriment pas naturellement par des règles directionnelles.

18.1 DÉMARCHE DE CONCEPTION D'UN MODÈLE

Ainsi que nous l'avons précisé dans le chapitre 16, nous distinguerons le problème de la **conception** du modèle sous la forme d'une expression abstraite, de celui de sa **traduction** pour un tableur.

La **conception du modèle** consiste à relever les données, les résultats, puis les règles qui permettent d'établir ces résultats à partir des données. Pratiquement, et à supposer que le domaine à modéliser soit bien connu, on tombera invariablement sur deux difficultés.

La première est d'établir *a priori* l'ensemble strictement nécessaire et suffisant des **données**.

La seconde difficulté est de ne retenir que les relations strictement nécessaires et suffisantes conduisant aux résultats à partir des données. En cherchant à atteindre un point au travers d'un réseau de chemins, le danger est grand de parcourir une route trop longue, de s'égarer dans la campagne ou de s'éloigner de la destination qu'on s'est fixée¹.

Lorsque le modèle est construit, on procède à sa **normalisation**, qui consiste à éliminer des maladrotes structures, et à sa **validation**, qui consiste à rechercher et à corriger divers types d'erreurs.

18.2 LES PRINCIPES

La démarche qu'on va suivre est basée sur quelques principes relativement intuitifs.

Le *premier principe* consiste à partir des résultats pour remonter vers les données. Il est plus sûr en effet de se demander comment calculer un résultat que d'imaginer ce qu'on pourrait bien calculer d'utile (c'est-à-dire qui nous rapproche des résultats) à partir des données dont on dispose et qui ne sont encore que partiellement connues.

Le *deuxième principe* consiste à n'adopter que des règles très simples, qu'on pourra facilement élaborer, justifier et expliquer. Cela nous amène en général à définir des grandeurs nouvelles, caractéristiques du domaine analysé, mais qui n'avaient pas été relevées en tant que données ou résultats. Ces grandeurs sont par conséquent du type interne (sauf grandeurs relevant du principe ci-dessous, qu'on reconnaîtra comme données).

Le *troisième principe* est d'élaborer, parallèlement à l'ensemble des règles, la liste des données. On partira d'un ensemble initial de données qui paraissent utiles lors d'une analyse rapide du domaine. Cet ensemble sera complété par les grandeurs mises en évidence lors de l'élaboration des règles, et qu'on juge devoir faire partie des données. En revanche, les données qui n'ont été utilisées dans aucune règle seront retirées de l'ensemble.

Le *quatrième principe* définit la terminaison du processus. L'analyse s'arrête lorsque, dans les règles déjà construites, chaque grandeur invoquée en partie droite est soit une donnée, soit une grandeur qui apparaît en partie gauche d'une règle.

Note méthodologique

Le lecteur attentif observera que, malgré la forte analogie qui existe entre le problème de l'expression abstraite d'une base de données que constitue son schéma conceptuel et celui de l'expression d'un modèle de calcul, les procédures proposées divergent considérablement. Les raisons de cette divergence sont faciles à expliquer.

Il serait tentant en effet d'adopter une démarche similaire, qui consiste à analyser des sources réelles telles que des textes en langage courant, proposition par proposition, et d'en exprimer formellement le contenu sous la forme de grandeurs et de règles, tout comme on exprimait, dans le chapitre 10, le contenu des textes analysés sous la forme de types d'entités, d'attributs et de types d'associations.

Cependant, la pratique d'une telle analyse conduit souvent à des modèles intrinsèquement *non directionnels*. Or de tels modèles ne conviennent pas aux tableurs actuels, qui exigent une forme directionnelle². L'expression qu'on obtiendrait à la

1. Ce langage *navigationnel* met en lumière le fait que les relations établissent des **chemins** des données vers les résultats, ce qui est intuitivement suggéré par le graphe des dépendances.

suite d'une telle analyse devrait dès lors être traduite en format directionnel. Ce processus de traduction serait loin d'être trivial et introduirait une difficulté théorique et pratique incompatible avec les objectifs de cet ouvrage³. D'autre part, la validation d'un modèle directionnel n'est pas toujours chose aisée dès que ce dernier décrit un domaine d'application complexe, comme en témoignera la section 18.5. On conçoit sans difficulté que la validation d'un modèle non directionnel est plus malaisée encore. On consultera pour s'en convaincre [Konopasec, 1984], qui suggère quelques pistes dans cette direction.

18.3 LA DÉMARCHE

On propose une démarche en trois phases : analyse, normalisation et validation.

La phase d'**analyse** consiste à construire un modèle en partant des résultats et en remontant vers les données.

La **normalisation** consiste à restructurer ce modèle de manière à le rendre plus lisible, plus simple, moins redondant, plus aisé à modifier.

La **validation** consiste à vérifier si le modèle satisfait certaines règles de cohérence et s'il passe certains tests d'exécution sur des jeux de données représentatifs.

18.3.1 Analyse

La démarche se déduit immédiatement des principes proposés en 18.2. On en donnera les étapes principales avant de traiter un exemple.

a) Étape 1

- On identifie les dimensions du problème.
- On repère les grandeurs du type **résultat**. On en donne une définition précise en langage courant. On spécifie leur type de valeurs et leur unité de mesure éventuelle.
- On repère le maximum de grandeurs du type **données**. On en donne une définition précise en langage courant et on spécifie leur type de valeurs et leur unité de mesure éventuelle.
- On détermine pour chaque grandeur ses dimensions éventuelles.

b) Étape 2

- Tant qu'il existe un résultat ou une grandeur interne non encore défini (on dira *non encore expliqué*) par une règle, on cherchera une règle simple qui le définisse.

2. On ne peut considérer aujourd'hui que les fonctions de résolution d'équations (par exemple le solveur d'Excel) constituent véritablement des outils complets et autonomes.

3. On examinera par exemple l'exercice 18.7, qui s'exprime naturellement selon un modèle non directionnel mais qui, malgré son extrême simplicité, pose quelque difficulté dans sa traduction en un modèle directionnel.

- Cette règle fait appel à des grandeurs qui sont soit déjà connues (données, internes ou même résultats), soit non encore répertoriées. Dans ce dernier cas, on se pose la question de savoir si une telle grandeur ne serait pas une donnée non encore repérée, auquel cas on l'ajoute à leur ensemble. S'il ne s'agit pas d'une donnée, on ajoute la grandeur à l'ensemble des grandeurs internes.
- Dans tous les cas, on détermine si elle est dimensionnée, on en donne une définition précise en langage courant et on spécifie son type de valeurs et son unité de mesure éventuelle.
- Cette phase est terminée lorsque toute grandeur soit est une donnée, soit a été expliquée par une règle.

Suggestion pratique

Dans une règle, on souligne le nom de toute grandeur intervenant en partie droite et qui est une donnée; en outre, quand on a trouvé une règle définissant une grandeur interne ou un résultat, on souligne son nom partout où il apparaît en partie droite des règles déjà définies. La phase est terminée lorsque tous les noms apparaissant en partie droite sont soulignés et que tous les résultats sont expliqués.

c) Étape 3

- On établit les conditions de validité du modèle en précisant les **règles de contrainte** s'appliquant aux données, ainsi qu'à certaines des grandeurs dérivables. On peut ainsi être amené à définir des grandeurs internes, qui seront alors traitées comme indiqué dans la phase 2.

d) Étape 4

- On élimine les données qui n'ont pas été utilisées dans au moins une règle de définition ou une contrainte.

Remarque

Dans les phases 2 et 3, il peut être utile de choisir une règle qui est définie non pas comme une expression explicite de ses arguments, mais comme un sous-modèle déjà défini ou dont l'élaboration est remise à plus tard. Dans ce cas, la partie gauche (celle des grandeurs définies ou expliquées) peut être constituée d'une liste de grandeurs correspondant aux résultats du sous-modèle.

e) Traitement d'un exemple

On illustrera le processus de construction progressive par le traitement d'un exemple déjà utilisé dans le chapitre 17. Nous envisagerons de résoudre le problème du calcul du traitement d'un employé durant un certain nombre de mois.

La dimension du problème apparaît immédiatement :

les mois de la période considérée,

Les résultats qu'on aimerait obtenir sont :

le traitement net payé chaque mois,

le total des retenues fiscales,
le total des cotisations sociales.

Les données qui semblent utiles comprennent certainement les suivantes :

l'ancienneté à chaque mois,
le montant des primes de chaque mois,
l'index de chaque mois.

On peut déjà écrire, en choisissant judicieusement les noms :

Données

$T = t_1..t_n$: entier (mois) ; mois de la période pour laquelle on considère le calcul
 $ANCIENNETE_T$: entier (année) ; ancienneté de l'employé au mois T
 $PRIMES_T$: entier (F) ; primes auxquelles l'employé a droit au mois T
 $INDEX_T$: réel ; index du mois T

Résultats

NET_PAYE_T : entier (F) ; traitement net payé à l'employé le mois T
 $TOTAL_RETENUE$: entier (F) ; total des retenues fiscales pour la période
 $TOTAL_COTISATION$: entier (F) ; total des cotisations sociales pour la période

Partant des résultats, nous devons trouver une expression simple pour chacun d'eux. Le net payé d'un mois T est obtenu en retirant la retenue fiscale du montant net imposable. Le total des retenues correspond au cumul des retenues fiscales de chaque mois, et le total des cotisations s'obtient par le cumul des cotisations sociales de chaque mois. Nous pouvons donc définir trois grandeurs internes et trois règles :

Données

$T = t_1..t_n$: entier (mois) ; mois de la période pour laquelle on considère le calcul
 $ANCIENNETE_T$: entier (année) ; ancienneté de l'employé au mois T
 $PRIMES_T$: entier (F) ; primes auxquelles l'employé a droit au mois T
 $INDEX_T$: réel ; index du mois T

Résultats

NET_PAYE_T : entier (F) ; traitement net payé à l'employé le mois T
 $TOTAL_RETENUE$: entier (F) ; total des retenues fiscales pour la période
 $TOTAL_COTISATION$: entier (F) ; total des cotisations sociales pour la période

Grandeurs internes

$NET_IMPOSABLE_T$: entier (F) ; montant net imposable de l'employé au mois T
 $RETENUE_FISCALE_T$: entier (F) ; retenue fiscale imposée à l'employé au mois T
 $COTISATION_SOCIALE_T$: entier (F) ; retenue de cotisation sociale de l'employé au mois T

Règles

$$\begin{aligned}\text{NET_PAYE}_T &= \text{NET_IMPOSABLE}_T - \text{RETENUE_FISCALE}_T \\ \text{TOTAL_RETENUE} &= \sum_T \text{RETENUE_FISCALE}_T \\ \text{TOTAL_COTISATION} &= \sum_T \text{COTISATION_SOCIALE}_T\end{aligned}$$

Nous ne pouvons encore souligner aucune grandeur en partie droite des règles. Cherchons à expliquer chacune des grandeurs internes. Le montant net imposable est le montant brut diminué de la cotisation sociale. La retenue fiscale est une fonction du net imposable (cette fonction peut être exprimée par une table de paliers de revenus et une règle proportionnelle entre deux paliers, ou encore par une formule de calcul); nous cacherons cette fonction sous la forme du sous-modèle calcul_RETENUE.

Nous mettons ainsi en évidence deux nouvelles grandeurs : le *montant brut* (BRUT) et le *niveau* (NIVEAU). Il apparaît rapidement que ce dernier est une donnée qui n'avait pas été repérée initialement et que le premier est une grandeur calculable (donc interne). En outre, nous soulignons, à titre de contrôle, les grandeurs en partie droite qui sont soit des données, soit des grandeurs expliquées jusqu'ici. Nous en sommes dès lors au point suivant :

Modèle TRAITEMENT

Données

$T = t_1..t_n$: entier (mois)	; mois de la période pour laquelle on considère le calcul
ANCIENNETE_T : entier (année)	; ancienneté de l'employé au mois T
PRIMES_T : entier (F)	; primes auxquelles l'employé a droit au mois T
INDEX_T : réel	; index du mois T
NIVEAU_T : entier	; niveau de l'employé au mois T

Résultats

NET_PAYE_T : entier (F)	; traitement net payé à l'employé le mois T
TOTAL_RETENUE : entier (F)	; total des retenues fiscales pour la période
TOTAL_COTISATION : entier (F)	; total des cotisations sociales pour la période

Grandeurs internes

NET_IMPOSABLE_T : entier (F)	; montant net imposable de l'employé au mois T
RETENUE_FISCALE_T : entier (F)	; retenue fiscale imposée à l'employé au mois T
$\text{COTISATION_SOCIALE}_T$: entier (F)	; retenue de cotisation sociale de l'employé au mois T
BRUT_T : entier (F)	; traitement brut de l'employé au mois T

Règles

$$\begin{aligned}\text{NET_PAYE}_T &= \text{NET_IMPOSABLE}_T - \text{RETENUE_FISCALE}_T \\ \text{TOTAL_RETENUE} &= \sum_T \text{RETENUE_FISCALE}_T \\ \text{TOTAL_COTISATION} &= \sum_T \text{COTISATION_SOCIALE}_T\end{aligned}$$

```

COTISATION_SOCIALET = (0,1+NIVEAUT/100)*BRUTT
NET_IMPOSABLET = BRUTT - COTISATION_SOCIALET
RETENUE_FISCALET = calcul_RETENUE(NET_IMPOSABLET)

```

Modèle calcul_RETENUE

Données

NET : entier (F) ; traitement avant imposition fiscale

Résultats

RETENUE : entier (F) ; retenue fiscale

Règles

RETENUE = ...

Le montant brut dépend du niveau, de l'ancienneté et des primes de l'employé, ainsi que de l'index. Il semble réaliste de construire un modèle spécialisé pour son calcul. Nous l'appellerons calcul_BRUT.

On vérifie que toutes les données identifiées jusqu'ici sont utilisées.

L'analyse des données nous amène à fixer les éventuelles contraintes de validité. A titre d'exemple, on ne retient que l'intervalle de valeurs de NIVEAU. Pour être complet nous devrions également examiner les autres données.

On obtient alors le modèle de la figure 18.1. Ce modèle est complet car on a pu y souligner toutes les grandeurs apparaissant en partie droite. Il reste cependant à construire les sous-modèles calcul_BRUT et calcul_RETENUE, ce qui est laissé à l'initiative du lecteur.

Modèle TRAITEMENT

Données

T = t₁..t_n : entier (mois) ; mois de la période pour laquelle on considère le calcul
 ANCIENNETE_T : entier (année) ; ancienneté de l'employé au mois T
 PRIMES_T : entier (F) ; primes auxquelles l'employé a droit au mois T
 INDEX_T : réel ; index du mois T
 NIVEAU_T : entier ; niveau de l'employé au mois T

Résultats

NET_PAYE_T : entier (F) ; traitement net payé à l'employé le mois T
 TOTAL_RETENUE : entier (F) ; total des retenues fiscales pour la période
 TOTAL_COTISATION : entier (F) ; total des cotisations sociales pour la période

Grandeurs internes

NET_IMPOSABLE_T : entier (F) ; montant net imposable de l'employé au mois T
 RETENUE_FISCALE_T : entier (F) ; retenue fiscale imposée à l'employé au mois T

COTISATION_SOCIALE_T : entier (F) ; retenue de cotisation sociale
de l'employé au mois T
BRUT_T : entier (F) ; traitement brut de l'employé au mois T

Règles

$NET_PAYE_T = NET_IMPOSABLE_T - RETENUE_FISCALE_T$
 $TOTAL_RETENUE = \sum_T RETENUE_FISCALE_T$
 $TOTAL_COTISATION = \sum_T COTISATION_SOCIALE_T$
 $COTISATION_SOCIALE_T = (0,1 + NIVEAU_T/100) * BRUT_T$
 $NET_IMPOSABLE_T = BRUT_T - COTISATION_SOCIALE_T$
 $RETENUE_FISCALE_T = \text{calcul_RETENUE}(NET_IMPOSABLE_T)$
 $BRUT_T = \text{calcul_BRUT}(ANCIENNETE_T, NIVEAU_T, PRIMES_T, INDEX_T)$
 $2 \leq NIVEAU \leq 12$

Modèle calcul_RETENUE

Données

NET : entier (F) ; traitement avant imposition fiscale

Résultats

RETENUE : entier (F) ; retenue fiscale

Règles

RETENUE = ...

Modèle calcul_BRUT

Données

ANCIENNETE : entier (année) ; ancienneté de l'employé
 NIVEAU : entier ; niveau de l'employé
 PRIMES : entier (F) ; primes auxquelles l'employé a droit
 INDEX : réel ; index courant

Résultats

BRUT : entier (F) ; traitement brut

Grandeurs internes

BRUT_D_B : entier (F) ; traitement brut de base

Règles

BRUT_D_B = ...
 $BRUT = (BRUT_D_B + PRIMES) * INDEX$

Figure 18.1 - Traitement d'un exemple - Modèle final

18.3.2 Normalisation du modèle

Un modèle d'une certaine complexité peut souffrir de défauts structurels ou de maladresses de construction qui risquent d'en diminuer la lisibilité, et partant, la fiabilité et son aptitude à évoluer en toute sécurité. On ne cherche donc pas ici à détecter des erreurs fondamentales, mais plutôt à améliorer les qualités du modèle en tant que support de communication. On vérifiera notamment les points suivants :

a) Complétude

Le modèle est-il complet? Chaque grandeur a-t-elle un type, une unité, un commentaire explicatif?

b) Élimination de la redondance⁴

En principe, une grandeur exprime un concept pertinent du domaine d'application. D'autre part, chaque concept pertinent doit être représenté une et une seule fois. On vérifiera qu'il en est ainsi, non seulement par l'examen de la liste des grandeurs, mais aussi par la comparaison des définitions.

- Deux grandeurs qui ont la même expression de définition décrivent peut-être le même concept; dans ce cas, on élimine l'une d'entre elles.
- Deux grandeurs qui, après remplacement des grandeurs en partie droite par leur définition, ont la même expression de définition décrivent peut-être le même concept.

Dans l'exemple ci-dessous, après remplacement de C et D par leur définition, on observe que les grandeurs E1 et E2 ont la même expression de définition. Il faut se demander si ces grandeurs n'ont pas la même signification.

$$\begin{aligned}C &= A + B \\D &= 2 \cdot A - B \\E1 &= 2 \cdot C \\E2 &= D + 3 \cdot B\end{aligned}$$

- Si la définition d'une grandeur inclut celle d'une autre grandeur, il est possible que cette dernière intervienne dans la définition de la première. Dans ce cas, on peut simplifier la première définition en y remplaçant la définition de la seconde grandeur par celle-ci.

Dans l'exemple ci-dessous, on observe que la définition de E1 peut être retrouvée dans celle de E2.

$$\begin{aligned}E1 &= 2 \cdot A + 1 \\E2 &= 2 \cdot B \cdot A + B\end{aligned}$$

On peut se demander si la formulation suivante ne serait pas plus pertinente :

$$\begin{aligned}E1 &= 2 \cdot A + 1 \\E2 &= B \cdot E1\end{aligned}$$

c) Restructuration

Un modèle complexe peut être peu lisible, et par là, difficile à valider, à corriger et à modifier. On examinera l'opportunité de restructurer le modèle de manière à en rendre la structure plus simple. En particulier, on recherchera la possibilité d'en extraire des sous-modèles naturels. D'autre part, on cherchera également à simpli-

4. Fonction suggérée dans [Hick, 91].

fier certains fragments par l'application d'autres opérations de restructuration, telles que les transformations de contraction, de factorisation de règles et d'éclatement⁵.

18.3.3 Validation du modèle

Un modèle sera examiné afin de déterminer s'il ne contient pas d'erreurs de logique. Il est impossible dans le cadre d'un tel ouvrage de couvrir entièrement un tel sujet. Celui-ci relève en effet de théories souvent encore en développement⁶. Nous donnerons cependant quelques recommandations utiles correspondant à des erreurs typiques, recommandations qui s'organisent selon deux axes : la vérification de la cohérence et les tests.

a) Vérification de la cohérence du modèle

Il est possible de détecter la présence de certaines erreurs majeures par l'analyse de la forme d'un modèle, et particulièrement de ses règles. C'est ainsi qu'on analysera successivement

- la validité de la structure globale du modèle,
- la structure des règles de définition multiple,
- la structure des règles de récurrence,
- la cohérence des unités,
- la cohérence des dimensions,
- la cohérence des domaines de valeurs.

Ces points de validation sont développés dans la section 18.5.

b) Test du modèle

Le test d'un modèle consiste à l'exécuter sur un jeu de données réel ou imaginaire, afin de vérifier l'exactitude des résultats qu'il produit. Cette exécution peut être manuelle, et se baser sur l'expression abstraite du modèle, ou automatique, dans le cas où le modèle a été implanté dans une feuille de calcul.

Le principal problème est celui du choix judicieux des jeux de test, car un test est généralement une opération coûteuse en temps. Une approche pragmatique consiste à travailler comme suit :

- On établit des jeux de données représentatifs, constitués de valeurs normales (pas de cas particuliers, pas de valeurs extrêmes), et on évalue le modèle pour ces données.
- On repère pour chaque donnée les valeurs remarquables de son domaine de valeurs. Cette notion est à interpréter pour chaque donnée : valeur minimum, valeur maximum, 0, 1, par exemple. Le calcul des domaines de valeurs,

5. Les matériaux relatifs aux techniques de transformation de modèles de calcul sont disponibles sur le site de l'ouvrage.

6. On peut citer par exemple l'interprétation abstraite, qui cherche à déterminer la cohérence de spécifications conceptuelles ou opérationnelles, ou encore la théorie des tests de programmes.

évoqué en 18.5.6, est extrêmement utile pour déterminer ces valeurs remarquables.

On construit alors des jeux de tests en partant d'un jeu représentatif, et en y introduisant une à une, pour chaque donnée, chaque valeur remarquable de son domaine de valeurs. On construit également des jeux de tests en introduisant simultanément des valeurs remarquables pour plusieurs données.

- Si le modèle est décomposable ou comporte des fragments identifiables, on teste chacun d'eux individuellement.
- On enregistre soigneusement ces jeux de tests, afin de les réutiliser, soit lors de problèmes ultérieurs, soit lors de la modification du modèle, afin de vérifier sa conformité par rapport à l'ancienne version (tests dits de *régression*). A ce titre, ces jeux de tests et les résultats qu'ils produisent sont à considérer comme faisant partie intégrante de la documentation du modèle.

18.3.4 Généralisation par dimensionnement

Nous signalons ici une approche qui peut simplifier la tâche d'analyse d'un problème comportant plusieurs dimensions. Il est en effet parfois difficile de formaliser un tel problème en considérant toutes les dimensions à la fois.

L'idée consiste à ignorer une ou plusieurs des dimensions et à construire le modèle réduit qui décrit cette vue simplifiée du problème. Lorsque ce modèle simplifié est disponible et qu'il a été validé, voire testé, on le modifie de manière à y inclure une des dimensions écartées. Cette opération consiste à

- déterminer les grandeurs qui dépendent de cette dimension, et les ajuster; pour ce faire, on dimensionne les données dépendant de cette dimension, puis on calcule les dimensions des grandeurs qui ne dépendent que des données (selon les règles 18.5.5); on poursuit en examinant successivement toutes les grandeurs jusqu'à ce que tous les résultats aient été examinés et, si nécessaire, dimensionnés;
- ajouter les grandeurs dérivées dont la définition inclut une fonction agrégative basée sur la nouvelle dimension;
- si nécessaire, vérifier la cohérence des dimensions des règles (selon les principes proposés en 18.5.5), afin de détecter les erreurs éventuelles; si la propagation des dimensions s'est déroulée correctement, cette étape est inutile.

Cette approche a été suivie, bien qu'informellement, dans la construction du modèle 17.9 à partir du modèle 17.1, et dans celle du modèle 17.10 à partir du modèle 17.9. Observons encore qu'on peut traduire cette démarche en donnant une forme particulière au modèle final dans lequel un modèle non dimensionné est exprimé comme un sous-modèle dont l'invocation lui transmet des grandeurs dimensionnées. Le modèle 17.14 en est une illustration directe : le modèle complet est temporel (dépend du temps T), mais les sous-modèles sont indépendants du temps.

18.4 SOUS-MODÈLES NON DIRECTIONNELS

Il peut arriver que la démarche générale de résolution proposée à la section 18.3 ne soit pas entièrement applicable au problème posé. Il en est ainsi par exemple lorsqu'un sous-ensemble de grandeurs s'expriment naturellement par un jeu de relations non directionnelles. Ces grandeurs sont donc connues par un système d'équations et non pas par leur définition. Considérons l'exemple d'un modèle dont une grandeur G obéit à une loi $G = f(X)$ telle que la suivante, où k_0 et k_1 sont perçus comme des paramètres et X comme une grandeur variable :

$$G = k_1 * X + k_0$$

Dans le problème posé, on suppose que les paramètres k_0 et k_1 ne sont pas connus. En revanche, on connaît les valeurs de G pour deux valeurs particulières de x (il s'agit par exemple de deux mesures expérimentales) :

$$\begin{aligned} G_1 &= f(X_1) \\ G_2 &= f(X_2) \end{aligned}$$

ou plus précisément :

$$\begin{aligned} G_1 &= k_1 * X_1 + k_0 \\ G_2 &= k_1 * X_2 + k_0 \end{aligned} \quad (\text{expr. 18.1})$$

Une première idée de modèle se présenterait donc comme suit :

Données

G_1, G_2, X_1, X_2
 X
 . . .

Résultats

G
 . . .

Grandeurs internes

k_0, k_1

Règles

. . .
 $G = k_1 * X + k_0$
 . . .

Cependant le problème ainsi posé ne suggère pas immédiatement d'expressions de définition explicites de k_0 et k_1 . Pour obtenir ces règles, il nous faut procéder à une **résolution analytique** des relations (expr. 18.1) selon k_1 et k_2 en fonction de G_1 , G_2 , X_1 , X_2 . On obtient alors les expressions interdépendantes (expr. 18.2), ou les expressions indépendantes (expr. 18.3) :

$$\begin{aligned} k1 &= (G1-G2) / (X1-X2) \\ k0 &= G1-k1*X1 \end{aligned} \quad (\text{expr. 18.2})$$

$$\begin{aligned} k1 &= (G1-G2) / (X1-X2) \\ k0 &= G1 - (G1-G2) / (X1-X2) * X1 \end{aligned} \quad (\text{expr. 18.3})$$

Le modèle peut alors s'écrire comme suit :

Données

G1, G2, X1, X2
X
. . .

Résultats

G
. . .

Grandeurs internes

k0, k1

Règles

. . .
k1 = (G1-G2) / (X1-X2)
k0 = G1 - (G1-G2) / (X1-X2) * X1
G = k1*X + k0
. . .

Il apparaît donc qu'une partie du modèle (la définition de k1 et k0) échappe à la démarche systématique et intuitive proposée jusqu'ici, et qu'elle réclame un traitement analytique préalable pour se conformer à la structure de simples règles de définition directionnelles.

Il apparaît en outre que les définitions des paramètres k1 et k0, ainsi que leur procédé d'obtention, sont de nature purement technique et perturbent la structure du modèle initial, désormais beaucoup moins lisible. Il est préférable de les **encapsuler** dans un sous-modèle spécifique, comme suggéré dans les définitions ci-dessous :

Modèle principal

Données

X
G1, G2, X1, X2

Résultats

G
. . .

Grandeurs internes

k0, k1

Règles

```

. . .
{ $k_1, k_0$ } = calcul-paramètres ( $G_1, G_2, X_1, X_2$ )
 $G = k_1 * X + k_0$ 
. . .

```

Modèle calcul-paramètres

Données

G_1, G_2, X_1, X_2

Résultats

k_1, k_0

Règles

$k_1 = (G_1 - G_2) / (X_1 - X_2)$

$k_0 = G_1 - (G_1 - G_2) / (X_1 - X_2) * X_1$

Cette structure de modèle est d'autant plus pertinente que la résolution analytique suggérée n'est pas toujours possible. Il faudra parfois faire appel à des techniques de résolution numériques de systèmes d'équations (section 17.11) lorsque l'outil le permettra (résolveurs d'équations ou certains tableurs tels qu'EXCEL).

18.5 COHÉRENCE D'UN MODÈLE

En toute généralité, un modèle est correct s'il se comporte comme le domaine d'application qu'il est chargé de décrire, dans toutes les situations tombant dans son domaine de validité. En d'autres termes, étant donné des valeurs des données mesurant les propriétés correspondantes d'une situation réelle, le modèle doit produire des résultats conformes aux propriétés sur lesquelles on s'interroge. S'il est le plus souvent impossible à vérifier de manière absolue qu'un modèle jouit de cette qualité, on peut tout au moins énoncer quelques règles que tout modèle doit respecter. Un modèle qui respecte ces règles peut être correct (sans que cela soit garanti), tandis qu'un modèle qui ne les respecte pas est incorrect de façon certaine et produira tôt ou tard, s'il fonctionne, des résultats erronés.

Sans prétendre en aucune manière à l'exhaustivité, nous décrirons des règles qui correspondent aux principales erreurs commises par les modélisateurs débutants : règles de cohérence structurelle, de cohérence des règles de définition multiple, de cohérence des règles de récurrence, de cohérence des unités, de cohérence des dimensions et enfin celles, très importantes, de cohérence des domaines de valeurs.

18.5.1 Cohérence structurelle

Le modèle doit être conforme à la structure définie au chapitre 15.

- Toutes les données sont utilisées : chaque donnée doit apparaître dans la partie droite d'au moins une règle de définition.
- Toute grandeur qui n'est pas une donnée est dérivable (résultat ou grandeur interne) et reçoit une définition : elle doit donc apparaître en partie gauche d'une règle de définition.

- Une donnée ne peut être définie par une règle de définition.
- Une grandeur dérivable n'est définie qu'une seule fois : elle ne peut donc apparaître en partie gauche que d'une seule règle de définition.
- Une grandeur dérivable est utilisée : elle doit être un résultat, ou apparaître en partie droite d'une règle de définition ou dans une règle de contrainte.

La procédure de construction proposée dans la section 18.3 garantit que le modèle obtenu respecte ces règles.

18.5.2 Cohérence des règles de définition multiple

La structure d'une définition multiple peut être complexe et susceptible de comporter des erreurs. On vérifiera qu'elle est complète, qu'elle n'est pas ambiguë et qu'elle ne comporte pas de branches mortes.

- En principe, une définition multiple doit être *complète* : pour tout état du modèle⁷, une des conditions de la définition doit être vraie, et par conséquent, la grandeur doit être définie. La définition ci-dessous n'est en principe pas complète, car ALLOCATION est indéfinie pour NET_PAYE = 1000 :

```

ALLOCATION = 100      si NET_PAYE < 1000
              0,1*NET_PAYE si 1000 < NET_PAYE < 3000
              300      si NET_PAYE ≥ 3000

```

La complétude n'est pas toujours aisée à prouver. Ainsi, dans un modèle qui contiendrait en outre la contrainte

```
NET_PAYE ≠ 1000
```

la définition de ALLOCATION serait complète, puisque le cas NET_PAYE = 1000 ne peut survenir.

Cette règle peut cependant être nuancée : dans les situations où une grandeur à définition multiple n'intervient dans le calcul d'aucun résultat, sa valeur est indifférente. Tel serait le cas du fragment suivant, où C peut être indéfinie (si A = B), mais n'intervient dans le calcul de D que si elle est définie.

```

C = 0    si A > B
      1    si A < B
D = 2*C   si A ≠ B
      -1   si A = B

```

Notons qu'une règle qui contient une branche *sinon* est complète par construction.

- Une définition multiple doit être *non ambiguë* : pour tout état du modèle, une seule des conditions de la définition peut être vraie. Il ne peut arriver que la grandeur soit définie plus d'une fois, comme dans la définition suivante, qui est ambiguë.

7. Si chaque grandeur du modèle reçoit une valeur, l'ensemble de ces valeurs correspond à un état du modèle et représente un état du domaine d'application.

```

ALLOCATION = 100          si NET_PAYE < 1000
                      0,1*NET_PAYE si 1000 ≤ NET_PAYE < 3000
                      300          si NET_PAYE ≥ 2000

```

En effet, pour $2000 \leq \text{NET_PAYE} < 3000$, ALLOCATION reçoit deux définitions distinctes. Tout comme la complétude, la non-ambiguïté peut s'avérer difficile à prouver.

- Une définition multiple ne peut comporter de *branches mortes* : pour toute condition de la définition, il doit exister au moins un état pour lequel elle est vraie et un état pour lequel elle est fausse. Dans l'exemple ci-dessous, la définition de D est complète et cohérente. Cependant, la condition (C1 et C2) sera toujours fausse, quelles que soient les valeurs de A et B, et définit donc une branche morte, tandis que la condition (non C1 ou non C2) sera toujours vraie. S'il ne s'agit probablement pas d'une erreur *bloquante* du modèle, cette définition révèle presque certainement une erreur d'analyse.

```

C1 = (A = B)
C2 = (A ≠ B)
D  = 0 si (C1 et C2)
    1 si (non C1 ou non C2)

```

18.5.3 Cohérence des règles de récurrence

Les règles de récurrence sont fréquemment le siège d'erreurs de construction : initialisation manquante, erreur d'indice, etc. On vérifiera en particulier

- qu'à chaque règle de récurrence correspond une (ou plusieurs) règle d'initialisation qui permette le démarrage de la récurrence en fixant la valeur de départ de la variable dimensionnée;
- que la règle d'initialisation, accompagnée de sa règle de récurrence, couvre bien toutes les valeurs de la dimension (ou en tout cas le sous-ensemble pertinent) une et une seule fois : en particulier, on sera attentif au traitement de la première valeur de la dimension (est-elle traitée, l'est-elle dans l'initialisation seulement?), de la valeur suivante (est-elle traitée?), de la dernière valeur;
- que la règle d'initialisation, accompagnée de sa règle de récurrence, ne couvre que les valeurs pertinentes de la dimension : ne spécifie-t-elle pas une valeur qui précède la première, ou une valeur qui suit la dernière (comportement aux limites)?

Le lecteur est invité à évaluer les six tentatives suivantes de description de la série de Fibonacci (tout terme, sauf les deux premiers qui valent 1, est égal à la somme des deux termes qui le précèdent) : 1, 1, 2, 3, 5, 8, 13, etc.

Essai n° 1 $F_1 = 1$
 $F_i = F_{i-1} + F_{i-2} \quad i = 2..n$

Essai n° 2 $F_1 = 1$
 $F_i = F_{i-1} + F_{i-2} \quad i = 3..n$

Essai n° 3 $F_1 = 1$

$$F_2 = 1$$

$$F_i = F_{i-1} + F_{i-2} \quad i = 3..n$$

Essai n° 4

$$F_1 = 1$$

$$F_{i+1} = F_i + F_{i-1} \quad i = 2..n$$

Essai n° 5

$$F_1 = 1$$

$$F_2 = 1$$

$$F_{i+1} = F_i + F_{i-1} \quad i = 2..n-1$$

Essai n° 6

$$F_i = F_{i-1} + F_{i-2} \quad \begin{array}{l} \text{si } i > 2 \\ 1 \end{array} \quad i = 1..n$$

18.5.4 Cohérence des unités

On n'additionne pas, dit-on, des pommes et des poires. Il en est de même dans les expressions apparaissant dans un modèle. Les grandeurs qui interviennent dans ces expressions doivent respecter des règles de cohérence. Cette vérification, généralement assez simple, permet de déceler rapidement des erreurs importantes dans un modèle.

- Les arguments des opérateurs utilisés dans les expressions ont des types conformes à la spécification de ces opérateurs. Par exemple, si on peut soustraire ou ajouter un entier à une date, ou encore calculer la différence entre deux dates, on ne peut additionner ni multiplier deux dates; on peut soustraire deux températures, mais pas les additionner. Une grandeur définie et son expression de définition doivent être du même type. Des expressions apparaissant dans une condition de comparaison d'une règle de contrainte doivent être de types comparables.
- Dans une règle de définition, la grandeur définie et l'expression de définition ont même unité. La règle du modèle ci-dessous, relatif à un mouvement uniformément accéléré, est cohérente du point de vue des unités.

Données

V0 : réel (m/s)	; vitesse de départ
V1 : réel (m/s)	; vitesse finale
D : entier (m)	; distance parcourue

Résultat

A : réel (m/s ²)	; accélération uniforme du véhicule
------------------------------	-------------------------------------

Règles

$$A = 0,5 * (V1^2 - V0^2) / D$$

En effet, en appliquant les règles de calcul qui seront vues ci-dessous, l'unité de la partie droite se calcule comme suit : $((\text{m/s})^2 - (\text{m/s})^2) / \text{m} = (\text{m/s})^2 / \text{m} = \text{m/s}^2$

- Les termes d'une somme (différence) ont même unité. L'exemple ci-dessous est incorrect, car on y confond poids et valeur financière d'un bien.

Données

D : d_1, d_2, \dots, d_N ; départements de production
 V_D : entier (t) ; volume (en tonnes) de production du département D
 I : entier (F) ; valeur de la consommation interne (en euros)

Résultat

E : entier (t) ; volume exporté

Règles

$$E = \sum_D V_D - I$$

- Une comparaison apparaissant dans une condition (dans une contrainte ou dans une définition multiple) fait intervenir des expressions ayant même unité. Le fragment ci-dessous est incorrect. On admet que la constante 0 est exprimée dans l'unité (F).

Données

P : p_1, p_2, \dots, p_N ; postes budgétaires
 T : t_1, t_2, \dots, t_M ; périodes budgétaires
 B_P : entier (F) ; montant initial du budget du poste P
 $D_{P,T}$: entier (F) ; dépense de la période T sur le poste P
 D_{Max} : entier ; dépense maximale, en % du montant initial du budget

Règles

. . .

$$0 < D_{P,T} < D_{Max}$$

- Lors de l'invocation d'un sous-modèle, les arguments en correspondance ont même unité. Le fragment ci-dessous est incorrect (on rappelle que le radian n'est pas une unité, mais un rapport).

Données

L : réel (m) ; longueur du bras de la cabine de simulation spatiale

Grandeurs internes

V : réel (tour/s) ; vitesse de rotation de la cabine en tours par seconde

Résultat

VM : réel (m/s) ; vitesse linéaire de la cabine

Règles

$$V = \dots$$

$$VM = \text{VITESSE-LINEAIRE}(V, L)$$

Modèle VITESSE-LINEAIRE**Données**

VA : réel (rad/s) ; vitesse angulaire du mobile
 R : réel (m) ; rayon de la trajectoire

Résultat

V_L : réel (m/s) ; vitesse linéaire du mobile

Règles

$$V_L = 2 \cdot \pi \cdot V_A \cdot R$$

- Il doit y avoir compatibilité entre l'unité d'une grandeur temporelle et la périodicité de sa dimension. Le fragment ci-dessous est incorrect.

Données

T : 1990..1995

I_T : entier (t/semestre) ; volume semestriel des importations

Calcul de l'unité d'une expression

Il existe des règles permettant de déduire l'unité d'une expression à partir des unités de ses composants. Le lecteur intéressé trouvera ces règles sur le site de l'ouvrage.

18.5.5 Cohérence des dimensions

Les erreurs d'indices, ou de dimensions, sont souvent le signe d'une erreur d'analyse, soit dans l'expression de définition (formule erronée : mauvais indice de sommation, oubli d'une sommation), soit dans la structure des grandeurs (oubli d'une dépendance : l'*index* dépend du *temps*, mais pas de l'*employé*). Il est donc important de vérifier l'usage des dimensions⁸. Les règles ci-dessous permettent de relever des erreurs classiques. Dans cette section, comme dans cet ouvrage d'ailleurs, on fait l'hypothèse que les dimensions sont indépendantes.

Appelons **dimensions** d'une grandeur G l'ensemble, noté $\dim(G)$, des dimensions de cette grandeur. Par exemple,

$$\begin{aligned} \dim(D_I) &= \{I\} \\ \dim(D_{I,J}) &= \{I, J\} \\ \dim(D) &= \{\} \end{aligned}$$

- Dans une règle de définition, les dimensions de la grandeur définie forment un sur-ensemble de celles de l'expression de définition. Plus précisément, étant donné la règle de définition $G = E$, on doit avoir

$$\dim(E) \subseteq \dim(G)$$

En cas de définition multiple, cette propriété doit être respectée par chacune des branches de définition. Considérons « E_i si C_i », la i ème branche de la définition. Elle doit vérifier :

$$\begin{aligned} \dim(E_i) &\subseteq \dim(G) \\ \dim(C_i) &\subseteq \dim(G) \end{aligned}$$

Selon ce principe, les règles ci-dessous sont cohérentes :

8. Il s'agit ici des dimensions considérées globalement (comme dans D_I) et non de leurs valeurs élémentaires (comme dans D_{i1}).

$$\begin{aligned}
D_I &= 3*A_I - B_I \\
D_I &= A + B \\
D_I &= 3*A - B_I*C_I \\
D_{I,J} &= 3*A_I + 1 \\
D_{I,J} &= A_I + 1 \text{ si } E_J = F_J \\
&\quad B_J \quad \text{si } E_J \neq F_J
\end{aligned}$$

alors que les suivantes ne le sont pas :

$$\begin{aligned}
D_I &= A_{I,J} - 1 \\
D_I &= A_I \quad \text{si } B_{I,J} = 1 \\
&\quad -A_I \quad \text{si } B_{I,J} \neq 1
\end{aligned}$$

Il est facile de se convaincre de la portée de ces propriétés en développant une règle de définition selon ses dimensions. Considérons d'abord une règle cohérente telle que

$$D_{I,J} = 3*A_I + 1$$

où I a pour domaine de valeurs $\{1, 2\}$ et J , $\{a, b\}$. Il vient :

$$\begin{aligned}
D_{1,a} &= 3*A_1 + 1 \\
D_{1,b} &= 3*A_1 + 1 \\
D_{2,a} &= 3*A_2 + 1 \\
D_{2,b} &= 3*A_2 + 1
\end{aligned}$$

Ces règles sont manifestement bien formées. En revanche, appliquons ce développement à la règle suivante, déclarée incohérente

$$D_I = A_{I,J} - 1$$

où I et J ont mêmes domaines de valeurs que ci-dessus. Il vient :

$$\begin{aligned}
D_1 &= A_{1,a} - 1 \\
D_1 &= A_{1,b} - 1 \\
D_2 &= A_{2,a} - 1 \\
D_2 &= A_{2,b} - 1
\end{aligned}$$

Il est clair que ce fragment de modèle est incorrect puisque les grandeurs D_1 et D_2 y sont définies plus d'une fois.

- *En principe*, une règle de contrainte qui s'exprime sous la forme « $E1 \text{ op } E2$ », où op est un opérateur de comparaison, respecte la propriété

$$\dim(E1) \subseteq \dim(E2) \quad \text{ou} \quad \dim(E2) \subseteq \dim(E1)$$

Considérons la contrainte suivante, qui respecte cette propriété :

$$D_I < A_{I,J}$$

où I a pour domaine de valeurs $\{1, 2\}$ et J , $\{a, b\}$. Il vient en développant :

$$\begin{aligned}
D_1 &< A_{1,a} \\
D_1 &< A_{1,b} \\
D_2 &< A_{2,a} \\
D_2 &< A_{2,b}
\end{aligned}$$

Ces règles de contrainte sont correctes et raisonnables.

Examinons maintenant la contrainte suivante, qui ne respecte pas cette propriété :

$$D_I < A_J$$

Son développement donne les contraintes

$$D_1 < A_a$$

$$D_1 < A_b$$

$$D_2 < A_a$$

$$D_2 < A_b$$

Sans être incorrect, ce fragment est peu orthodoxe, et pourrait révéler une erreur d'analyse. On s'en assurera en vérifiant par exemple que ce fragment est bien équivalent à la contrainte suivante, plus claire, plus concise, et qui respecte la propriété proposée :

$$\max_I(D_I) < \min_J(A_J)$$

- *En principe*, la dimension d'une fonction agrégative appartient aux dimensions de son argument. Il est prudent de vérifier que dans l'expression $\text{op}_I(E)$ on a la propriété

$$I \in \dim(E) \text{ pour } \text{op} \in \{\Sigma, \Pi, \min, \max, \text{moy}, \text{ou}, \text{etc.}\}.$$

Par exemple, les expressions $\Sigma_I A_I$ et $\Pi_I A_{I,J}$ sont régulières. En revanche, les expressions $\Sigma_I A_J$ et $\Sigma_I A$ ne le sont pas, et seront vérifiées soigneusement. On s'assurera en particulier qu'elle peuvent être simplifiées selon les équivalences suivantes, qui donnent des expressions régulières, claires et concises :

$$\Sigma_I A_J = \text{nombre}(I) * A_J$$

$$\Sigma_I A = \text{nombre}(I) * A$$

Calcul des dimensions d'une expression

Il existe des règles permettant de déduire les dimensions d'une expression à partir de celles de ses composants. Le lecteur intéressé trouvera ces règles sur le site de l'ouvrage.

18.5.6 Cohérence des domaines de valeurs du modèle

Nous abordons ici un problème plus complexe et polymorphe, que nous introduirons par deux exemples classiques.

- Considérons l'expression de définition

$$Y = \arcsin(2 + X^2)$$

La fonction \arcsin réclame un argument appartenant au domaine de valeurs $[-1, 1]$. Cette règle est donc correcte lorsque $-1 \leq 2 + X^2 \leq 1$, ou encore lorsque $-3 \leq X^2 \leq -1$. Il est clair qu'aucune valeur réelle de X ne peut satisfaire cette condition, car elle implique $X^2 < 0$.

L'analyse du domaine de valeurs de \arcsin nous permet de conclure que celui-ci est vide et que cette fonction n'est jamais définie. La règle de définition est donc équivalente à

$$Y = \text{erreur}$$

ce qui n'était probablement pas l'intention de son auteur !

- L'exemple suivant nous permet d'illustrer une situation moins extrême, mais d'autant plus délicate :

$$Y = \sqrt{B-1}$$

La fonction \sqrt{x} réclame un argument X non négatif. Par conséquent, cette règle est correcte seulement si $B \geq 1$. La règle est donc définie pour certaines valeurs de B , et pas pour les autres. Plus précisément, cette règle de définition est équivalente à

$$Y = \begin{array}{ll} \sqrt{B-1} & \text{si } B \geq 1 \\ \text{erreur} & \text{sinon} \end{array}$$

Idéalement la seconde branche de cette règle devrait être morte. Il faut donc prouver que la condition $B \geq 1$ est garantie par le modèle. Cette preuve peut se trouver dans des règles de contrainte dont on peut déduire la propriété $B \geq 1$. Tel serait le cas si on trouvait dans le modèle :

$$\begin{array}{l} A \geq 1 \\ B > 2 * A \end{array}$$

S'il n'existe pas de contraintes de ce type, la preuve doit être trouvée par l'analyse de la règle de définition de B . Si par exemple on trouve la définition :

$$B = 2 + \sin^2(A)$$

alors on en déduit que $B \geq 2$, et donc que $B \geq 1$. Cette preuve peut cependant n'être pas aussi simple. Si la définition de B se présente comme suit

$$B = A + 1$$

alors la propriété $B \geq 1$ est prouvée si $A \geq 0$. C'est donc cette dernière propriété qu'il faut maintenant démontrer selon le même procédé.

Sans entrer dans le détail de cas particuliers, on peut décrire le problème comme suit. Considérons la règle de définition

$$G = f(D)$$

où G est la grandeur définie, f une fonction quelconque, D un argument de cette règle (D est une grandeur dérivable ou une donnée du modèle). On appelle respectivement

- **domaine** de f , noté $\text{dom}(f)$, l'ensemble de toutes les valeurs pour lesquelles la fonction f est définie, c'est-à-dire pour lesquelles elle renvoie une valeur effective correcte ;
- **ensemble des valeurs** de la grandeur D , noté $\text{val}(D)$, l'ensemble de toutes les valeurs que peut prendre la grandeur D lors de l'exécution du modèle ;

- **codomaine** de la fonction f pour l'ensemble de valeurs V , noté $\text{codom}(f(V))$, l'ensemble des valeurs de $f(v)$ pour toutes les valeurs v de V .

On a les propriétés suivantes :

- L'ensemble des valeurs d'une grandeur doit être inclus dans le domaine des valeurs des fonctions dans lesquelles elle intervient comme argument. Si ce n'est pas le cas, la règle peut assigner la valeur **erreur** à la grandeur définie. Cette propriété est une contrainte du modèle.

$$\text{val}(D) \subseteq \text{dom}(f)$$

- L'ensemble des valeurs d'une grandeur dérivable est le codomaine de f pour l'ensemble des valeurs de l'argument de f . Cette propriété est une règle d'inférence du modèle.

$$\text{val}(G) = \text{codom}(f(\text{val}(D)))$$

Ces relations sont essentielles dans la validation du modèle où la règle apparaît. Elles permettent de poser les conditions de validité d'une règle et de propager le calcul des ensembles de valeurs des grandeurs.

Ces définitions et propriétés appliquées à une simple règle de définition sont généralisables au modèle tout entier, qui peut être perçu comme la règle de définition de ses résultats en fonction de ses données. Reprenons en effet la formulation :

Données :	D
Résultats :	R
Règles :	R = F(D)

En généralisant, on a donc :

$$\begin{aligned} \text{val}(D) &\subseteq \text{dom}(F) \\ \text{val}(R) &= \text{codom}(F(\text{val}(D))) \end{aligned}$$

Le problème de la validité du modèle se résume donc à ceci : prouver que les données admissibles D sont un sous-ensemble du domaine de la fonction F que constitue le modèle. On appellera **données admissibles** le sous-ensemble des données qui satisfont à leurs types respectifs et aux règles de contrainte qui les concernent dans le modèle (règles de validation des données).

Ce problème est en général extrêmement complexe et ne peut être résolu que par des techniques d'analyse statique [Lecharlier, 1991]. On en trouvera sur le site de l'ouvrage une analyse sommaire.

Correction d'un modèle

Une question en pratique : comment modifier un modèle de manière que, pour toutes les valeurs correctes des données, il fournisse des valeurs correctes des

9. $\text{codom}(f(V)) = \{x \mid x = f(v) \text{ \& } v \in V\}$. On admet que V ne soit pas entièrement inclus dans $\text{dom}(f)$, et donc que $\text{codom}(f(V))$ contienne la valeur **erreur**.

résultats? On propose trois techniques dont l'applicabilité dépend de la complexité du modèle.

1. Restriction du domaine de valeurs du modèle

Cette technique consiste en une redéfinition des données correctes par la modification ou l'ajout des contraintes de validation des données. On cherche à renforcer ces contraintes de manière que les données qui les satisfont produisent toujours des résultats corrects. Nous partirons de la règle de définition d'un résultat qui est susceptible de renvoyer la valeur **erreur**, et déterminerons, en remontant vers les données, les codomaines et domaines successifs, jusqu'à obtenir les propriétés des données qui déterminent ces ensembles de valeurs. Illustrons cette technique par la modification du modèle ci-dessous :

Donnée

A : réel

Résultats

B : réel

C : réel

Règles

$$B = 2 + \sqrt{A-1}$$

$$C = \arcsin(B-2)$$

$$A \geq 0$$

Calcul des valeurs de B admissibles dans l'expression $\arcsin(B-2)$:

$$-1 \leq B-2 \leq 1$$

$$1 \leq B \leq 3$$

En remplaçant B par sa définition, on obtient la contrainte :

$$1 \leq 2 + \sqrt{A-1} \leq 3$$

puis en réduisant :

$$-1 \leq \sqrt{A-1} \leq 1$$

Étant donné le codomaine $[0..max]$ de la fonction $\sqrt{}$, cette relation se réduit à :

$$0 \leq \sqrt{A-1} \leq 1$$

puis à :

$$0 \leq A-1 \leq 1$$

on en déduit donc la contrainte sur A :

$$1 \leq A \leq 2$$

On peut alors proposer une version correcte du modèle :

Donnée

A : réel

Résultats

B : réel

C : réel

Règles

$$B = 2 + \sqrt{A-1}$$

$$C = \arcsin(B-2)$$

$$1 \leq A \leq 2$$

2. Ajout de préconditions de surveillance

Cette technique est recommandée lorsque le calcul des contraintes de validation sur les données est impossible (certaines fonctions mathématiques n'ont pas d'inverse) ou impraticable. Elle consiste à *protéger* les grandeurs dérivées qui risquent de se voir assigner la valeur erreur par une précondition sur le domaine de valeurs de leurs règles de définition. Intrinsèquement, le modèle n'est pas modifié, mais la violation de ces préconditions avertit immédiatement l'utilisateur de la survenance d'une situation anormale, et de l'endroit où l'erreur se produit. Le diagnostic est à la charge de l'utilisateur, mais il peut être grandement facilité par les informations ainsi fournies. Traitons de nouveau l'exemple ci-dessus en utilisant cette technique.

L'expression $\arcsin(B-2)$ est correcte si

$$1 \leq B \leq 3$$

tandis que l'expression $2 + \sqrt{A-1}$ l'est si

$$A \geq 1$$

On corrige donc le modèle comme suit.

Donnée

A : réel

Résultats

B : réel

C : réel

Règles

$$B = 2 + \sqrt{A-1}$$

$$C = \arcsin(B-2)$$

$$A \geq 1$$

$$1 \leq B \leq 3$$

Proposons un exemple plus concret, que nous analyserons et corrigerons selon les deux techniques décrites ci-dessus. Le modèle de départ se présente comme proposé en 18.2.

Une analyse rapide montre que $RATIO_D$ est indéterminé pour $SOLDES = 0$. Observant que $SOLDE_D \geq 0$, suite aux contraintes sur $BUDGET_D$ et $DEPENSE_D$, on en

déduit que $SOLDE_D$ ne peut être nul que si chaque $SOLDE_D$ est nul, c'est-à-dire si $BUDGET_D = DEPENSE_D$ pour chaque valeur de D . Cet état de choses est permis par les contraintes de validation des données : rien n'interdit que tous les départements épuisent leur budget.

La correction du modèle selon la première technique conduit à ajouter une règle de contrainte sur les données qui correspond à $\sum_D (BUDGET_D - DEPENSE_D) > 0$. Telle quelle, cette contrainte n'est pas opérationnelle, car elle ne peut s'appliquer à une ou plusieurs grandeurs prises isolément.

Données

$D : d_1..d_n$; départements
$BUDGET_D : \text{entier (F)}$; budgets des départements
$DEPENSE_D : \text{entier (F)}$; dépenses des départements

Résultat

$SOLDE_D : \text{entier (F)}$; soldes des départements
$SOLDES : \text{entier (F)}$; solde général
$RATIO_D : \text{réel}$; ratios des soldes des départements

Règles

$SOLDE_D = BUDGET_D - DEPENSE_D$
 $SOLDES = \sum_D SOLDE_D$
 $RATIO_D = SOLDE_D / SOLDES$
 $0 \leq DEPENSE_D \leq BUDGET_D$

Figure 18.2 - Un modèle incorrect

On pourrait par exemple la traduire en la transposant sur la dernière donnée à introduire, soit par exemple $DEPENSE_{d_n}$:

$$DEPENSE_{d_n} < \sum_{D'} (BUDGET_{D'} - DEPENSE_{D'}) + BUDGET_{d_n} \quad (D' = d_1..d_{n-1})$$

Il est clair que cette contrainte est tout à fait impraticable, d'une part parce qu'elle impose un ordre de saisie des données, d'autre part parce qu'elle reporte sur cette dernière donnée la responsabilité d'une erreur qui peut avoir été commise sur une autre donnée.

On préférera alors la seconde technique, illustrée dans la figure 18.3. L'erreur peut se produire, mais l'utilisateur est averti de sa survenue et de son origine.

Données

$D : d_1..d_n$; départements
$BUDGET_D : \text{entier (F)}$; budgets des départements
$DEPENSE_D : \text{entier (F)}$; dépenses des départements

Résultat

$SOLDE_D : \text{entier (F)}$; soldes des départements
$SOLDES : \text{entier (F)}$; solde général
$RATIO_D : \text{réel}$; ratios des soldes des départements

Règles

$$\begin{aligned} \text{SOLDE}_D &= \text{BUDGET}_D - \text{DEPENSE}_D \\ \text{SOLDES} &= \sum_D \text{SOLDE}_D \\ \text{RATIO}_D &= \text{SOLDE}_D / \text{SOLDES} \\ 0 &\leq \text{DEPENSE}_D \leq \text{BUDGET}_D \\ \text{SOLDES} &> 0 \end{aligned}$$

Figure 18.3 - Correction du modèle 18.2 par précondition de surveillance

3. Absorption par définition multiple

Si la première technique n'est pas applicable, il est possible de laisser l'incident se produire, mais de le contrôler (l'absorber) dans la définition même de la grandeur menacée. Cette dernière reçoit une définition multiple dans laquelle une branche assigne une valeur conventionnelle à la grandeur lorsque le domaine de sa fonction est violé. Reprenons le modèle analysé au début de cette section et décidons que pour les valeurs de B qui ne sont pas comprises entre 1 et 3, C reçoit la valeur 0 (ou m, ou absent, ou ...). La définition de B est protégée de la même manière. Le modèle corrigé se présenterait comme suit.

Donnée

A : réel

Résultats

B : réel
C : réel

Règles

$$\begin{aligned} B &= 2 + \sqrt{A-1} && \text{si } A \geq 1 \\ &\mathbf{absent} && \text{sinon} \\ C &= \arcsin(B-2) && \text{si } 1 \leq B \leq 3 \\ &0 && \text{sinon} \end{aligned}$$

Cette technique s'applique particulièrement bien au problème 18.2, qu'on a corrigé à la figure 18.4 en modifiant la définition des RATIO_D de telle sorte que ceux-ci prennent une valeur conventionnelle (par exemple 0) en cas de SOLDES nul.

Données

D : $d_1 \dots d_n$; départements
BUDGET _D : entier (F)	; budgets des départements
DEPENSE _D : entier (F)	; dépenses des départements

Résultat

SOLDE _D : entier (F)	; soldes des départements
SOLDES : entier (F)	; solde général
RATIO _D : réel	; ratios des soldes des départements

Règles

$$\begin{aligned} \text{SOLDE}_D &= \text{BUDGET}_D - \text{DEPENSE}_D \\ \text{SOLDES} &= \sum_D \text{SOLDE}_D \end{aligned}$$

$$\begin{aligned} \text{RATIO}_D &= \text{SOLDE}_D / \text{SOLDES} & \text{si } \text{SOLDES} > 0 \\ &= 0 & \text{si } \text{SOLDES} = 0 \\ 0 &\leq \text{DEPENSE}_D \leq \text{BUDGET}_D \end{aligned}$$

Figure 18.4 - Correction du modèle 18.2 par définition multiple

18.6 EXERCICES

18.6.1 Modèles élémentaires

Proposez un modèle abstrait pour chacun des domaines d'application décrits ci-dessous.

- 18.1 A partir des trois paramètres **taux d'intérêt**, **précompte mobilier** (en %) et **capital initial**, donner le tableau, pour une période de 10 ans, de l'évolution annuelle du capital, des intérêts bruts de l'année écoulée ainsi que des intérêts nets.
- 18.2 On désire gérer les **résultats d'examens** d'une classe d'élèves (une seule session) pour un ensemble de matières. On calculera la moyenne par élève, par matière et pour la classe.
- 18.3 On ajoutera au problème ci-dessus des indicateurs qui spécifient pour chaque élève s'il se situe au-dessus ou au-dessous de la moyenne de la classe. On fera de même pour chaque matière, ce qui permet d'évaluer chaque enseignant par rapport à ses collègues.
- 18.4 Toujours dans le cadre du problème précédent, on inclura en outre un coefficient de pondération pour chaque matière, indiquant quelle est sa part dans le calcul des moyennes par élève et pour la classe. Ce coefficient doit être compris entre 0,5 et 1.
On vérifiera que les indicateurs se comportent de manière cohérente par rapport à ces poids.
- 18.5 On considère un véhicule automobile dont on désire étudier le coût en carburant. Ce coût est à calculer sur une distance donnée, à parcourir à vitesse constante. Le prix du carburant est connu. La consommation obéit à une loi polynomiale quadratique. En outre, des essais préalables ont permis de prendre les mesures de consommation suivantes :

à 60 km/h	: 0,078 l/km
à 90 km/h	: 0,092 l/km
à 120 km/h	: 0,115 l/km

Suggestion : on pensera à définir un sous-modèle technique de calcul des coefficients.

18.6 Construire le modèle de la résolution d'une équation du second degré.

18.7 Un porte-monnaie contient N pièces d'un montant total de S €. On y trouve des pièces de 0,01 €, des pièces de 0,05 € et des pièces de 0,10 € (P de plus que de pièces de 0,01 €). Établir un modèle qui permette de connaître le nombre de pièces de chaque type. Attention, toute combinaison de données ne conduit pas nécessairement à un résultat valide.

Suggestion : l'expression naturelle des relations entre grandeurs ne correspond pas nécessairement à des règles de définition.

18.6.2 Modèles avancés

Proposer un modèle abstrait pour chacun des domaines d'application décrits ci-dessous.

18.8 Un constructeur automobile veut étudier le bénéfice qu'il retire de la vente de différents modèles de véhicule dans différents pays. Il connaît la quantité fabriquée, la quantité vendue, le prix de revient et le prix de vente de chaque modèle dans chaque pays. Il veut connaître le bénéfice par modèle, le bénéfice par pays et le bénéfice par modèle dans chaque pays. Pour simplifier le raisonnement, un véhicule acheté dans un pays n'y est pas nécessairement fabriqué.

Suggestion. Le prix de vente dans un pays dépend de celui-ci tandis que le prix de revient dépend de l'ensemble de tous les pays.

18.9 Le constructeur de la question 18.8 apprend avec étonnement que dans les pays qu'il occupe règne encore une mentalité archaïque, sans doute entretenue par des groupuscules extrémistes, selon laquelle la propension à acheter un véhicule d'une marque est une fonction directe de l'image sociale du constructeur. Etudiant la chose de plus près, notre constructeur constate que les ventes dépendent, selon une loi quadratique, de la proportion de salariés de ses usines qui ont été licenciés ou engagés durant l'année courante dans le pays. Trois mesures ont été faites : (1) si on licencie tout le personnel, les ventes tombent à 10% du chiffre de l'année précédente, (2) à niveau d'emploi égal, les ventes se maintiennent au niveau antérieur, (3) si on double le nombre de salariés, le nombre de ventes augmente de 150%. Cette règle est indépendante des pays et des modèles. Affinez le modèle de calcul précédent de manière telle qu'il permette au constructeur d'étudier des plans de restructuration dans chaque pays. On connaît la situation pour l'année précédente dans chaque pays, on fixe la proportion (de 0 à 3; 1 signifiant le *statu quo*) de salariés pour cette année par rapport à l'année précédente et on observe l'évolution du bénéfice total.

- 18.10 Les stocks d'une entreprise sont constitués de produits finis et de matières premières. La fabrication d'un produit fini fait intervenir des matières premières, ainsi que certaines machines. Pour chaque produit fini, on connaît (1) la quantité de chaque matière première utilisée pour sa fabrication (2) le temps d'utilisation de chaque machine qui est intervenue dans sa fabrication. Ces informations sont relatives à la fabrication d'une unité du produit fini considéré. On connaît en outre le prix unitaire de chaque matière première. On connaît enfin la quantité en stock de chacun de ces produits et matières, ainsi que le coût d'une heure d'utilisation de chaque machine. On demande la valeur totale des stocks en euros.

Exemple réduit (1 produit fini X, 2 matières premières Y et Z, 1 machine A) : il existe 500 kg du produit fini X, lequel a nécessité, pour la fabrication d'un kg, 0,5 heure de la machine A, 0,2 kg de matière première Y et 0,8 kg de matière première Z. Il y a 350 kg de Y en stock et son coût est de 120 ι/kg, tandis qu'il y a 60 kg de Z, son coût étant de 720 ι/kg. Le coût de l'heure d'utilisation de la machine A est de 2500 ι. La valeur totale des stocks est de 1 010 200 ι.

Suggestion : pour simplifier, on considère que chaque matière première et chaque machine interviennent dans tout produit fini, éventuellement en quantité ou durée nulle.

- 18.11 On considère une entreprise de distribution dont les règles de facturation obéissent aux conventions suivantes, considérées pour une commande d'un client.

Le client possède un compte caractérisé par un montant; il a reçu un certain nombre d'expéditions dont on connaît chaque montant. La commande à facturer est constituée d'un certain nombre de lignes, spécifiant chacune une quantité et un numéro de produit. Chaque produit est caractérisé par son prix unitaire, son taux de TVA et son poids unitaire. La facture d'une commande comporte, outre le prix des produits commandés, TVA comprise, le coût du port. Ce dernier n'est compté qu'à partir d'un kg, selon un tarif proportionnel. L'entreprise consent une réduction de 5 % sur le prix des produits lorsque le total des montants des expéditions passées, augmenté du montant du compte, est au moins égal à 2 000 ι.

- 18.12 On s'intéresse au coût de stockage d'un fichier. Ce coût est proportionnel à la taille du fichier et à sa durée de vie. Dans ce coût interviennent les frais liés à l'unité à disque, ainsi qu'une participation aux frais généraux liés à la totalité de l'installation informatique. Les frais liés à l'unité à disque constituent la part (dépendant du volume et de la durée) du fichier dans le prix d'achat et le coût de maintenance (étant donné une période d'amortissement standard de l'équipement et la capacité de l'unité). Les frais généraux se calculent comme une fraction du total des frais généraux de l'installation. Cette fraction est le rapport entre d'une part les prix d'achat et de maintenance (durant la période d'amortissement) de l'unité et d'autre part les prix d'achat et de maintenance

(même période d'amortissement) de l'installation complète. Quant aux frais généraux relatifs à l'installation complète, ils comprennent le coût de l'assurance, les salaires des employés, les frais de la location des locaux et le coût de l'énergie.

18.13 Avant de lancer un produit, une entreprise étudie le prix à l'unité qu'elle en demandera. Le calcul de ce prix ne semble pas aisé *a priori*; c'est pourquoi on accepte de ne retenir que des lois très simples. En conséquence, on se limitera à étudier le problème de l'évolution de ce prix unitaire en fonction du coût de la campagne publicitaire, du coût de fabrication à l'unité et du bénéfice total que l'on se fixe comme objectif. On retient bien sûr le fait que le prix de revient représente le coût publicitaire plus le coût de fabrication. On admet aussi de façon simpliste que la quantité vendue dépendra exclusivement de l'importance de la campagne publicitaire, celle-ci étant mesurée par son coût. Consultés sur ce point, les experts en marketing donnent les explications suivantes :

- sans publicité aucune, le marché absorbera une quantité Q_0 ;
- quel que soit l'effort publicitaire, le marché ne consommera jamais plus que Q_1 ;
- on admet que la relation entre la quantité vendue Q et le coût de la campagne de publicité P est de la forme suivante :

$$Q = k_1 (2 - 1 / (P + k_2))$$

Proposer un modèle de calcul du prix à l'unité du produit.

Suggestions : on cherchera avant tout un graphe des dépendances sans circuit en choisissant judicieusement les grandeurs internes. Le calcul des paramètres k_1 et k_2 peut avantageusement être encapsulé dans un sous-modèle technique.

18.14 Les résultats obtenus à l'aide du modèle de l'exercice 18.13 sont à ce point encourageants que l'entreprise décide d'affiner l'outil qu'elle a utilisé pour calculer le prix du produit. Elle se propose donc de faire intervenir en outre le fait que le coût de fabrication à l'unité décroît en fonction de la quantité vendue (et donc produite, car on ne considère pas les invendus). Le service des fabrications fournit les renseignements suivants :

- pour une seule pièce, le coût est égal à C_1 ;
- pour une très grande quantité, le coût unitaire ne descendra pas au-dessous de C_2 ;
- on admet que la relation entre le coût unitaire de fabrication et la quantité produite est de la forme suivante :

$$P = k_1 (2 + 1 / (Q + k_2))$$

18.15 Enhardie par les analyses précédentes, l'équipe chargée de l'étude commerciale décide de considérer une donnée supplémentaire du problème. En effet, les premières démarches avaient ignoré le fait que le marché répond non seulement en fonction de la campagne publicitaire, mais aussi du prix de vente auquel le produit est proposé. Cependant, le couplage de ces deux facteurs effraie quelque peu les membres de l'équipe. On décide alors d'ignorer le rôle direct de la campagne publicitaire sur la demande (c'est-à-dire ici sur la quantité qui sera vendue). Son coût sera simplement ajouté au prix de revient global. La relation entre le prix unitaire du produit et la quantité vendue est décrite comme suit :

- si le prix était de P_1 , la demande serait de D_1 ;
- lorsque le prix augmente de p %, alors la demande diminue de d %.

Suggestions : on remarquera que le prix dépend de la quantité et que la quantité dépend du prix. D'autre part, la propriété de variation de la demande D en fonction du prix P peut s'exprimer formellement comme suit :

$$\frac{\Delta P}{P \cdot P} = \frac{\Delta D}{-d \cdot D}$$

En posant $\gamma = -\frac{d}{p}$

il vient $\frac{\Delta D}{\Delta P} = \gamma \frac{D}{P}$

En considérant la fonction $D = D(P)$, et en faisant tendre ΔD et ΔP vers 0, il vient

$$D'(P) = \gamma \frac{D(P)}{P}$$

ainsi que $\frac{D'(P)}{D(P)} = \gamma \frac{1}{P}$

Sachant que la primitive de $1/X$ est $\ln(X)$ et que la primitive de $f'(X)/f(X)$ est $\ln(f(X))$, on obtient aussi

$$\ln(D) = \gamma \ln(P) + k$$

Il reste à expliciter D en fonction de P et à calculer la constante d'intégration k .

18.16 Pourrait-on établir un modèle qui envisage à la fois l'influence de la **campagne publicitaire** et celle du **prix unitaire** sur le volume des ventes ?

18.17 On reprend l'étude de cas n° 2, dont on étend la portée comme suit :

1. La charge peut varier d'une escale à l'autre.
2. Le carburant est acheté en devises locales (FFR, PFR, \$), au tarif local (en unité de devise par kilo). Le tarif local dépend du jour, de même que le cours des différentes devises.

3. On considère le cas d'un appareil qui effectue plusieurs vols consécutifs. La quantité résiduelle à l'escale d'arrivée d'un vol est la quantité résiduelle de départ du vol suivant.

18.18 A la fin de chaque mois les membres d'une famille se répartissent les rentrées financières mensuelles, après en avoir retiré les dépenses et un certain montant d'épargne. Le montant reçu par un parent est triple de celui que reçoit chaque enfant. Les rentrées sont constituées des allocations familiales (fonction du nombre d'enfants) et du revenu net de chacun des parents (il y a 1 ou 2 conjoints). S'il y a deux revenus, il faut en outre considérer une retenue fiscale supplémentaire due au cumul. Celle-ci dépend du nombre de personnes du ménage et de la somme des deux revenus nets. L'épargne est égale à un certain pourcentage du solde disponible après déduction de la retenue de cumul et des dépenses. Ces dernières se répartissent en dépenses fixes, indépendantes des personnes, et en dépenses individuelles faites par les membres de la famille. Pour chaque jour du mois, on connaît le total des dépenses fixes, ainsi que le total des dépenses individuelles de chaque personne.

On désire connaître le montant reçu par chaque personne à la fin du mois (on ne considère que le mois courant).

18.19 On considère un véhicule automobile régulièrement emprunté par N personnes qui, pour chaque voyage qu'elles effectuent, se partagent les frais de celui-ci. Le coût d'un voyage (on connaît la distance parcourue et le temps, la vitesse étant considérée comme constante) couvre la consommation de carburant, les frais annexes l'amortissement du prix d'achat relatif aux kilomètres parcourus. Le prix du carburant est connu. La consommation (en litres aux 100 km) obéit à la loi exponentielle $C = a \cdot 10^{b \cdot v}$. En outre, des essais préalables ont permis de prendre les mesures de consommation suivantes :

à 60 km/h, consommation = C_{60} l/km

à 90 km/h, consommation = C_{90} l/km

à 120 km/h, consommation = C_{120} l/km

On connaît le prix d'achat du véhicule, qui doit être amorti linéairement en 6 ans, à raison d'une distance annuelle moyenne fixée. On connaît également les frais annexes dont une partie est annuelle et l'autre est rapportée au km parcouru.

On désire connaître la part de chaque personne pour un voyage déterminé.

18.20 Une personne désire acheter une voiture. Elle dispose de la somme nécessaire. Cependant, elle hésite entre trois formules de paiement. La première consiste à payer la voiture au comptant. La deuxième est un financement qui lui est proposé par le vendeur : chaque mois, durant une période déterminée, l'acheteur paie un pourcentage déterminé de la somme

empruntée. La troisième formule lui est proposée par un organisme bancaire : le prêt est consenti à un taux déterminé et le remboursement se fait par mensualités constantes, calculées selon la formule ci-dessous, durant une période déterminée. Dans les deux cas d'emprunt, un acompte identique doit être payé lors de la réception du véhicule. Sachant qu'en cas d'emprunt la somme non payée dont dispose l'acheteur peut être déposée en banque à un taux mensuel donné, mais que l'on y retirera également chaque mois le montant du remboursement mensuel, on se demande laquelle des trois formules est la meilleure pour l'acheteur. On peut adopter la règle :

$$\text{Mensualité} = (\text{SommeEmpruntée} * \text{Taux}) / (1 - (1 + \text{Taux})^{-\text{Durée}})$$

- 18.21 On considère un ensemble de clients et un ensemble d'usines. On considère également un (unique) type de matériau que fabriquent les usines et que demandent les clients. Chaque client a besoin d'une certaine quantité quotidienne de matériau (DEMANDE), et chaque usine peut produire chaque jour une certaine quantité de ce matériau (CAPACITE). Entre chaque usine et chaque client il existe une voie de transport caractérisée par sa longueur. On connaît le coût de transport au kilomètre d'une unité de matériau.

D'autre part, on connaît aussi le prix de vente unitaire (départ usine) du matériau pour chaque usine. Pour satisfaire ses besoins, chaque client se fait envoyer quotidiennement, de chaque usine (en fait de certaines d'entre elles), *via* la voie de transport qui les relie, une quantité déterminée de matériau. On admet de représenter par une quantité nulle le fait qu'un client déterminé ne demande rien à une usine déterminée. Chaque usine consent à ses clients une remise de 5 % pour des envois quotidiens dépassant une quantité minimum déterminée (celle-ci dépend de chaque usine). Chez chaque client, le déchargement d'une livraison réelle (c'est-à-dire de quantité non nulle) venant d'une usine est comptabilisé à un coût constant déterminé, quelle que soit l'usine de départ. Ce coût dépend du client.

On admet que toutes les quantités et les coûts sont invariables dans le temps, c'est-à-dire que la situation est identique d'un jour à l'autre.

On demande le coût total par jour pour l'ensemble des clients. On veillera à déterminer les contraintes de validité du modèle.

On résoudra ce problème par un modèle de calcul pur, puis par un modèle étendu à une base de données.

18.6.3 Validation de modèles

- 18.22 Étudier la validité de chacun des quatre fragments suivants :

- $Y = \arcsin(2 + X^2)$
- $X = \sin(Z) - 3$
 $Y = \ln(X)$
- $X = \arcsin(Z) - 3$
 $Z^2 > 1$

$$\bullet A = 2 - 1/(B-3)$$

$$B = A + 1$$

18.23 Déterminer si ce modèle est correct. Sinon, proposer les corrections nécessaires.

Données

$M : m_1..m_n$; membres du club
 COTISATION : entier ; cotisation annuelle
 APPORT_M : entier (F) ; apport financier de chaque membre
 DEPENSES : entier (F) ; dépenses du club durant l'année

Résultat

PRIME_M : entier (F) ; prime revenant à chaque membre

Grandeurs internes

ENTREE_M : entier (F) ; entrée de chaque membre après déduction de la cotisation
 SOLDE : entier (F) ; solde restant en caisse en fin d'exercice
 PRIME_M : réel ; solde restitué à chaque membre au prorata de sa contribution

Règles

$ENTREE_M = APPORT_M - COTISATION$
 $ENTREES = \sum_M ENTREE_M$
 $SOLDE = ENTREES - DEPENSES$
 $PRIME_M = SOLDE * (ENTREE_M / ENTREES)$
 $COTISATION \geq 0$
 $DEPENSES \geq 0$

18.24 Déterminer si le modèle ci-dessous est correct. Sinon, proposer les corrections nécessaires.

Données

A : réel

Résultat

D : réel

Grandeurs internes

B, C : réel

Règles

$B = A^2$
 $C = 2 * A$
 $D = \sqrt{B^2 - C^2}$
 $5 \leq A \leq 10$

18.6.4 ... et en guise de dessert

18.25 Une corde pend de part et d'autre d'une palissade de sorte que les deux parties sont d'égale longueur. Elle pèse $\frac{1}{3}$ de livre par pied de longueur. A une extrémité est suspendu un chimpanzé tenant une banane. A l'autre extrémité est attachée une masse dont le poids est celui du chimpanzé. La banane pèse deux onces par pouce. La corde est aussi longue (en pieds) que le chimpanzé est âgé (en années) et le poids du chimpanzé (en onces) est de même valeur que l'âge de sa mère. Ensemble, le chimpanzé et sa mère totalisent 30 ans d'âge. La moitié du poids du chimpanzé, ajoutée à celui de la banane, est égale à $\frac{1}{4}$ du poids de la masse et de la corde réunis. La mère du chimpanzé a la moitié de l'âge qu'aura le chimpanzé quand il aura 3 fois l'âge qu'avait sa mère quand elle avait la moitié de l'âge que le chimpanzé aura au moment où il aura l'âge de sa mère quand elle aura 4 fois l'âge du chimpanzé quand il avait 2 fois l'âge de sa mère quand elle avait $\frac{1}{3}$ de l'âge du chimpanzé au moment où il avait l'âge que sa mère avait quand elle avait 3 fois l'âge du chimpanzé quand il avait $\frac{1}{4}$ de l'âge qu'il a maintenant¹⁰.

Quelle est la longueur de la banane ?

Note : 1 pied = 30,5 cm, 1 pouce = 2,54 cm, 1 livre = 453 g, 1 once = 28,35 g.

Suggestion : la relation entre les âges du chimpanzé et de sa mère s'exprimera aisément si on la fractionne selon chaque âge cité dans la phrase. On simplifiera éventuellement ce fragment du modèle (par contraction) pour obtenir une simple relation linéaire.

10. Librement inspiré de : Fixx, J. *Games for the Super-intelligent*, Doubleday, Garden City, New York, 1972.

Chapitre 19

Implantation d'un modèle dans une feuille de calcul

L'implantation d'un modèle dans une feuille de calcul s'effectue en deux phases. La première est celle du choix de la position et du format de chaque grandeur dans la feuille de calcul par l'élaboration de la maquette du tableau. La seconde consiste à traduire les règles en expressions acceptées par le tableur. On propose également un procédé de traduction de modèles dans un langage algorithmique.

19.1 ÉLABORATION D'UNE MAQUETTE

Durant cette phase, on choisit la position des composants du modèle dans la feuille électronique. Idéalement, ce travail s'effectue sur une feuille de papier quadrillé ou dans une feuille de calcul dans laquelle on se contente d'indiquer l'emplacement des grandeurs et les libellés. On choisira généralement de regrouper les données (par exemple en haut à gauche), afin d'en faciliter la saisie, ainsi que les résultats (par exemple en bas à droite), afin d'en rendre la lecture plus aisée. On choisira soigneusement les titres, les encadrements, les couleurs, les commentaires, les unités, ainsi que toute information textuelle qui permettra à l'utilisateur de se servir du tableau en toute sécurité.

Nous présenterons quelques règles intuitives de disposition et de traduction des principaux concepts qui réclament une décision de localisation : les grandeurs dimensionnées, les grandeurs internes et les sous-modèles.

19.1.1 Représentation des grandeurs dimensionnées

Les grandeurs dimensionnées seront le plus souvent présentées sous une forme tabulaire. Ainsi, si plusieurs grandeurs dépendent d'une même dimension, on assignera à celles-ci des colonnes (ou lignes) adjacentes, chaque ligne (colonne) correspondant à une valeur de la dimension.

La maquette de la figure 19.1 représente une implantation des grandeurs suivantes :

MOIS : m_1, m_2, \dots, m_{12}
ANCIEN_{MOIS}
NIVEAU_{MOIS}
...
BRUT_{MOIS}
...
NET_{MOIS}

MOIS	ANCIEN	NIVEAU	...	BRUT	...	NET
m_1	$ancien_1$	$niveau_1$...	$brut_1$...	net_1
m_2	$ancien_2$	$niveau_2$...	$brut_2$...	net_2
...
m_{12}	$ancien_{12}$	$niveau_{12}$...	$brut_{12}$...	net_{12}

Figure 19.1 - Implantation de plusieurs grandeurs dépendant d'une dimension (MOIS)

Si une grandeur est fonction de deux dimensions, on lui consacrera une plage dont les colonnes correspondent aux valeurs d'une dimension et les lignes aux valeurs de l'autre dimension (figure 19.2).

MOIS : m_1, m_2, \dots, m_{12}
EMPLOYE : e_1, e_2, \dots, e_{80}
NET_{MOIS, EMPLOYE}

NET	e_1	e_2	...	e_{80}
m_1	$net_{1,1}$	$net_{2,1}$...	$net_{80,1}$
m_2	$net_{1,2}$	$net_{2,2}$...	$net_{80,2}$
...
m_{12}	$net_{1,12}$	$net_{2,12}$...	$net_{80,12}$

Figure 19.2 - Implantation d'une grandeur dépendant de deux dimensions

Si plusieurs grandeurs dépendent de deux dimensions, on peut proposer trois implantations typiques (parmi d'autres).

La première technique consiste à représenter chaque grandeur indépendamment des autres sous la forme d'un tableau comme suggéré à la figure 19.2. La figure 19.3 montre que ces tableaux peuvent être alignés de manière à permettre une lecture transversale selon la même valeur de MOIS. D'ailleurs, les valeurs de cette dimen-

sion (MOIS) peuvent n'apparaître que dans le tableau de la première grandeur (BRUT), afin de gagner de la place dans les autres tableaux.

MOIS : m_1, m_2, \dots, m_{12}
 EMPLOYE : e_1, e_2, \dots, e_{80}
 BRUT_{MOIS, EMPLOYE}
 NET_{MOIS, EMPLOYE}

BRUT	e_1	...	e_{80}	NET	e_1	...	e_{80}
m_1	$net_{1,1}$...	$net_{80,1}$	m_1	$br_{1,1}$...	$br_{80,1}$
m_2	$net_{1,2}$...	$net_{80,2}$	m_2	$br_{1,2}$...	$br_{80,2}$
...
m_{12}	$net_{1,12}$...	$net_{80,12}$	m_{12}	$br_{1,12}$...	$br_{80,12}$

Figure 19.3 - Implantation de plusieurs grandeurs dépendant de deux dimensions (1)

Selon la deuxième technique (figure 19.4), c'est chaque valeur de l'une des dimensions (MOIS) qui fait l'objet d'un tableau isolé. Dans chaque tableau, les valeurs de chaque grandeur sont présentées selon toutes les valeurs de l'autre dimension (EMPLOYE).

MOIS : m_1, m_2, \dots, m_{12}
 EMPLOYE : e_1, e_2, \dots, e_{80}
 BRUT_{MOIS, EMPLOYE}
 NET_{MOIS, EMPLOYE}

mois m_1	BRUT _{m_1}	NET _{m_1}	mois m_2	BRUT _{m_2}	NET _{m_2}
e_1	$br_{1,1}$	$net_{1,1}$	e_1	$br_{1,2}$	$net_{1,2}$
e_2	$br_{2,1}$	$net_{2,1}$	e_2	$br_{2,2}$	$net_{2,2}$
...
e_{80}	$br_{80,1}$	$net_{80,1}$	e_{80}	$br_{80,2}$	$net_{80,2}$

Figure 19.4 - Implantation de plusieurs grandeurs dépendant de deux dimensions (2)

La troisième technique (figure 19.5) regroupe toutes les valeurs en un seul tableau, grâce à une combinaison hiérarchique selon l'une des directions, ou même les deux. Cette disposition est très générale et permet de prendre en charge un nombre quelconque de dimensions et de grandeurs¹.

Les tableaux multifeuille offrent une représentation immédiate de grandeurs à trois dimensions : deux dimensions sont représentées comme ci-dessus et chaque feuille correspond à une valeur de la troisième dimension.

1. C'est l'occasion de mentionner une fonctionnalité de présentation proposée initialement par IMPROV (Lotus, 1992) et adoptée depuis par d'autres tableurs : le pivotement. Elle consiste à restructurer dynamiquement un tel tableau en déplaçant les en-têtes de lignes et colonnes de manière à présenter les données selon des vues différentes.

		e_1	e_2	...	e_{80}
m_1	NET	$net_{1,1}$	$net_{1,2}$...	$net_{1,80}$
	BRUT	$br_{1,1}$	$br_{1,2}$		$br_{1,80}$
m_2	NET	$net_{2,1}$	$net_{2,2}$...	$net_{2,80}$
	BRUT	$br_{2,1}$	$br_{2,2}$		$br_{2,80}$
...	
m_{12}	NET	$net_{12,1}$	$net_{12,2}$...	$net_{12,80}$
	BRUT	$br_{12,1}$	$br_{12,2}$		$br_{12,80}$

Figure 19.5 - Implantation de plusieurs grandeurs dépendant de deux dimensions ou plus (3)

D'autres techniques existent, offertes par l'un ou l'autre tableur. EXCEL par exemple propose deux fonctionnalités directement utilisables dans certains cas de figure se présentant fréquemment : tables de données et scénarios.

- Les **tables de données** (section 15.9) offrent une représentation particulièrement concise de fragments de modèles qui ont soit une donnée dimensionnée selon I et N résultats dimensionnés selon I, soit deux données dimensionnées, l'une selon I et l'autre selon J, et un résultat dimensionné selon I et J. L'intérêt de cette technique est qu'elle ne réclame qu'un seul exemplaire de chaque règle de définition (leur dimensionnement est implicite, comme le montrent les figures 15.6 et 15.7).
- Les **scénarios** (section 15.10) peuvent également exprimer facilement des données qui sont toutes dimensionnées selon I. En effet, soient A_I, B_I, \dots, F_I les données du (fragment de) modèle. Si la dimension I ne comporte pas trop de valeurs, il est possible de définir, pour chaque valeur de I, un scénario qui contient une valeur de chaque donnée. On définira ainsi les scénarios $\{A_1, B_1, \dots, F_1\}$, $\{A_2, B_2, \dots, F_2\}$, etc. Le modèle sera appliqué sur chacun de ces scénarios, soit manuellement, soit en enchaînement automatique. Tout comme dans la technique précédente, les règles ne sont pas dimensionnées selon I, ce qui conduit à un modèle très simple.

19.1.2 Les grandeurs internes

Les grandeurs internes correspondent à des concepts qu'on désirera généralement cacher. Si c'est le cas, cela se fera soit par la contraction du modèle, qui élimine des grandeurs internes², soit en éloignant les cellules correspondantes de la partie visible du tableau, soit en usant de jeux de caractères, couleurs, largeurs de colonne ou formats qui en diminuent la visibilité, soit encore en donnant à ces cellules le statut invisible.

2. La contraction, la démodularisation et l'éclatement sont des transformations de modèles. Celles-ci sont décrites dans un document disponible sur le site de l'ouvrage.

La traduction d'un sous-modèle, et plus généralement d'un modèle, sous la forme d'une fonction programmée (fonction macro d'EXCEL par exemple) offre une encapsulation idéale de la description interne du modèle, puisque celle-ci est implantée dans une feuille spéciale (feuille macro) qui est invisible lors de l'exécution. On rappelle cependant que ce qu'on gagne en lisibilité des résultats à l'écran est perdu en qualité de structure du modèle, puisque celui-ci est désormais réalisé par un algorithme.

19.1.3 Les sous-modèles

En règle générale, on cherchera à cacher les concepts des sous-modèles. La représentation des sous-modèles n'est pas toujours immédiate. Contrairement aux langages de troisième génération³, les tableurs n'offrent pas toujours des concepts permettant leur traduction directe. Nous proposerons quelques techniques qui systématisent cette représentation.

- *Dans tous les cas.*

La technique la plus simple, mais la moins élégante et souvent la plus coûteuse en mémoire, consiste à remplacer chaque invocation d'un sous-modèle par le développement de ce dernier (transformation de *dém modularisation*). On fait donc disparaître les sous-modèles en tant que tels. Les grandeurs internes des sous-modèles deviennent des grandeurs internes du modèle invoquant; elles seront donc traitées comme indiqué ci-dessus. Ainsi, dans le modèle de la figure 17.14, on remplace l'expression :

$$\text{COTISATION_SOCIALE}_T = \text{calcul_COTISATION}(\text{NIVEAU}_T, \text{BRUT}_T)$$

par la suivante :

$$\text{COTISATION_SOCIALE}_T = (0,1 + \text{NIVEAU}_T/100) * \text{BRUT}_T$$

- *Le sous-modèle est invoqué une seule fois, c'est-à-dire à un seul endroit, et sur des données non dimensionnées (cf. modèle de la figure 17.13).*

Le sous-modèle est implanté dans une plage isolée hors de la vue de l'utilisateur. Il dispose de cellules propres représentant ses données, ses résultats et ses grandeurs internes.

1. La cellule représentant une donnée du sous-modèle contient une formule qui y range le contenu de la cellule correspondant à l'argument de son invocation. Par exemple, la cellule de la donnée ANC du sous-modèle calcul_BRUT contient la formule « = α », où α est l'adresse de la cellule de la grandeur ANCIENNETE du modèle TRAITEMENT.
2. Inversement, la cellule représentant une grandeur du modèle invoquant qui est définie par le sous-modèle (c'est-à-dire correspondant à un résultat de celui-ci) contiendra une formule qui y range le contenu de la cellule du

3. Via les notions de fonction et procédure par exemple, comme on les trouve en C, Pascal, BASIC et Java.

résultat du sous-modèle. Par exemple, la grandeur BRUT_DE_BASE du modèle TRAITEMENT sera représentée par une cellule qui contient la formule « = β », où β est l'adresse de la cellule du résultat BRUT_D_B du sous-modèle calcul_BRUT.

- *Le sous-modèle est invoqué plus d'une fois, c'est-à-dire à plusieurs endroits, ou sur des données dimensionnées (cf. modèle de la figure 17.14).*
Tous les tableurs classiques ne disposent généralement pas encore de mécanismes, analogues à celui des procédures dans les langages de troisième génération, permettant de représenter immédiatement les invocations successives d'un sous-modèle pour une liste de données.
Il est cependant possible de pallier cette lacune lorsque le tableur offre la possibilité de définir des macros. En particulier, EXCEL propose la notion de *fonction personnalisée* ou fonction macro. Une telle fonction, qui est à définir par l'utilisateur, peut être invoquée comme une fonction intrinsèque classique. Une fonction macro peut recevoir des arguments en entrée (= données) et renvoie une valeur (= résultat).
Lorsque le sous-modèle définit plus d'une grandeur résultat, la technique de la fonction n'est plus adéquate, car cette dernière ne renvoie qu'une seule grandeur, ou au mieux un ensemble correspondant à une plage de cellules. On a cependant vu qu'il était toujours possible de remplacer une fonction à N résultats par N fonctions à un résultat, grâce à une transformation *d'éclatement*. Si le sous-modèle n'est pas trop complexe, cette technique permet une implantation élégante, bien qu'elle puisse présenter d'importantes redondances dans les définitions.

19.1.4 Exemple de maquette

La figure 19.6 représente une maquette possible de l'implantation du modèle 17.9. La colonne Niv. OK indique si la valeur de Niveau respecte la condition de validité. La cellule contenant tok indique si toutes les valeurs de Niv. OK sont à VRAI.

Traitements d'un employé										
Mois	Anc.	Niveau	Niv. OK	Primes	Index	Brut	Cot. Soc.	Net Impos.	Ret. Fisc.	Net Payé
m1	a1	n1	ok1	p1	i1	b1	cs1	ni1	rf1	np1
m2	a2	n2	ok2	p2	i2	b2	cs2	ni2	rf2	np2
m3	a3	n3	ok3	p3	i3	b3	cs3	ni3	rf3	np3
m4	a4	n4	ok4	p4	i4	b4	cs4	ni4	rf4	np4
m5	a5	n5	ok5	p5	i5	b5	cs5	ni5	rf5	np5
m6	a6	n6	ok6	p6	i6	b6	cs6	ni6	rf6	np6
m7	a7	n7	ok7	p7	i7	b7	cs7	ni7	rf7	np7
m8	a8	n8	ok8	p8	i8	b8	cs8	ni8	rf8	np8

Niveau OK ?

tok

Tot. cotis.

tcs

Tot. ret.

tr

Figure 19.6 - Une maquette du modèle 17.9

19.1.5 Ergonomie des modèles

Un modèle est un outil de travail pour ses utilisateurs. A ce titre, il est logique qu'on le soumette à des règles d'ergonomie afin de le rendre facile à utiliser. Cette qualité ne relève pas (seulement) de la philanthropie, mais plutôt de la sécurité : si le modèle qui apparaît à l'écran est clair, lisible, structuré logiquement en fonction du mode naturel de travail de son utilisateur, il gagnera en fiabilité. On cherchera à présenter les informations par agrégats logiques en fonction de leur nécessité et de manière telle que les relations entre ces informations apparaissent clairement.

Nous analyserons deux règles intuitives qui doivent guider la présentation des composants d'un modèle complexe. La première concerne l'unité de travail de l'utilisateur, la seconde la classification des informations.

a) Unité de travail

Considérons à nouveau le modèle de la figure 17.10, qui décrit les revenus d'un ensemble d'employés pendant un certain nombre de mois. Ce modèle est neutre vis-à-vis de l'utilisation qu'on pourrait en faire. Il convient parfaitement à l'utilisateur qui désirerait étudier l'ensemble des employés pour un mois déterminé, par exemple pour établir les salaires du mois courant. Il est aussi adapté aux besoins de l'utilisateur qui se concentre sur l'historique d'un employé déterminé, et pour lequel il désire consulter la liste des informations mensuelles pour les mois écoulés. La maquette, cependant, traduit l'une ou l'autre de ces perspectives, mais ne peut en toute généralité favoriser les deux dès qu'on est confronté à des modèles complexes. Une maquette est structurée en fonction de l'usage qu'on prévoit de faire du modèle. Si on prévoit plus d'un usage, et que chacun d'eux exige une présentation spécifique, il faudra définir plusieurs maquettes qui représentent le modèle abstrait, en tout ou en partie. On rappelle que les tableurs modernes admettent qu'on définisse plusieurs vues du même modèle, et que le problème qu'on pose ici peut se traduire aisément en EXCEL (par exemple) sous la forme d'une feuille de calcul contenant le modèle complet et d'un jeu de feuilles pour chaque vue distincte de ce modèle (section 15.3). Pour des raisons de facilité de manipulation, on pourra organiser la feuille de manière telle qu'un écran contienne la totalité des informations sur lesquelles l'utilisateur travaille. Il est en effet très facile de sauter d'un écran à l'autre dans les quatre directions.

Plus précisément, ce problème peut se poser en termes d'**unité de travail**⁴. On désigne sous ce terme le concept autour duquel s'organise le travail de l'utilisateur. La discussion qui précède suggère deux unités de travail distinctes : le *mois* d'une part, ce qui conduit à une feuille de calcul par mois, et l'*employé* d'autre part, ce qui suggère une feuille de calcul par employé. On pourrait envisager une unité plus fine encore, comme l'*employé durant un mois*, ce qui conduirait à une feuille par employé et par mois. Cette structuration du modèle est basée sur ses dimensions :

4. Ce concept est proche des *objets métier* de Merise et autres *business objects*.

l'unité de travail correspond à une dimension (ou une combinaison de dimensions) qu'on privilégie au détriment des autres.

b) Classification des informations

On envisagera une autre décomposition des informations visibles (données et résultats), non plus selon les dimensions du modèle, mais selon le rôle que jouent ces informations. Nous citerons trois exemples de critères de classification :

- *Le cycle de vie des informations.* Certaines informations sont très stables et ne sont modifiées que très rarement, alors que d'autres sont modifiées fréquemment. On peut proposer de localiser ces grandeurs dans des emplacements (feuilles par exemple) distincts.
- *Le contrôle de l'utilisateur sur les informations.* L'utilisateur n'a aucun contrôle sur le cours des devises. En revanche, c'est à lui que revient la décision d'acheter ou de vendre des titres. Il peut être opportun de classer ces informations de manière distincte. Remarquons que selon ce critère, modifier les données de l'une ou de l'autre de ces catégories relève respectivement de la *simulation* et de la *décision*.
- *Les facettes des informations.* Les informations d'un modèle peuvent relever d'aspects différents, mais complémentaires du domaine d'application, de telle manière qu'un utilisateur déterminé soit intéressé par l'un seulement de ces aspects. Il est inutile de présenter à cet utilisateur les informations qui ne le concernent pas.

19.2 TRADUCTION DES RÈGLES

19.2.1 Principes généraux

Le principe du tableur est d'assigner une cellule à chaque grandeur retenue dans le modèle. Il faut alors traduire les règles selon la syntaxe du langage du tableur. Comme dans une formule une grandeur est désignée par l'adresse de sa cellule, il est pratique de se construire un dictionnaire des grandeurs qui indique l'adresse à laquelle chacune est implantée. Ainsi traduite, une règle est devenue exécutable, mais aussi difficilement lisible. Les tableurs offrent cependant la possibilité d'assigner un nom à une cellule ou à une plage. Il est alors permis d'utiliser ce nom pour désigner la cellule ou la plage. L'expression des formules de calcul est plus fiable et plus lisible.

Ainsi, la règle $TOTAL_RETENUE_E = \sum_T RETENUE_FISCALE_T$ sera représentée

- soit par le stockage de l'expression `somme(I4..I7)` dans la cellule I9,
- soit, si le nom `TOTAL_RETENUE_E` a été attribué à la cellule I9 et le nom `RETENUE_FISCALE` à la plage I4..I7, par le stockage de l'expression `somme(RETENUE_FISCALE)` dans la cellule `TOTAL_RETENUE_E`.

Si une expression fait appel à une fonction qui n'est pas disponible dans le tableur choisi, on simulera celle-ci par un sous-modèle, comme suggéré par la figure 17.15 pour la fonction agrégative OU_T .

Si le tableau ainsi constitué doit être utilisé par d'autres personnes que son auteur, il sera prudent de protéger les formules et les libellés contre les modifications intempestives.

19.2.2 Grandeurs à définition multiple

La formule de définition d'une telle grandeur s'exprime immédiatement par une fonction *si*. La règle ci-dessous est extraite de la figure 17.4. On donne la formule EXCEL (disposée en plusieurs lignes pour des raisons de lisibilité) à ranger dans la cellule MONTANT.

```

MONTANT =      0          si non PRET_ACCORDE
               5 x SALAIRE si PRET_MODERE
               10 x SALAIRE si PRET_ELEVE

= si(non(PRET.ACCORDE);
    0;
    si(PRET.MODERE;
        5*SALAIRE;
        si(PRET.ELEVE;
            10*SALAIRE;
            #N/A)))

```

Lorsqu'on a prouvé que la règle était complète, la dernière branche de la formule ne sera jamais réalisée. En effet,

$PRET_ELEVE = \text{non}((\text{non } PRET_ACCORDE) \text{ ou } PRET_MODERE).$

La règle et la formule peuvent être réécrites plus simplement comme suit :

```

MONTANT =      0          si non PRET_ACCORDE
               5 x SALAIRE si PRET_MODERE
               10 x SALAIRE sinon

= si(non PRET.ACCORDE);
    0;
    si(PRET.MODERE;
        5*SALAIRE;
        10*SALAIRE))

```

19.2.3 Règles de récurrence et règles récursives

Des *règles de récurrence* peuvent se représenter selon deux formats. Le premier format montre explicitement chaque valeur de la série. Les expressions se placeront en colonne ou en ligne comme pour les grandeurs dimensionnées ordinaires. Le second format (cité pour mémoire) correspond à un mode d'utilisation dans lequel on ne désire voir qu'une seule valeur de la série à la fois, le calcul de la valeur

suivante étant commandé par une nouvelle activation du modèle. Cette représentation est semblable à celle des règles récursives que nous allons examiner.

Une *règle récursive* (ou plus généralement un ensemble de règles récursives) peut se développer sous la forme de règles de récurrence comme décrit à la section 17.11. Il est cependant possible de donner aux formules de calcul une forme qui est proche de l'expression utilisée dans le modèle. L'évaluation est alors réalisée de manière itérative, par l'exécution des formules soit un certain nombre de fois, soit jusqu'à ce qu'une condition de convergence soit réalisée.

Rappelons également que certains tableurs possèdent un résolveur d'équations, tel que le *solveur* d'EXCEL, qui peut être utilisé pour résoudre des fragments de modèles qui se présentent comme des systèmes d'équations, éventuellement sous contraintes (section 17.12).

Considérons à titre d'exemple un système de calcul du montant net N tel que la retenue fiscale R est calculée comme une proportion p, non pas du montant brut B, mais du net. Le modèle, déjà présenté en 17.11, est le suivant :

$$\begin{aligned} R &= p * N \\ N &= B - R \end{aligned}$$

Lors de l'implantation dans la feuille électronique, on attribue une cellule aux grandeurs B, p, R et N, ces deux dernières étant définies par des formules de calcul dérivées des règles ci-dessus. A chaque exécution du modèle, les deux expressions sont recalculées. On observe une convergence rapide vers les valeurs présentées en 17.11. En fonction de la puissance du tableur, et si ce dernier accepte de telles formules (qui constituent ce qu'on appelle des **références circulaires** – voir section 15.7) le pilotage des itérations se fera selon l'une des trois méthodes suivantes :

- commande manuelle de chaque évaluation;
- contrôle programmé (macros);
- contrôle automatique par le tableur (mode de calcul itératif dans EXCEL).

19.2.4 Les contraintes

Ces règles ont le plus souvent pour objectif de vérifier la validité des données introduites au moment de l'exécution du modèle, ou de surveiller le comportement de modèles complexes (voir figure 18.2). En cas d'introduction d'une donnée erronée, le tableur doit réagir de manière perceptible. Il convient donc de traduire ces contraintes de manière qu'elles détectent les anomalies et qu'elles déclenchent le cas échéant un signal visuel ou sonore.

On suggère par exemple les principes de traduction suivants :

- pour toute contrainte C_i , réserver une cellule visible V_i qui contient l'expression

$$\text{si}(C_i; \text{vrai}; \text{faux}) \text{ ou } \text{si}(C_i; 1; 0) \text{ ou simplement } C_i$$

- regrouper ces cellules de manière à centraliser visuellement la validation des données,
- réserver une cellule qui affiche un indicateur global signalant s'il existe ou non au moins une erreur; son contenu pourrait être :

et (v_1, v_2, \dots, v_n)

Les branches vrai et faux seront généralement remplacées par des messages plus explicites, ou par l'exécution d'actions d'alerte.

19.3 SÉQUENTIALISATION D'UN MODÈLE

Lors des discussions concernant la construction d'un modèle, il n'a jamais été fait mention d'un quelconque ordre de présentation des règles. En effet, tant sur le plan du raisonnement chez le concepteur, que sur celui du mode d'évaluation d'un modèle par un tableur, l'ordre des règles est parfaitement indifférent. Nous avons par exemple raisonné en partant des résultats pour remonter vers les données, alors qu'on pourrait objecter que l'ordre de calcul des grandeurs sera probablement inverse. En outre, un tableur effectuera le calcul de manière à n'évaluer une formule que lorsque toutes les grandeurs qui y interviennent en partie droite auront déjà été évaluées. Pour ce faire, le tableur construit un graphe de dépendance qui lui indique quelles cellules (et donc quelles formules) doivent être évaluées et dans quel ordre. Cependant, un tableur peut aussi effectuer l'évaluation d'une manière différente : ligne par ligne ou colonne par colonne. Dans certains cas, le mode de calcul peut être choisi par l'utilisateur. Si en revanche on ne dispose pas d'un tableur, ou si le problème réclame des moyens plus puissants, il faudra rédiger un programme de calcul dans un langage de programmation algorithmique tel que Pascal, C, FORTRAN ou BASIC, c'est-à-dire qu'il faudra imposer *a priori* un ordre séquentiel des calculs. Tel sera aussi le cas des procédures et des fonctions (macros) qu'on intègre dans une feuille de calcul (ou feuille macro). Leur corps n'est en effet plus déclaratif, comme le serait un modèle traditionnel, mais bien procédural comme dans n'importe quel langage algorithmique (section 15.11).

En résumé, on peut retenir quatre circonstances dans lesquelles on ne peut malheureusement pas ignorer l'ordre de calcul des formules :

- le tableur est primitif et incapable de trouver l'ordre logique de calcul (tel était le cas de VisiCalc, l'ancêtre des tableurs actuels);
- le tableur est plus récent, mais l'ordre d'évaluation standard n'est pas l'ordre désiré;
- le modèle se présente sous la forme d'une procédure ou d'une fonction;
- le modèle doit être réalisé à l'aide d'un langage algorithmique.

Il est donc indispensable dans ces circonstances de **séquentialiser** les formules de manière à imposer l'ordre d'évaluation logique. Nous examinerons une procédure qui produit simplement une liste des grandeurs internes et des résultats dont l'ordre

respecte la logique de l'évaluation de leurs formules. Nous n'envisagerons pas le cas des modèles récurifs.

Cette procédure est basée sur le graphe de dépendance du modèle. Redessignons le graphe d'une manière telle que les flèches respectent les deux principes suivants :

- une flèche est rectiligne;
- une flèche est strictement orientée vers le bas.

De cette manière, on ne peut, en suivant les flèches, que **descendre** des données vers les résultats. En outre, si une grandeur doit être calculée avant une autre, elle est située au-dessus de cette dernière⁵. Il suffit alors de projeter les grandeurs sur un axe vertical pour obtenir une liste respectant l'ordre logique d'évaluation⁶. Si deux grandeurs sont projetées au même endroit, leur ordre est indifférent et peut être choisi arbitrairement. La figure 19.7 montre le résultat de la séquentialisation du modèle de la figure 17.3.

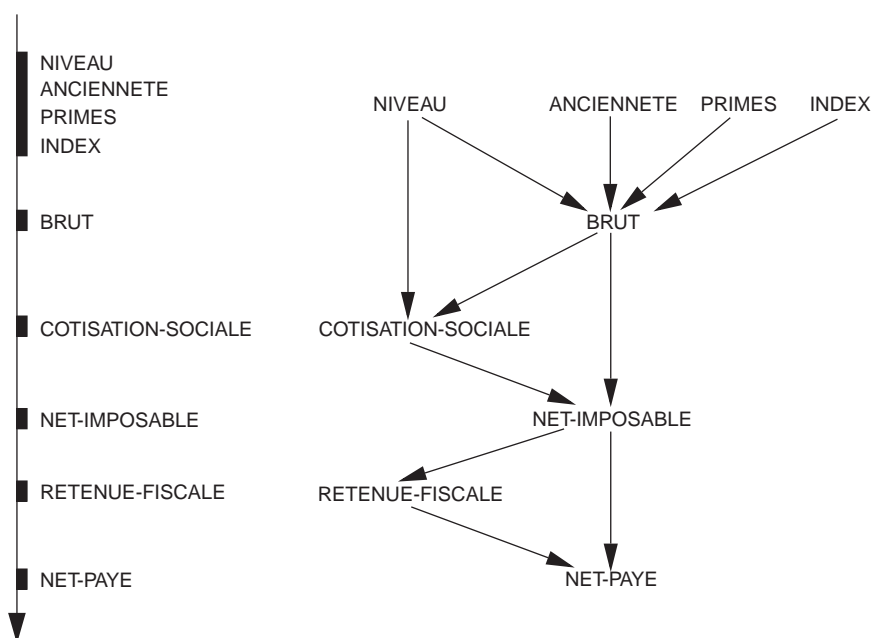


Figure 19.7 - Séquentialisation du modèle 17.3

Les grandeurs {ANCIENNETE, NIVEAU, PRIMES, INDEX} forment le premier lot. Cela implique qu'il faut saisir (ou calculer, s'il ne s'agissait pas que de données) leurs valeurs avant toute autre chose. Le deuxième lot est constitué de {BRUT}, indiquant qu'il faut ensuite calculer la valeur du BRUT. Le raisonnement se poursuit

5. Cela n'est possible que parce que le graphe ne contient pas de circuit, en raison de l'absence de règles récurifs.

6. Ce processus correspond à un **tri topologique**. Il s'agit de choisir un ordre strict (parmi plusieurs possibles) dans un ensemble d'éléments muni d'une relation de préordre.

avec les lots successifs {COTISATION-SOCIALE}, {NET-IMPOSABLE}, {RETENUE-FISCALE} et {NET-PAYE}. En ce qui concerne les entrées (quand saisir une donnée) et les sorties (quand afficher un résultat), ce graphe indique quand une donnée doit être saisie *au plus tard* et quand un résultat peut être affiché *au plus tôt*.

On peut alors proposer l'algorithme suivant, dans lequel les grandeurs mentionnées dans une liste { } peuvent être évaluées, saisies ou affichées dans un ordre quelconque.

```
saisir {NIVEAU, ANCIENNETE, PRIMES, INDEX}
calculer BRUT
calculer COTISATION-SOCIALE
calculer NET-IMPOSABLE
calculer RETENUE-FISCALE
calculer NET-PAYE
afficher NET-PAYE
```

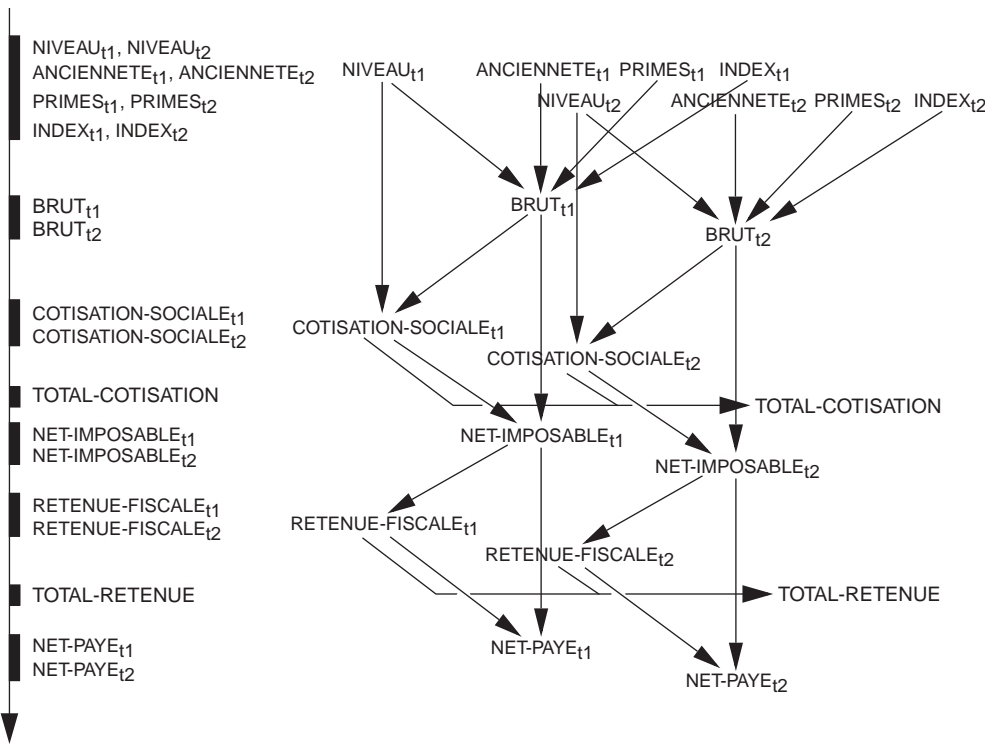


Figure 19.8 - Séquentialisation du modèle 17.9. Afin d'alléger le schéma, on ne représente que deux valeurs de la dimension temporelle

Les modèles dimensionnés se traitent d'une manière analogue, comme le montre la figure 19.8, relative à la séquentialisation du modèle 17.9. Afin de simplifier, et en accord avec la traduction en langage algorithmique, la dimension n'a pas été incluse

dans le raisonnement comme une grandeur ordinaire. On a en outre limité les valeurs de la grandeur temporelle à t_1 et t_2 , ce qui est suffisant pour illustrer la structure générale.

Cette séquentialisation conduit à l'algorithme suivant⁷, dans lequel on a ignoré les opérations d'affichage :

```
saisir{ANCIENNETEt1, ANCIENNETEt2, NIVEAUt1, NIVEAUt2, PRIMESt1,
      PRIMEt2, INDEXt1, INDEXt2}
calculer {BRUTt1, BRUTt2}
calculer {COTISATION-SOCIALEt1, COTISATION-SOCIALEt2}
calculer TOTAL-COTISATION
calculer {NET-IMPOSABLEt1, NET-IMPOSABLEt2}
calculer {RETENUE-FISCALEt1, RETENUE-FISCALEt2}
calculer TOTAL-RETENUE
calculer {NET-PAYEt1, NET-PAYEt2}
```

On remarquera que ce modèle admet plus d'une séquentialisation. On pourrait par exemple proposer une organisation des calculs selon laquelle toutes les grandeurs seraient saisies et calculées pour chacune des valeurs successives de la dimension. L'algorithme se présenterait comme suit :

```
saisir {ANCIENNETEt1, NIVEAUt1, PRIMESt1, INDEXt1}
calculer BRUTt1
calculer COTISATION-SOCIALEt1
calculer NET-IMPOSABLEt1
calculer RETENUE-FISCALEt1
calculer NET-PAYEt1

saisir {ANCIENNETEt2, NIVEAUt2, PRIMESt2, INDEXt2}
calculer BRUTt2
calculer COTISATION-SOCIALEt2
calculer NET-IMPOSABLEt2
calculer RETENUE-FISCALEt2
calculer NET-PAYEt2

calculer TOTAL-COTISATION
calculer TOTAL-RETENUE
```

19.4 RÉALISATION D'UN PROGRAMME SÉQUENTIEL

La traduction de ces algorithmes en un programme séquentiel est immédiate. Chaque grandeur peut être représentée par une variable du programme. Une grandeur indexée sera représentée par un tableau. À partir de la liste des grandeurs consi-

7. Il s'agit en fait d'une famille d'algorithmes dans la mesure où l'ordre de certains calculs est indifférent.

dérées dans l'ordre où elles se présentent dans leur projection, on peut écrire les instructions d'assignation. Le programme ci-dessous a été réalisé en Pascal. Il correspond à une variante de la séquentialisation de la figure 19.8 (le calcul de TOTAL-RETENUE et TOTAL-COTISATION a été reporté en fin de programme). Le second type de séquentialisation pourrait conduire à un autre programme qui lit des données dans une base de données.

```

program TR_EMPLOYE;
uses EMP_IN, EMP_OUT, EMP_SUB;
const   max_nT = 1000;      {nombre maximum de périodes}
var     T : 1..max_nT;      {période courante}
        nT : integer;      {nombre réel de périodes}
        ANCIENNETE :      array[1..max_nT] of integer;
        NIVEAU :          array[1..max_nT] of integer;
        PRIMES :          array[1..max_nT] of integer;
        INDEX :           array[1..max_nT] of integer;

        NET_IMPOSABLE :   array[1..max_nT] of real;
        RET_FISCALE :     array[1..max_nT] of real;
        COTIS_SOCIALE :   array[1..max_nT] of real;
        BRUT :            array[1..max_nT] of real;

        NET_PAYE :        array[1..max_nT] of real;
        TOT_RETENUE_E :   real;
        TOT_COTIS_E :     real;

begin
    lire_Nombre_Periodes(nT,max_nT);
    for T := 1 to nT do begin
        lire_ANCIEN(T,ANCIENNETE[T]);
        lire_NIVEAU(T,NIVEAU[T]);
        while not ((NIVEAU[T] >= 2) and (NIVEAU[T] <= 12)) do
            lire_NIVEAU(T,NIVEAU[T]);
        lire_PRIME(T,PRIME[T]);
        lire_INDEX(T,INDEX[T]);
    end;
    for T := 1 to nT do begin
        BRUT[T] := (BR(ANCIENNETE[T],NIVEAU[T]) + PRIME[T])
            * INDEX[T];
        COTIS_SOCIALE[T] := CS(BRUT[T],NIVEAU[T]);
        NET_IMPOSABLE[T] := BRUT[T] - COTIS_SOCIALE[T];
        RET_FISCALE[T] := RF(NET_IMPOSABLE[T]);
        NET_PAYE[T] := NET_IMPOSABLE[T] - RET_FISCALE[T];
    end;
    TOT_RETENUE_E := 0;
    for T := 1 to nT do
        TOT_RETENUE_E := TOT_RETENUE_E + RET_FISCALE[T];
    TOT_COTIS_E := 0;
    for T := 1 to nT do
        TOT_COTIS_E := TOT_COTIS_E + COTIS_SOCIALE[T];
    afficher_NET_PAYE(T,NET_PAYE);
    afficher_TOT_RETENUE(TOT_RETENUE);
    afficher_TOT_COTIS(TOT_COTISATION)
end.

```

19.5 EXERCICES

- 19.1 Planter une solution de l'exercice 18.4 dans une feuille de calcul.
- 19.2 Planter une solution de l'exercice 18.10 dans une feuille de calcul.
- 19.3 Planter une solution de l'exercice 18.14 dans une feuille de calcul.
- 19.4 On considère l'exercice 18.23. Planter dans une feuille de calcul les différentes solutions corrigées. Expérimenter en assignant aux données des valeurs provoquant des erreurs.

- 19.5 Évaluer le modèle de l'étude de cas n° 2 (ou celui qui répond à l'exercice 18.17) à l'aide du jeu de test suivant (valeurs de 1994) :

Escales : Charles de Gaulle (CDG), Bahrein (BAH), Singapour (SIN), Djakarta (HLP), Nouméa (NOU), Papeete (PPT), Los Angeles (LAX).

Départ : Quantité résiduelle de l'ordre de 10 000 kilos.

Appareil : Charge utile maximum de 45 tonnes. Charge réelle de 30 tonnes. Réservoirs d'une capacité totale de 120 000 litres. Consommation à vide : 7 kg/km. Consommation de transport de charge : 0,17 kg par tonne utile par km.

Devises : 1 PFR (franc pacifique) = 0,055 FFR, 1 USD (US \$) = 5 FFR.

Tarifs : CDG : 2 FFR/kg; BAH : 1 USD/gallon; SIN : 1,1 USD/gallon; HLP : 1,25 USD/gallon; NOU : 2,3 FFR/kg; PPT : 35 PFR/kg; LAX : 1,3 USD/gallon. Un gallon vaut 3,7 litres, et un litre de carburant pèse 0,84 kg à 20°C (température supposée constante à toutes les escales).

Distances : CDG-BAH : 4900 km; BAH-SIN : 5500 km; SIN-HLP : 2700 km; HLP-NOU : 4200 km; NOU-PPT : 2100 km; PPT-LAX : 5300 km.

On expérimentera ce modèle pour diverses valeurs des quantités de carburant achetées aux escales.

- 19.6 On admet que la taille de la population d'une espèce animale particulière évolue dans le temps selon la loi suivante :

$$N_t = r * (1 - N_{t-1}) * N_{t-1} \quad t = 1 \dots M$$

$$0 \leq N_t \leq 1$$

Expliquer en français le comportement de la population d'après ce qu'on peut retirer de cette loi. Construire un modèle et le traduire dans un tableur. Représentez graphiquement l'évolution de la population.

- 19.7 Le procédé de séquentialisation proposé exclut explicitement les modèles qui comprennent des règles récursives. Proposer une extension de ce procédé qui prenne ces règles en charge. Proposer également une technique de traitement de ces modèles dans un langage procédural (distinguer les langages admettant la récursivité de ceux qui ne l'offrent pas).

Chapitre 20

Modèles : études de cas

Ce chapitre reprend les deux domaines d'application décrits au chapitre 12 ainsi que leur analyse sous l'angle des relations de calcul qui les décrivent. Pour chacun de ces domaines, on élabore de manière systématique un modèle de calcul qui le représente au mieux.

20.1 INTRODUCTION

Nous reprendrons les deux domaines d'application étudiés au chapitre 12 dans le cadre des bases de données en nous intéressant cette fois aux informations quantitatives (numériques et logiques) dérivées qu'on peut en tirer. Comme décrit dans la troisième partie de [Hainaut, 1994] et dans le site Web de l'ouvrage, ces informations dérivent de données qui ne sont rien d'autre que des attributs mis en évidence dans la phase d'analyse conceptuelle (schémas des figures 12.1 et 12.3). Afin de mieux illustrer les concepts des modèles de calcul, les énoncés seront légèrement reformulés tout en restant conformes à leur version base de données.

20.2 LES ANIMAUX DU ZOO

Par rapport à l'énoncé présenté au chapitre 12, on précise les informations qu'on demande au modèle d'établir.

20.2.1 Énoncé

On considère un zoo pendant une durée déterminée (un certain nombre de jours). Ce zoo possède des animaux qui suivent chacun un régime alimentaire. Un régime est constitué d'un mélange d'ingrédients, chacun en quantité déterminée. Le régime d'un animal peut varier d'un jour à l'autre. Chaque animal est caractérisé, en fonction de son espèce, par ses besoins minima en nutriments (calcium, protéines, etc.), exprimés en mg par unité de poids de l'animal. On connaît aussi la teneur de chaque ingrédient en nutriments, exprimée en mg par kg d'ingrédient. Chaque ingrédient a un coût unitaire. Chaque animal requiert des soins journaliers, dont on connaît le coût pour chaque jour.

On demande le coût de chaque animal pour chaque jour. On demande aussi si chacun de ses régimes quotidiens est correct du point de vue de ses besoins en nutriments. On demande enfin le coût total des animaux pour la période.

20.2.2 Construction du modèle abstrait

L'énoncé cite explicitement des grandeurs qui sont susceptibles d'apparaître dans le modèle. Avant d'établir la liste de ces grandeurs et de les classer selon leur type, on se propose de compléter le texte de l'énoncé en y indiquant le nom des grandeurs que ce texte suggère, ainsi que leurs dimensions. Une brève analyse pourrait donner ce qui suit.

On considère un zoo pendant une durée déterminée (un certain nombre de jours (\mathcal{J})). Ce zoo possède des animaux (\mathcal{A}) qui suivent chacun un régime alimentaire (dépendant de \mathcal{A}). Un régime est constitué d'un mélange d'ingrédients (\mathcal{I}), chacun en quantité déterminée (Q_{Ingr}). Le régime d'un animal peut varier d'un jour à l'autre (dépend de \mathcal{J}). Chaque animal est caractérisé, en fonction de son espèce, par ses besoins minima ($\text{Besoins}_{\mathcal{J}}$) en nutriments (calcium, protéines, etc.) (\mathcal{N}), exprimés en mg par unité de poids de l'animal (un animal a donc un poids Poids). On connaît aussi la teneur de chaque ingrédient en nutriments (Q_{Nutr}), exprimée en mg par kg d'ingrédient. Chaque ingrédient a un coût unitaire ($\text{Coût}_{\text{Ingr}}$). Chaque animal requiert des soins journaliers, dont on connaît le coût pour chaque jour ($\text{Coût}_{\mathcal{S}}$).

On demande le coût de chaque animal pour chaque jour (Coût). On demande aussi si chacun de ses régimes quotidiens est correct du point de vue de ses besoins en nutriments (BesSatisf). On demande enfin le coût total des animaux pour la période ($\text{Coût}_{\text{TotPer}}$).

Cette première analyse montre que le modèle comporte quatre dimensions : les animaux (\mathcal{A}), les jours (\mathcal{J}), les ingrédients (\mathcal{I}) et les nutriments (\mathcal{N}). Il est possible d'aborder la construction du modèle de plusieurs manières :

- l'approche directe : on part de l'énoncé tel qu'il est proposé, on y repère les résultats et les données probables, on construit les règles comme proposé à la section 18.3;

- l'approche par généralisation : on repère une situation simplifiée, telle que celle d'**un animal pendant un jour**, puis on généralise par dimensionnement selon A et J comme décrit en 18.3.4;
- l'approche par les sous-modèles : on repère et modélise la même situation élémentaire, puis on l'*encapsule* sous la forme d'un sous-modèle. On construit alors le modèle général en tenant compte de l'existence de ce sous-modèle.

Étant donné le nombre de dimensions, nous éviterons l'approche directe pour adopter les deux dernières approches car elles sont aisément généralisables à des problèmes de grande taille¹.

Tentons de tirer de l'énoncé les faits qui concernent le cas d'*un animal pendant un jour*. On ignore donc les dimensions A et J. Il vient, en indiquant les dimensions des grandeurs :

Un régime est constitué d'un mélange d'ingrédients (I), chacun en quantité déterminée (QIngr_I). L'animal est caractérisé par ses besoins minima (Besoins_{JN}) en nutriments (calcium, protéines, etc.) (N), exprimés en mg par unité de poids (Poids). On connaît aussi la teneur de chaque ingrédient en nutriments (QNutr_{I,N}), exprimée en mg par kg d'ingrédient. Chaque ingrédient a un coût unitaire (CoutIngr_I). L'animal requiert des soins, dont on connaît le coût (CoutS).

On demande le coût de l'animal (Cout). On demande aussi si son régime est correct du point de vue de ses besoins en nutriments (BesSatisf).

Le repérage des résultats est immédiat (Cout, BesSatisf). Les données sont les dimensions I et N, et les grandeurs Poids, CoutS, QIngr_I, QNutr_{I,N}, Besoins_{JN}, CoutIngr_I. La construction du modèle complet est très simple. Une version en est proposée à la figure 20.1.

Données

I : i ₁ ..i _p	; identification des ingrédients
N : n ₁ ..n _m	; identification des nutriments
Poids : réel (kg)	; poids de l'animal
CoutS : réel (1)	; coût des soins de l'animal
QIngr _I : réel (kg)	; quantité de l'ingrédient I dans le régime de l'animal
QNutr _{I,N} : réel (mg/kg)	; quantité de nutriment N dans l'ingrédient I
Besoins _{JN} : réel (mg/kg)	; besoins journaliers de l'animal en nutriment N
CoutIngr _I : réel (1/kg)	; coût de l'ingrédient I

Résultats

Cout : réel (1)	; coût de l'animal
BesSatisf : logique	; le régime de l'animal est-il satisfaisant?

1. Malgré sa simplicité, ce cas donne du fil à retordre à plus d'un étudiant. Le principal écueil est ici la maîtrise des dimensions.

Grandeurs internes

CoutN : réel (F) ; coût en nourriture de l'animal
 Apport_N : réel (mg) ; apport en nutriment N du régime de l'animal

Règles

Cout = CoutN + CoutS
 BesSatisf = et_N(Apport_N ≥ BesoinsJ_N * Poids)

$$\text{CoutN} = \sum_I Q_{\text{Ingr}_I} * \text{CoutIngr}_I$$

$$\text{Apport}_N = \sum_I Q_{\text{Ingr}_I} * Q_{\text{Nutr}_{I,N}}$$

Figure 20.1 - Modèle du calcul du coût d'un animal pendant une journée

De par sa simplicité, ce modèle ne réclame aucune normalisation, et sa validation ne pose pas de difficulté : on prend cependant la peine de vérifier la cohérence des unités et des dimensions.

Le problème réduit étant résolu, nous pouvons à présent généraliser le modèle obtenu en introduisant successivement la dimension A, puis la dimension J. Chaque grandeur est réexaminée (et son unité éventuellement corrigée) afin de déterminer si elle dépend ou non de chacune de ces grandeurs. On observe ainsi que Poids, CoutS, QIngr_I, BesoinsJ_N, Cout, BesSatisf, CoutN, Apport_N, dépendent de A, et que CoutS, QIngr_I, Cout, BesSatisf, CoutN, Apport_N dépendent de J.

On observe enfin que le modèle complet définit un nouveau résultat, Cout-TotPer. Le modèle final, qui correspond à l'énoncé complet, est donné dans la figure 20.2. Sa validation ne pose pas non plus de problèmes particuliers.

Données

J : j₁..j_K ; identification des jours de la période considérée
 A : a₁..a_L ; identification des animaux
 I : i₁..i_P ; identification des ingrédients
 N : n₁..n_M ; identification des nutriments
 Poids_A : réel (kg) ; poids de l'animal A
 CoutS_{A,J} : réel (1/j) ; coût des soins de l'animal A le jour J
 QIngr_{A,I,J} : réel (kg/j) ; quantité de l'ingrédient I dans le régime de l'animal A le jour J
 QNutr_{I,N} : réel (mg/kg) ; quantité de nutriment N dans l'ingrédient I
 BesoinsJ_{A,N} : réel (mg/kg/j) ; besoins journaliers de l'animal A en nutriment N
 CoutIngr_I : réel (1/kg) ; coût de l'ingrédient I

Résultats

Cout_{A,J} : réel (1/j) ; coût de l'animal A le jour J
 BesSatisf_{A,J} : logique ; le régime de l'animal A le jour J est-il satisfaisant ?
 CoutTotPer : entier (1) ; coût total des animaux durant la période considérée

Grandeurs internes

CoutN_{A,J} : réel (1/j) ; coût en nourriture de l'animal A le jour J

$\text{Apport}_{A,N,J}$: réel (mg/j) ; apport en nutriment N du régime de l'animal A le jour J
 CoutTotA_A : réel (1) ; coût de l'animal A pour la période

Règles

$\text{Cout}_{A,J} = \text{CoutN}_{A,J} + \text{CoutS}_{A,J}$
 $\text{BesSatisf}_{A,J} = \text{et}_N(\text{Apport}_{A,N,J} \geq \text{BesoinsJ}_{A,N} * \text{Poids}_A)$
 $\text{CoutTotPer} = \sum_A \text{CoutTotA}_A$
 $\text{CoutTotA}_A = \sum_J \text{Cout}_{A,J}$
 $\text{CoutN}_{A,J} = \sum_I \text{QIngr}_{A,I,J} * \text{CoutIngr}_I$
 $\text{Apport}_{A,N,J} = \sum_I \text{QIngr}_{A,I,J} * \text{QNutr}_{I,N}$

Figure 20.2 - Modèle du calcul des coûts relatifs aux animaux d'un zoo

Partant de la solution réduite de la figure 20.1, nous l'exprimons à présent comme un sous-modèle auquel nous donnons le nom de COUT-A-J. Ce sous-modèle calculant le coût et la satisfaction des besoins d'un animal durant une journée, les règles du modèle principal deviennent particulièrement simples. Elles consistent essentiellement à invoquer le sous-modèle pour chaque valeur de A et de J et à définir le résultat CoutTotPer . Les données et les résultats sont ceux du modèle 20.2.

Une expression possible du modèle final est présentée à la figure 20.3.

Données

J : $j_1..j_K$; identification des jours de la période considérée
A : $a_1..a_L$; identification des animaux
I : $i_1..i_P$; identification des ingrédients
N : $n_1..n_M$; identification des nutriments
 Poids_A : réel (kg) ; poids de l'animal A
 $\text{CoutS}_{A,J}$: réel (1/j) ; coût des soins de l'animal A le jour J
 $\text{QIngr}_{A,I,J}$: réel (kg/j) ; quantité de l'ingrédient I dans le régime de l'animal A le jour J
 $\text{QNutr}_{I,N}$: réel (mg/kg) ; quantité de nutriment N dans l'ingrédient I
 $\text{BesoinsJ}_{A,N}$: réel (mg/kg/j) ; besoins journaliers de l'animal A en nutriment N
 CoutIngr_I : réel (1/kg) ; coût de l'ingrédient I

Résultats

$\text{Cout}_{A,J}$: réel (1/j) ; coût de l'animal A le jour J
 $\text{BesSatisf}_{A,J}$: logique ; le régime de l'animal A le jour J est-il satisfaisant ?
 CoutTotPer : entier (1) ; coût total des animaux durant la période considérée

Grandeurs internes

CoutTotA_A : réel (1) ; coût de l'animal A pour la période

Règles

$\{\text{Cout}_{A,J}, \text{BesSatisf}_{A,J}\} = \text{COUT-A-J}(\text{I}, \text{N}, \text{Poids}_A, \text{CoutS}_{A,J}, \text{QIngr}_{A,I,J}, \text{QNutr}_{I,N}, \text{BesoinsJ}_{A,N}, \text{CoutIngr}_I)$

$$\text{CoutTotPer} = \sum_A \text{CoutTotA}_A$$

$$\text{CoutTotA}_A = \sum_J \text{Cout}_{A,J}$$

modèle COUT-A-J

; calcul du coût et de la satisfaction des besoins d'un animal pour un jour déterminé

Données

$I : i_1..ip$; identification des ingrédients
$N : n_1..n_M$; identification des nutriments
$\text{Poids} : \text{réel (kg)}$; poids de l'animal
$\text{CoutS} : \text{réel (1)}$; coût des soins de l'animal
$\text{QIngr}_I : \text{réel (kg)}$; quantité de l'ingrédient I dans le régime de l'animal
$\text{QNutr}_{I,N} : \text{réel (mg/kg)}$; quantité de nutriment N dans l'ingrédient I
$\text{Besoins}_{JN} : \text{réel (mg/kg)}$; besoins journaliers de l'animal en nutriment N
$\text{CoutIngr}_I : \text{réel (1/kg)}$; coût de l'ingrédient I

Résultats

$\text{Cout} : \text{réel (1)}$; coût de l'animal
$\text{BesSatisf} : \text{logique}$; le régime de l'animal est-il satisfaisant?

Grandeurs internes

$\text{CoutN} : \text{réel (1)}$; coût en nourriture de l'animal
$\text{Apport}_N : \text{réel (mg)}$; apport en nutriment N du régime de l'animal

Règles

$\text{Cout} = \text{CoutN} + \text{CoutS}$
 $\text{BesSatisf} = \text{et}_N(\text{Apport}_N \geq \text{Besoins}_{JN} * \text{Poids})$
 $\text{CoutN} = \sum_I \text{QIngr}_I * \text{CoutIngr}_I$
 $\text{Apport}_N = \sum_I \text{QIngr}_I * \text{QNutr}_{I,N}$

Figure 20.3 - Modèle du calcul des coûts relatifs aux animaux d'un zoo - Version modulaire

20.2.3 Implantation du modèle dans une feuille de calcul

Eu égard à la complexité (cependant très relative) du modèle par rapport à ce que nous avons rencontré jusqu'ici, il nous semble utile d'envisager quelque discussion sur les choix d'implantation et leur impact sur l'utilisation pratique.

Nous considérerons d'abord l'hypothèse d'une traduction rapide, qui pourrait convenir à une situation réduite en nombres d'animaux, de jours, de nutriments et d'ingrédients. La figure 20.4 montre la maquette d'un modèle qui occupe une seule feuille de calcul et qu'on a voulue aussi compacte que possible. Les données (notées *dd*) ont été groupées en haut de la feuille et les résultats (notées *rr*) ont été rassemblés vers le bas. La ségrégation n'a pas été effectuée lorsque le mélange de données et de résultats dans un même tableau était pertinent. Signalons encore, pour la

compréhension de la figure, que les grandeurs internes n’ont pas été considérées et que les domaines de valeurs des dimensions ont été choisis comme suit :

A = {a1,a2,a3}
J = {j1,j2,j3,j4,j5}
I = {i1,i2,i3,i4}
N = {n1,n2,n3,n4}.

		QIngr										Cout Ingr					
		j1	j2	j3	j4	j5											
a1	i1	dd	dd	dd	dd	dd	i1	dd	dd	dd	dd	i1	dd				
	i2	dd	dd	dd	dd	dd		dd	dd	dd	i2	dd					
	i3	dd	dd	dd	dd	dd		dd	dd	dd	i3	dd					
	i4	dd	dd	dd	dd	dd		dd	dd	dd	i4	dd					
a2	i1	dd	dd	dd	dd	dd	i2	dd	dd	dd	dd	Poids					
	i2	dd	dd	dd	dd	dd		dd	dd	dd	dd						
	i3	dd	dd	dd	dd	dd		dd	dd	dd	dd						
	i4	dd	dd	dd	dd	dd		dd	dd	dd	dd						
a3	i1	dd	dd	dd	dd	dd	i3	dd	dd	dd	dd						
	i2	dd	dd	dd	dd	dd		dd	dd	dd	dd						
	i3	dd	dd	dd	dd	dd		dd	dd	dd	dd						
	i4	dd	dd	dd	dd	dd		dd	dd	dd	dd						
		QNutr															
		n1	n2	n3	n4												
	i1	dd	dd	dd	dd			dd	dd	dd	dd						
	i2	dd	dd	dd	dd			dd	dd	dd	dd						
	i3	dd	dd	dd	dd			dd	dd	dd	dd						
	i4	dd	dd	dd	dd			dd	dd	dd	dd						
		BesoinsJ															
		n1	n2	n3	n4												
	a1	dd	dd	dd	dd			dd	dd	dd	dd						
	a2	dd	dd	dd	dd			dd	dd	dd	dd						
	a3	dd	dd	dd	dd			dd	dd	dd	dd						
		j1	j2	j3	j4	j5											
a1	CoutS	dd	dd	dd	dd	dd											
	Cout	rr	rr	rr	rr	rr											
	BesSatisf	rr	rr	rr	rr	rr											
a2	CoutS	dd	dd	dd	dd	dd											
	Cout	rr	rr	rr	rr	rr											
	BesSatisf	rr	rr	rr	rr	rr											
a2	CoutS	dd	dd	dd	dd	dd											
	Cout	rr	rr	rr	rr	rr											
	BesSatisf	rr	rr	rr	rr	rr											
		CoutTotPer		rr													

Figure 20.4 - Suggestion d’implantation du modèle 20.2 dans une feuille de calcul

Imaginons ensuite un problème de plus grande ampleur tel que, par exemple, le nombre d’animaux soit plus important, la période considérée plus longue et l’alimentation des animaux plus variée. On doit dans ce cas se poser la question de l’ergonomie de l’outil qu’on va construire, en prenant en compte la façon dont l’utilisateur désire utiliser le modèle. On fait l’hypothèse que l’objectif est la gestion quotidienne des animaux du point de vue alimentaire et médical et qu’on peut ignorer les éventuelles demandes de type analytique relevant de l’aide à la décision². Deux questions se posent alors.

1. La question de l'*unité de travail*

Quelle est l'unité de travail, c'est-à-dire la dimension principale autour de laquelle le travail de l'utilisateur va s'organiser prioritairement ? Cette question ne concerne de toute évidence que les animaux et les jours. Concrètement, l'utilisateur désire-t-il travailler par animal ou par jour de la période ?

Un travail centré sur *l'animal* suggère un modèle distinct (dans une feuille de calcul spécifique par exemple) pour chaque animal, dans lequel on reprend toutes les données et tous les résultats qui concernent un animal durant la période considérée. Du point de vue de la dimension temporelle \mathbb{T} , cette présentation peut être qualifiée de *diachronique* : la perception première est celle d'un animal au cours du temps.

Un travail centré sur les tâches *quotidiennes*, en revanche, suggère un modèle (une feuille de calcul) pour chaque jour de la période, dans lequel on reprend les informations de tous les animaux pour cette journée. Du point de vue de la dimension temporelle \mathbb{T} , cette présentation est *synchronique* : la perception première est celle de tous les phénomènes à un instant déterminé.

Ces deux perspectives sont d'ailleurs aussi utiles l'une que l'autre. On peut raisonnablement penser que le vétérinaire préférera la première présentation (une fiche par animal), tandis que le préparateur travaillera sur la seconde (une fiche par jour). Les tableurs permettent de définir ces vues multiples, soit sous la forme de feuilles distinctes, soient *via* les transformations par pivotement.

2. La question de la *classification des données*

On observe rapidement que les données ne sont pas toutes de même nature. On peut par exemple considérer que les données

- `BesoinsJ`, `QIngr` et `Poids` concernent le *vétérinaire*,
- `CoutIngr` et `QNutr` concernent le *responsable de l'approvisionnement*,
- `CoutS` concernent le *chef du personnel*.

Ne serait-il pas utile de partitionner ces données en tableaux séparés et de les présenter dans des feuilles de données personnalisées pour chaque responsable ?

A titre d'illustration, on propose dans la figure 20.5 une présentation possible de la fiche décrivant un animal. On a cherché à regrouper toutes les informations, données et résultats relatifs à chaque animal dans une fiche individuelle. Les données et les résultats plus globaux ne sont pas repris ici ; ils seront représentés dans des feuilles séparées.

2. Telles que celles qui correspondraient à des choix quant aux animaux à conserver, à revendre, à acheter, aux négociations avec les fournisseurs d'aliments, à la structure des coûts, etc.

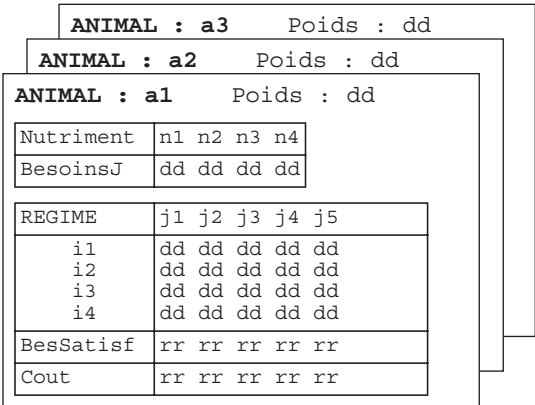


Figure 20.5 - Maquette d'un fragment du modèle 20.2 pour une utilisation centrée sur les animaux

20.3 VOYAGES AÉRIENS

L'énoncé initial présenté dans la section 12.3 exprime explicitement l'objectif de calcul du coût des vols; il semble convenir tel quel. Il présente cependant deux difficultés qui résultent de la prise en compte d'un ensemble de vols :

- ces vols introduisent une dimension supplémentaire par rapport au problème d'un vol particulier,
- chaque vol dépend du précédent effectué par le même appareil, car la quantité de carburant restant dans les réservoirs en fin de vol est la quantité de départ du vol suivant de l'appareil, à moins qu'un événement perturbateur tel qu'un entretien vienne modifier cette relation.

Ces caractéristiques rendent l'analyse plus complexe qu'il n'est utile sur le plan des principes qu'on veut illustrer ici. Nous limiterons donc le problème à celui du calcul du coût d'un vol unique.

20.3.1 Construction du modèle abstrait

Tout comme dans l'étude précédente, essayons de compléter le texte de l'énoncé, réduit au problème d'un seul vol, en y indiquant le nom des grandeurs que ce texte suggère. On obtient :

Un vol relie deux aéroports (qu'on appellera aussi escales) en passant par un certain nombre d'autres escales (E). L'appareil qui effectue le vol est caractérisé par la capacité de ses réservoirs (CapRes) (en kilos de carburant) ainsi que la consommation à vide, équipage compris (ConsVide) (en kilos de carburant par km). On connaît aussi sa consommation supplémentaire par kilo de charge (ConsCharge) (en kilos de carburant par kilo de charge et par km). La consommation à vide n'inclut pas le transport du carburant lui-

même. On admet que la charge utile (Charge) (fret et passagers) est constante pour toute la durée du vol. On connaît la longueur (Distance) de chaque tronçon du vol, c'est-à-dire la distance entre deux escales consécutives. On supposera que la consommation en vol (Q_{ConsKm}) est une fonction linéaire de la charge emportée (charge utile + carburant).

A chaque escale, l'appareil est ravitaillé en carburant (Q_{Emp}). Celui-ci est acheté au tarif local (Tarif) (en dollars par kilo). Lorsque l'appareil atterrit à la fin de chaque tronçon, ainsi qu'à l'aéroport de départ, ses réservoirs contiennent une quantité résiduelle (Q_{Res}) non consommée. Pour effectuer le tronçon suivant, il est généralement nécessaire d'ajouter au réservoir une quantité qui permet d'atteindre l'escale suivante (Q_{Ach}). On fera l'hypothèse que la valeur financière d'une quantité résiduelle est sa valeur de revente au tarif de l'endroit où cette quantité est observée, donc à l'atterrissage. On notera que l'appareil est présumé avoir consommé la quantité résiduelle à l'aéroport de départ ($Q_{\text{ResDepart}}$) et qu'il faut donc la lui imputer (ValeurDepart), mais qu'il n'a pas consommé celle qui subsiste après l'atterrissage final, et qu'il ne faut donc pas la lui imputer (ValeurFinale) puisqu'elle n'aura pas servi au vol.

Cette première analyse montre que le problème est structuré selon une seule dimension (E). Le résultat attendu est le coût du vol en carburant (CoutVol).

Nous utiliserons strictement l'approche directe telle qu'elle a été décrite dans la section 18.3. On remarque que ce problème présente un bel exemple de règles de récurrence. La figure 20.6 propose un modèle de calcul déduit de cet énoncé. Le lecteur vérifiera la cohérence des unités, des dimensions et des règles de récurrence.

Données

$E : e_1, e_2, \dots, e_N$; escales du vol, y compris départ et arrivée
$E' : e_1, e_2, \dots, e_{N-1}$; escales du vol, y compris départ mais arrivée non comprise
$Q_{\text{Emp}_{E'}} : \text{réel (kg)}$; quantité du carburant acheté à l'escale E'
$\text{Tarif}_E : \text{réel (\$/kg)}$; tarif du carburant à l'escale E
$Q_{\text{ResDepart}} : \text{réel (kg)}$; quantité résiduelle de carburant à l'aéroport de départ avant achat
$\text{Distance}_{E'} : \text{entier (km)}$; longueur du tronçon au départ de l'escale E'
$\text{ConsVide} : \text{réel (kg/km)}$; consommation de l'appareil à vide par km
$\text{ConsCharge} : \text{réel (kg/kg/km)}$; consommation de l'appareil pour 1 kg de charge par km
$\text{Charge} : \text{réel (kg)}$; quantité de charge emportée par l'appareil (fret, passagers)
$\text{CapRes} : \text{réel (kg)}$; capacité du réservoir de l'appareil

Résultats

$\text{CoutVol} : \text{réel (\$)}$; coût du vol en carburant
-------------------------------------	----------------------------

Grandeurs internes

$\text{ValeurDepart} : \text{réel (\$)}$; valeur du carburant résiduel à l'escale de départ avant achat
--	---

ValeurFinale : réel (\$)	; valeur du carburant résiduel à l'escale d'arrivée
CoutEscale _{E'} : réel (\$)	; coût en carburant du tronçon au départ de l'escale E'
QRes _E : réel (kg)	; quantité de carburant résiduel à l'escale E avant achat
QCons _{E'} : réel (kg)	; quantité du carburant consommé sur le tronçon au départ de l'escale E'
QConsKm _{E'} : réel (kg/km)	; quantité du carburant consommé par km sur le tronçon au départ de l'escale E'
ChargeMoy _{E'} : réel (kg)	; charge moyenne de l'appareil sur le tronçon au départ de l'escale E'
QCarbMoy _{E'} : réel (kg)	; quantité moyenne de carburant sur le tronçon au départ de l'escale E'
QDepart _{E'} : réel (kg)	; quantité de carburant au départ de l'escale E'
QMin _{E'} : réel (kg)	; quantité minimum de carburant à emporter au départ de l'escale E'
QMax _{E'} : réel (kg)	; quantité maximum de carburant à emporter au départ de l'escale E'

Règles

```

CoutVol = ValeurDepart + ΣE' CoutEscaleE' - ValeurFinale
ValeurDepart = QResDepart x Tarife1
ValeurFinale = QReseN x TarifeN
CoutEscaleE' = QEmpE' x TarifeE'
QRese1 = QResDepart
QResei = QResei-1 + QEmpei-1 - QConsei-1 (i = 2..N)

QConsE' = QConsKmE' x DistanceE'
QConsKmE' = ConsVide + ConsCharge x ChargeMoyE'
ChargeMoyE' = Charge + QCarbMoyE'
QCarbMoyei = (QDepartei + QResei+1) / 2 (i = 1..N-1)
QDepartE' = QResE' + QEmpE'

QMinE' ≤ QEmpE' ≤ QMaxE'
QMinE' = QConsE' - QResE'
QMaxE' = CapRes - QResE'

```

Figure 20.6 - Modèle du calcul du coût d'un vol

On peut envisager, à titre de normalisation, de simplifier ce modèle en cachant certaines grandeurs internes qui n'interviennent que très localement et ne sont pas cruciales pour la compréhension. On propose par exemple de contracter ce modèle en éliminant les grandeurs QDepart et QCarbMoy, qui n'interviennent que dans le détail du calcul de la charge moyenne. En revanche, on propose de conserver les grandeurs QMin et QMax, car elle seront utiles pour surveiller la validité du choix des quantités de carburant à acheter. Cette simplification réduira utilement l'implantation dans une feuille de calcul. La figure 20.7 montre les parties modifiées du modèle.

Données

. . . .

Résultats

CoutVol : réel (\$) ; coût du vol en carburant

Grandeurs internes

. . . .

ChargeMoy_E : réel (kg) ; charge moyenne de l'appareil sur le tronçon au départ de l'escale E'
QMin_E : réel (kg) ; quantité minimum de carburant à emporter au départ de l'escale E'
QMax_E : réel (kg) ; quantité maximum de carburant à emporter au départ de l'escale E'

Règles

. . . .

$QCons_{E'} = QConsKm_{E'} \times Distance_{E'}$
 $QConsKm_{E'} = ConsVide + ConsCharge \times ChargeMoy_{E'}$
 $ChargeMoy_{e_i} = Charge + (QRes_{e_i} + QEmpe_i + QRes_{e_{i+1}}) / 2$
(i = 1..N-1)
 $QMin_{E'} \leq QAch_{E'} \leq QMax_{E'}$
 $QMin_{E'} = QCons_{E'} - QRes_{E'}$
 $QMax_{E'} = CapRes - QRes_{E'}$

Figure 20.7 - Extrait du modèle du calcul du coût d'un vol après normalisation

Ainsi qu'en témoigne clairement le graphe de dépendance, ce modèle contient des circuits. Le fragment de ce graphe (figure 20.8) montre la présence de circuits entre les grandeurs de l'escale e_i et $QRes_{i+1}$. De tels circuits sont l'indice de règles récursives constituant un système d'équations à résoudre.

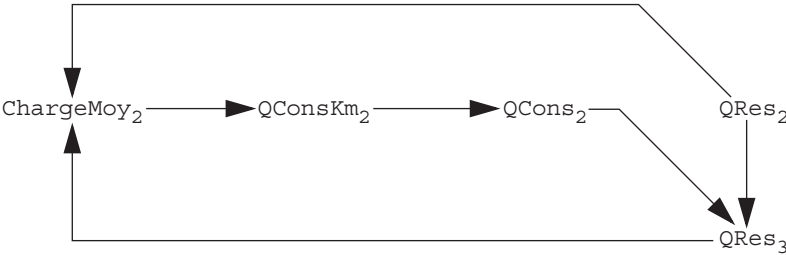


Figure 20.8 - Fragment du graphe de dépendance montrant l'existence de circuits

20.3.2 Implantation du modèle dans une feuille de calcul

Cet énoncé de pose pas de difficulté particulière. On laissera au lecteur le soin de proposer une maquette et une implantation dans le tableau de son choix (par exemple figure 20.9). Le tableur détectera des références circulaires qu'on prendra en compte

en imposant le mode de calcul itératif. Pour ce problème, une limite de 10 pour le nombre d'itérations s'avérera suffisante.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
2		ESCALES				QRésDépart	10000							
3		CODE	TARIF											
4		CDG	0.348			Charge	30000			Val. Départ	3,480			
5		BAH	0.270			Cap. Rés.	120000			Val. Finale	945			
6		SIN	0.300			Cons. Vide	7							
7		HLP	0.330			Cons. Charge	0.00015							
8		NOU	0.400											
9		PPT	0.335			Cout Vol	147,950							
10		LAX	0.350											
11														
12														
13														
14		Num	Esc.Dép.	Esc. Arr.	Distance	Ch. Moy.	Q ConsKm	Q Min	Q Emp.	Q Max	Q Cons.	Q Rés. Dép.	Q Rés. Arr.	Coût Esc.
15		1	CDG	BAH	4,900	82,523	19.38	84,954	90,000	110,000	94,954	10,000	5,046	31,320
16		2	BAH	SIN	5,500	90,475	20.57	108,096	112,000	114,954	113,142	5,046	3,904	30,240
17		3	SIN	HLP	2,700	55,263	15.29	37,378	42,000	116,096	41,282	3,904	4,622	12,600
18		4	HLP	NOU	4,200	72,184	17.83	70,254	75,000	115,378	74,876	4,622	4,746	24,750
19		5	NOU	PPT	2,100	49,586	14.44	25,573	30,000	115,254	30,320	4,746	4,427	12,000
20		6	PPT	LAX	5,300	85,064	19.76	100,299	103,000	115,573	104,726	4,427	2,701	34,505

Figure 20.9 - Fragment d'une feuille EXCEL contenant le modèle de calcul du coût d'un vol

Bibliographie

- ANSI - *ANSI Database Language SQL with Integrity Enhancement*, ANSI X3.135.1, 1989.
Disponible, ainsi que les rapports suivants, à l'AFNOR.
- AKOKA J. et COMYN-WATTIAU I., *Conception des bases de données relationnelles*, Vuibert, 2001.
- ASHCROFT E. A. et WADGE W. W. - LUCID, A nonprocedural language with iteration, in *Comm. ACM*, Vol. 20, pp. 519-526, July 1977.
- BATINI C., CERI S. et NAVATHE S., B. - *Conceptual Database Design - An Entity-Relationship Approach*, Benjamin/Cummings, 1992.
- BECKER K. - *Reusable Framework for Decision Support Systems Development*, Doctoral thesis, Institut d'Informatique, Facultés Universitaires de Namur, 1993.
- BLAHA M. et PERMERLANI W. - *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, 1998.
- BLANNING R., W. - *A Relational Theory of Model Management*, in *Decision Support Systems : Theory and Application*, Holsapple, C., Whinston, B., (Eds), pp. 19-53, Springer-Verlag, 1987.
- BLATTNER P., COOK K. et ULRICH L. - *Excel 2000, Le Macmillan*, Campus, 1999.
- BODART F. et PIGNEUR Y. - *Conception assistée des systèmes d'information - Méthode, modèles, outils*, Masson, 1994.
- BOOCH G. - *Object-Oriented Analysis and Design with Applications*, Benjamin Cummings, 1994.
- BOUZEGHOUB M., GARDARIN G. et VALDURIEZ P. - *Les objets*, Eyrolles, 1997.
- BOUZEGHOUB M. et JOUVE M. - *Le modèle relationnel*, Hermès, 1998.
- BROWN P. et GOULD P. - An experimental study of people creating spreadsheets, in *ACM TOIS*, Vol. 5, N° 3, pp. 258-272, 1987.
- CELKO, J. - *SQL avancé: programmation et techniques avancées*, Vuibert, 2000
- CERI S. et FRATERALI P. - *Designing Database Applications with Objects and Rules - The IDEA Methodology*, Addison-Wesley, 1997.
- CHAMBERLIN D. - *Using the New DB2, IBM's Object-Relationsl Database System*, Morgan Kaufmann, 1996.

- CHAMBERLIN D., D. et BOYCE R., F. - SEQUEL : A Structured English Query Language, in *Proc. ACM SIGMOD Workshop on Data Description, Access and Control*, Ann Arbor, Mich., May 1974.
- CHEN P., P. - The Entity-Relationship Model - Towards a Unified View of Data, in *ACM TODS*, Vol. 1, N° 1, pp. 9-36, 1976.
- CODD E., F. - A Relational Model of Data for Large Shared Data Banks, in *Comm. ACM*, Vol. 13, N° 6, June 1970.
- CODD E., F. - Extending the Database Relational Model to Capture more Meaning, in *ACM TODS*, Vol. 4, N° 4, Dec. 1979.
- CODD E., F. - *The Relational Model for Database Management - Version 2*, Addison-Wesley, 1990.
- CONNOLLOY T et BEGG C. - *Database Systems - A Practical Approach to Design, Implementation and Management*, Addison-Wesley, 2002.
- COURBON J-C. - *Systèmes d'information : structuration, modélisation et communication*, InterÉditions, 1993.
- DATE C. J. - *An Introduction to Database Systems*, Addison-Wesley, 1999.
- DATE C., J. - *Relational Database Writings - 1985-1989*, Addison-Wesley, 1990.
- DATE C., J. - *Relational Database Writings - 1989-1991*, Addison-Wesley, 1992.
- DATE C., J. et DARWEN H. - *The SQL Standard*, Addison-Wesley, 1997.
- DATE C., J. - *Introduction aux bases de données*, Vuibert Informatique, 2001.
- DEHENEFFE C., HAINAUT J-L. et TARDIEU H. - The Individual Model, in *Proc. of the Intern. Workshop on Data Structure Models for Information Systems*, Namur, May 1974, Presses Universitaires de Namur, 1975.
- DELMAL P. - *SQL 2 - SQL 3, Application à Oracle*, De Boeck Université, 2001.
- DELOBEL C. et ADIBA M. - *Bases de données et systèmes relationnels*, Dunod, 1982.
- DION, *Analyse numérique*, Module (Collection universitaire de mathématiques), 1999.
- D'SOUZA D. et WILLS A. - *Objects, Components and Frameworks with UML*, Addison-Wesley, 1999.
- ELMASRI R. et NAVATHE S. - *Fundamentals of Database Systems*, 3rd Edition, Addison-Wesley, 2000.
- FOWLER M. - *UML Distilled - Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.
- FRELAUT P-Y. - *Initiation au HTML*, OEM, 1999.
- GARDARIN G. - *Internet et bases de données*, Eyrolles, 1999.
- GARDARIN G. - *XML : des bases de données aux services web*, Dunod, 2003.
- GEIGER - *Inside ODBC*, Microsoft Press, 1995.
- GEOFFRION A., M. - An Introduction to Structured Modeling, in *Management Sciences*, Vol. 33, N° 5, pp. 547-589, May 1987.
- GEOFFRION A., M. - *The SML Language for Structured Modeling : Levels 1 and 2*, Spec. report , University of California, LA, April 1991.
- GEOFFRION A., M. - *The SML Language for Structured Modeling : Levels 3 and 4*, Spec. report , University of California, LA, July 1991.
- HAINAUT J-L. - *Bases de données et bases de connaissances en gestion des organisations*, notes de la cinquième Ecole d'Automne de l'AFCET sur les Bases de données et Bases de connaissances, Port Barcarès, novembre 1989, p. 121, AFCET, 1989.

- HAINAUT J.-L. - *Conception assistée des applications informatiques - Conception de la base de données*, Masson, 1986.
- HAINAUT J.-L. - Systèmes d'aide à la décision : une approche méthodologique intégrée pour l'utilisateur final, in *Actes du Congrès INFORSID 90*, Vol. 2, Biarritz, mai 1990, pp. 7-34, AFCET, 1990.
- HAINAUT J.-L., CHANDELON M., TONNEAU C. et JORIS M. - Contribution to a Theory of Database Reverse Engineering, in *Proc. on the IEEE W.C. on Reverse Engineering*, Baltimore, May 1993, IEEE Computer Society Press, 1993.
- HAINAUT J.-L. - *Bases de données et modèles de calcul - Outils et méthodes pour l'utilisateur*, InterEditions, 1994.
- HAINAUT J.-L. - *Projet de base de données historiques*, Collection Etudes de cas, Laboratoire d'ingénierie des applications de bases de données, Université de Namur, 2001a. (www.info.fundp.ac.be/libd)
- HAINAUT J.-L. - *Projet de génération de pages HTML/XML*, Collection Etudes de cas, Laboratoire d'ingénierie des applications de bases de données, Université de Namur, 2001b. (www.info.fundp.ac.be/libd)
- HAINAUT J.-L. - *Projet de rétro-ingénierie*, Collection Etudes de cas, Laboratoire d'ingénierie des applications de bases de données, Université de Namur, 2001c. (www.info.fundp.ac.be/libd)
- HAINAUT J.-L. - *Introduction to Database Reverse Engineering*, Laboratoire d'ingénierie des applications de bases de données, Université de Namur, 2002. (<http://www.info.fundp.ac.be/~dbm/publication/2002/DBRE-2002.pdf>)
- HAINAUT J.-L. - *Introduction to Database Design*, Laboratoire d'ingénierie des applications de bases de données, Université de Namur, 2002b. (<http://www.info.fundp.ac.be/~dbm/publication/2002/Mini-Tutoriel.pdf>)
- HALPIN T. - *Conceptual Schema and Relational Database Design*, Prentice-Hall, 1995
- DASPET E. et de GEYER C. - *PHP 5 avancé*, Eyrolles, 2004.
- HICK J.-M. et FORTEMPS J. - *Contribution à une méthodologie de développement d'applications d'aide à la décision*, mémoire de maîtrise, Institut d'Informatique, Facultés Universitaires de Namur, 1991.
- HOLSAPPLE C., W. et WHINSTON A., D. - *Business expert systems*, IRWIN, 1987.
- HOLSAPPLE C., W. et WHINSTON, A., D., (Eds) - *Decision Support Systems : Theory and Application*, Springer-Verlag, 1987.
- JACOBSON I., CHRISTERSON M., JONSSON P. et ÖVERGAARD G. - *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
- KONOPASEK M. et JAYARAMAN S. - *The TK!Solver Book - A Guide for Problem Solving*, Osborne-McGraw-Hill, 1984.
- KOSHAFIAN S., CHAN A., WONG A. et WONG, H. - *Client/Server SQL Applications*, Morgan Kaufmann, 1992.
- KRISHNAN R., XIAOPING LI et STEINER D. - A Knowledge-based Mathematical Model Formulation System, in *Comm. ACM*, Vol. 35, N°9, pp. 138-146, Sept. 1992.
- KYD C. et KINATA C. - *Les macros d'EXCEL*, Dunod Tech, 1992.
- LAZIMI R. - *E²R and object-oriented representation for data management, process modeling and decision support*, in *Proc. of the 8th Int.. conf. on the Entity-Relationship Approach*, Toronto, Oct. 1989, North-Holland, pp. 129-151, 1990.

- LAZIMI R. - Knowledge Representation and Modeling Support in Knowledge-Based Systems, in *Proc. of the 6th Int. conf. on the Entity-Relationship Approach*, New-York, Nov. 1987, pp. 133-161, North-Holland, 1988.
- LE CHARLIER B. - L'analyse statique des programmes par interprétation abstraite, in *Nouvelles de la Science et des Technologies*, Vol. 4, n° 4, pp. 19-25, GORDES, av. Jeanne, 44, B-1050 Bruxelles, 1991.
- MELTON J., SIMON A. - *SQL:1999 - Understanding Relational Language Components*, Morgan Kaufmann Publ., 2002.
- MELTON J. - *Advanced SQL:1999 - Understanding Object-Relational and Other Advanced Features*, Morgan Kaufmann Publ., 2003.
- MICHARD A. - *XML : Langage et Applications*, Eyrolles, 2001.
- NANCY D. et ESPINASSE B. - *Ingénierie des systèmes d'information Merise, Deuxième génération*, Sybex, 1996.
- PATTON N.(Ed.) - *Active Rules in Database Systems*, Springer-Verlag, Monographs in Computer Science, 1999.
- PICHAT E. et BODIN R. - *Ingénierie des données - Bases de données, Systèmes d'information, Modèles et langages*, Masson, 1990.
- PRYWES N., S. et LOCK E., D. - Use of the MODEL Equational Language and Program Generator by Management Professionals, in *Software Reusability - Vol 2 : Applications and Experience*, Biggers-taff, T., J., Perlis, A., J. (Eds), pp. 103-129, Addison-Wesley, 1989.
- PRYWES N., S. et PNUELY A. - Compilation of Nonprocedural Specification into Computer Programs, in *IEEE Trans. Soft. Eng.* SE-9, pp. 267-279, May 1983.
- REESE G. - *JDBC et JAVA, guide du programmeur*, O'Reilly France, 1998.
- RONEN B., PALLEY M., A. et LUCAS H., C. Jr - Spreadsheet Analysis and Design, in *Comm. ACM*, Vol. 32, N° 1, pp. 84-93, Jan. 1989.
- RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F. et LORENSEN F. - *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- SILVERSTON L., INMON W. H. et GRAZIANO K. - *The Data Model Resource Book, A Library of Logical Data Models and Data Warehouse Designs*, Wiley, 1997.
- SNODGRASS R., T. - *Developing Time-Oriented Database Applications in SQL*, Morgan-Kaufmann, 2000.
- SPACCAPIETRA S. et PARENT C. - View Integration : A Step Forward in Solving Structural Conflicts, *IEEE Trans. on Knowledge and Data Engineering*, October 1992.
- TEO T. et TAN M. - Spreadsheet development and "what-if" analysis: quantitative versus qualitative errors, in *Journal of Accounting, Management and Information Technologies*, Vol. 9, n° 3, pp 141-160, Elsevier, Oct. 1999.
- UML - *OMG Unified Modeling Language Specification*, Version 1.3 alpha R2, OMG Inc., 1999.

Index

A

- add
 - check 144
 - column 61, 120
 - constraint 63, 145
 - foreign key 63, 120
 - primary key 62, 120
 - unique 62
- adresse 319
 - absolue 323, 324
 - relative 323
- AGL 41, 201, 288
- agrégat physique (cluster) 50
- alias
 - de colonne 77
 - de table 87
- alter
 - column 62
 - table 61
- association (UML) 226
 - binaire 226
 - d'agrégation 228
 - de composition 228
 - qualifiée 228
- attribut 204
 - composé 225
 - d'un type d'entités 270
 - de classe 227
 - de types d'associations 224
 - facultatif 205
 - multivalué 225, 247
 - obligatoire 205

B

- base de données active 170
- base de données temporelles 174, 192
- between 70

C

- cardinalité
 - d'attribut 205
 - d'un role 209
- catalogue 147, 165
- cellules 312
- Chen 29
- chimpanzé 396
- circuit de dépendances 344, 424
- classe d'objets 226
- classe fonctionnelle
 - plusieurs-à-plusieurs 208, 275
 - un-à-plusieurs 206, 270
 - un-à-un 207, 273
- clé étrangère 34
 - delete mode 44
 - multi-composant 37, 41, 83, 89, 102
 - temporelle 178
 - transitive 41
 - update mode 45
- close 154
- coalescing 176
- COBOL 18, 152
- Codd 27
- codomaine 383
- coexistence 36
- cohérence
 - d'un modèle 374

- d'un modèle de calcul 370
- des dimensions 379
- des domaines de valeurs 381
- des unités 377
- règle de définition multiple 375
- règle de récurrence 376
- structurelle 374
- colonne 32
 - de référence 34
 - facultative 36
 - obligatoire 36, 45
- condition
 - d'association 82, 101
 - de convergence 354
 - de jointure 92
 - de non-association 84, 101
- consultation de table 320
- contrainte
 - d'intégrité 43, 46, 225
 - d'unicité 43, 216
 - dynamique 217, 282
 - référentielle 35, 43, 118, 145, 271
 - statique 217, 282
 - UML 229
- contrôle d'accès 52, 131
- convergence 406
- create
 - dbspace 64
 - domain 58
 - index 63
 - procedure 146
 - schema 56
 - table 57
 - trigger 147
 - unique index 63
 - view 134
- critère de groupement 108
- curseur 153
- cyclique
 - clé étrangère 96
 - structure de données 96, 99
 - type d'associations 210, 276

D

- Data Definition Language (DDL) 56
- DB2 27, 54
- dBase 17
- declare cursor 154
- déclencheur 146, 170, 180
- décomposition

- d'un type d'entités 259
- d'une table 46, 164
- default 59, 164
- delete 116, 154
- delete mode
 - cascade 118
 - no action 118
 - set null 119
- dépendance fonctionnelle 266
 - anormale 48, 260
 - dans une table 47
 - déterminant 48
- dictionnaire de données 148, 165
- dimension 348, 371, 379
- distinct 69
- divergence 355
- domaine d'application 199, 203, 336
- domaine de valeurs 57, 382
- données groupées 107
- drop
 - column 62, 120
 - constraint 63, 145
 - domain 62
 - index 64
 - table 61, 120
 - view 135

E

- ensemble vide 82, 88, 163
- entité 31, 203
- équation aux différences finies 353
- espace de stockage 50, 278
- Excel 315, 317
- exec SQL 152
- expression conditionnelle 321

F

- fetch 154
- feuille de calcul 312, 318
- Firebird 54
- fonction agrégative
 - Excel 320
 - modèle de calcul 349
 - SQL 80, 108, 158
- fonctions SQL 77
- foreign key 34, 60, 118
- forme normale d'une table
 - Boyce-Codd 49
 - troisième 49
- formule de calcul 314, 320

from (extensions) 138

G

génération

- de code 183, 195

- de code DDL 186

- de documents XML 189

- de données 183

- de migrateurs 184

- de pages HTML 187

grandeur 336

- à définition multiple 341, 375, 405

- dimensionnée 348, 398

- interne 340, 400

- logique 342

grant 132

grant option 132

graphe de dépendance 342

group by 108

groupe de lignes 107

groupements multi-niveaux 111

H

having 108

homonyme 252

I

identifiant

- composition de 214

- de jointure 107

- de table 33, 43, 276

- de type d'associations 224

- de type d'entités 211

- hybride 212, 229

- implicite 215

- minimal 35, 214

- primaire 34

- secondaire 34

in 70, 158

index 50, 278

information incomplète 156

INFORMIX 27, 54

INGRES 27, 54

insert 154

- select 116

- values 115

instantané 116

instantané (snapshot) 135

InterBase 56

is false 157

is null 70, 158

is true 157

is unknown 157

J

JDBC 155

join 91

jointure 91

- auto-jointure 98

- de deux tables 91

- de plus de deux tables 92

- externe 93, 141

- généralisée 103

- opérateurs 139

- semi-jointure 96

- temporelle 178

L

lignes célibataires 93

like 70

logique

- binaire 72

- équivalence 76

- implication 75

- ou exclusif 75

- ternaire 74, 156

lois de de Morgan 74

Lotus 1-2-3 17, 315

M

macro 327

MACSYMA 312

mainframe 17

maquette 397

masque SQL 70

MathCAD 312

MATHEMATICA 312

MERISE 28, 200, 288

méthode de classe 225, 227

Microsoft Access 40, 51, 54, 56

migration de données 183

modèle

- de calcul 311, 336

- dimensionné 347, 409

- directionnel 339

- Entité-association 29, 203

- non directionnel 372

- récuratif 352

modify (colonne) 62

modularisation 356

multiplicité (UML) 226

MySQL 54, 59

N

normalisation

d'un modèle de calcul 368, 423

d'un schéma conceptuel 258, 266

d'une table 45

not in 70, 158

not null 60, 158

null 36, 82, 88, 156

O

ODBC 155

open 154

Open Source 54

opérateur ensembliste

except 95

intersect 95

union 94

union all 95

ORACLE 27, 54

order by 113, 161

outer join 141

P

PostgreSQL 54

prédicat

assertion 144

de colonne 144

de table 144

primary key 59

privilège 131

procédure SQL 145

processeur d'équations 337

processeur de modèles 312

produit relationnel 93, 95, 139

programmation linéaire 355

programme d'application 152

projection temporelle 176

proposition

binaire 235

contradictoire 253

élémentaire 235

générale 237

irréductible 237

particulière 237

réductible 239

unaire 235

Q

quantificateur

all 89, 159

any 89, 159

existentiel 90

exists 88, 159

not exists 88, 159

universel 90

Quattro Pro 315

QUEL 27

R

redondance

dans un modèle de calcul 369

dans un schéma 250

interne 259

référence circulaire 325, 406

règle

de contrainte 339, 364, 406

de définition 338

de récurrence 351, 405

récursive 352, 406

relation 336

relation IS-A 223

requête récursive 99, 143

résolution d'équations 355

restructuration

d'un modèle de calcul 369

de schéma 243

rétro-ingénierie 284, 290

revoke 133

rôle 205

de cardinalité 1-N 283

facultatif 208

obligatoire 208

S

scénario 327, 400

schéma

conceptuel 203, 233, 294, 300

de tables 36, 269, 296, 303

script SQL 56

select 66

* 67

colonnes 67

distinct 69

données dérivées 76

SFW 137

séquentialisation d'un modèle 407

SGBD 26, 51

- solveur 328, 356
- sous 401
- sous-modèle 356, 401
- sous-requête 82, 100
- SQL 53
 - SQL-1999 53
 - SQL2 53
 - SQL3 53, 151
 - SQL-86 53
 - SQL-89 53
 - SQL-92 53
 - TSQL2 183
- SQL dynamique 155
- SQL/DS 27
- SQL-Server 27, 54
- StarOffice 315
- stored procedure 145
- structure d'ordre 190
- SYBASE 27, 54
- synonyme 252
- System R 27
- système en évolution 353
- système expert 309, 342

T

- table 31
- table de données 400
 - à double entrée 326
 - à simple entrée 326
- table de vérité 73
- table virtuelle 134
- tableur 312, 337

- TK !Solver 312
- trigger 146, 171, 181
- type d'associations 205, 270
- type d'associations N-aire 224
- type d'entités 203, 270

U

- UML 28, 226
- unique 59
- unknown 158
- update 117, 154
- update mode
 - cascade 119

V

- valeur cible 328
- valeur d'exception
 - absent 345
 - erreur 346, 382
- VisiCalc 17, 315
- Visual Basic 328
- vue relationnelle 133

W

- where
 - condition de totalité 90
 - conditions complexes 71
 - conditions simples 67
 - current 155
- with check option 137
- Works 315

