

Bases de données

Concepts, utilisation
et développement

Jean-Luc Hainaut

Professeur à la faculté d'informatique,
directeur de recherche au centre PReCISE,
université de Namur

4^e édition

DUNOD

Toutes les marques citées dans cet ouvrage
sont des marques déposées par leurs propriétaires respectifs

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, 2009, 2018
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-078672-5

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° 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.



Grand éditeur de SGBD SQL observant avec ironie, condescendance mais aussi un appétit non dissimulé les efforts des SGBD dits NoSQL¹.

1. Référence allégorique à la section 1.7.8. *Linogravure*, B. Hainaut, 2004

Table des matières

AVANT-PROPOS	27
CHAPITRE 1 • MOTIVATION ET INTRODUCTION	33
1.1 Des données et des hommes	33
1.2 Une première approche des données	34
1.3 Utiliser une base de données	38
1.4 Premières conclusions	39
1.5 Les systèmes de gestion de bases de données	40
1.6 Les défis des bases de données d'aujourd'hui	41
1.7 Un peu d'histoire	42
1.7.1 Le modèle collectiviste : CODASYL DBTG	43
1.7.2 La pensée unique : IMS	44
1.7.3 La naissance du modèle relationnel	44
1.7.4 Apparition d'un nouvel acteur : l'utilisateur final	45
1.7.5 La guerre des modèles	45
1.7.6 La chute d'Icare	46
1.7.7 Le modèle relationnel : la suite de l'histoire	46
1.7.8 Cronos dévore ses enfants	47

PARTIE 1 CONCEPTS DES BASES DE DONNÉES

CHAPITRE 2 • CONCEPTS DES BASES DE DONNÉES	49
2.1 Introduction	49
2.2 Tables, lignes et colonnes	50
2.3 La valeur null	51

2.4	Les identifiants	52
2.5	Les clés étrangères	53
2.6	Identifiants et clés étrangères facultatifs	54
2.7	Schéma et contenu d'une base de données	55
2.8	Représentation graphique d'un schéma	55
2.9	Un exemple de base de données	57
2.10	Modification et contraintes d'intégrité	58
2.10.1	Les colonnes obligatoires	58
2.10.2	Les contraintes d'unicité (identifiants)	60
2.10.3	Les contraintes référentielles (clés étrangères)	60
2.11	Redondances internes dans une table	62
2.11.1	Le phénomène de redondance interne	62
2.11.2	Normalisation par décomposition	63
2.12	Les structures physiques	65
2.13	Que retenir ?	66
2.14	Pour en savoir plus	67
2.15	Pratiquer les bases de données	67
2.16	Exercices	69
CHAPITRE 3 • MODÈLE RELATIONNEL ET NORMALISATION		71
3.1	Le phénomène de redondance interne	71
3.2	Le modèle relationnel de bases de données	72
3.2.1	Concepts de base	72
3.2.2	Opérateurs d'extraction de données	73
3.3	Identifiants	76
3.4	Dépendances fonctionnelles	77
3.4.1	Le phénomène de dépendance fonctionnelle	77
3.4.2	Un exemple plus complexe	78
3.4.3	Graphe ADF d'une relation (attributs et DF)	79
3.4.4	Propriétés des dépendances fonctionnelles	79
3.5	Contraintes d'inclusion et clés étrangères	82
3.6	Calcul des identifiants d'une relation	85
3.6.1	Relation entre identifiants et dépendances fonctionnelles	85
3.6.2	La relation possède un seul identifiant	86
3.6.3	La relation possède plusieurs identifiants	87
3.6.4	Rôle des attributs racines	88
3.7	Décomposition d'une relation	89

3.8	Normalisation d'une relation	90
3.8.1	Réexamen du phénomène de redondance interne	91
3.8.2	Analyse du phénomène	91
3.8.3	Relation normalisée	92
3.8.4	Normalisation d'une relation	92
3.8.5	Normalisation d'un noyau irréductible	95
3.8.6	SGBD et dépendances fonctionnelles	97
3.9	Les relations non en 1re forme normale	97
3.10	Que retenir ?	100
3.11	Pour en savoir plus	102
3.12	Exercices	102
	CHAPITRE 4 • ÉLÉMENTS DE TECHNOLOGIE DES BASES DE DONNÉES	105
4.1	Introduction	105
4.2	Les composants d'un ordinateur	106
4.3	Mémoire externe : le disque magnétique	107
4.3.1	Géométrie d'un disque	107
4.3.2	Lecture et écriture de données sur un disque	109
4.3.3	Optimisation : tampon d'échange et lecture anticipée	111
4.4	Les technologies alternatives	113
4.4.1	Les disques RAID	113
4.4.2	Les disques électroniques (SSD)	115
4.4.3	Les disques hybrides	117
4.4.4	Les mémoires à bande	117
4.4.5	Les machines de bases de données	118
4.5	Organisation d'un espace de stockage	120
4.5.1	Structure d'un fichier	120
4.5.2	Allocation d'espace à un fichier	121
4.5.3	Structure des enregistrements d'un fichier	121
4.5.4	Rangement des enregistrements dans un fichier	121
4.5.5	Le phénomène de fragmentation d'un fichier	123
4.5.6	Organisation des fichiers sur le support	124
4.6	Traitement séquentiel d'un fichier	124
4.6.1	Lecture séquentielle d'un fichier	125
4.6.2	Tri d'un fichier séquentiel	128
4.7	Les index	132
4.8	Organisation séquentielle indexée	134
4.8.1	Structure d'un fichier séquentiel indexé	134
4.8.2	Lecture et modification d'un fichier séquentiel indexé	135

4.8.3	Caractéristiques et performances d'un fichier séquentiel indexé	138
4.8.4	Étude de cas	145
4.9	Organisation calculée	146
4.9.1	Principes de l'accès calculé	146
4.9.2	Fonction de calcul d'adresse	147
4.9.3	Gestion des débordements	149
4.9.4	Caractéristiques et performances d'un fichier à accès calculé	150
4.9.5	Chargement et réorganisation d'un fichier à organisation calculée	153
4.9.6	Comparaison des deux techniques d'index primaires	154
4.10	Les index secondaires	154
4.10.1	Structure d'un index secondaire par listes de pointeurs	155
4.10.2	Performances d'un index secondaire par listes de pointeurs	156
4.10.3	Structure d'un index bitmap	159
4.10.4	Les index partiels	159
4.10.5	Les index sur expression	159
4.10.6	Les index secondaires augmentés	159
4.10.7	Disponibilité des techniques d'indexation primaires et secondaires	161
4.11	Accès par index ou recherche séquentielle ?	161
4.12	Quand créer un index ?	161
4.13	Les techniques d'agrégation (clustering)	162
4.13.1	Clustering index	163
4.13.2	Clusters	163
4.13.3	Performances des techniques d'agrégation	164
4.14	Stabilité des adresses	165
4.15	Un exemple : SQL Server	167
4.16	Compléments sur les fonctions de hachage	169
4.16.1	Le prédicteur à hachage (filtre de Bloom)	169
4.16.2	Hachage cryptographique (Secure hashing)	170
4.16.3	Notion d'identifiant universel (UUID)	171
4.17	Futurologie : un peu de poésie dans ce monde brutes !	172
4.17.1	Les mémoires ADN	172
4.17.2	Les mémoires atomiques	173
4.17.3	Les bases de données quantiques	174
4.18	Que retenir ?	176
4.19	Pour en savoir plus	177
4.20	Exercices	177

CHAPITRE 5 • LES SYSTÈMES DE GESTION DE BASES DE DONNÉES	181
5.1 Introduction	181
5.2 Les fonctions des systèmes de gestion de bases de données	182
5.3 Architecture des SGBD	184
5.4 Les transactions	186
5.5 Régulation de la concurrence	188
5.5.1 Mise à jour perdue	189
5.5.2 Données instables	189
5.5.3 Régulation de la concurrence par verrouillage	190
5.5.4 Sérialisabilité et verrouillage en 2 phases (2PL)	191
5.5.5 Interblocage	191
5.5.6 Les niveaux d'isolation	192
5.6 Protection contre les accidents	193
5.6.1 Sauvegardes et journaux	193
5.6.2 Reprise à froid	194
5.6.3 Reprise à chaud	194
5.6.4 Les fichiers différentiels	195
5.7 Les bases de données distribuées	196
5.7.1 Un premier exemple	196
5.7.2 Les types de distribution	198
5.7.3 Partitionnement du réseau	199
5.7.4 Le <code>commit</code> en deux phases (2PC)	200
5.7.5 Distribution des traitements	200
5.8 Contrôle des accès	201
5.8.1 Les modèles de contrôle d'accès	201
5.8.2 Protection des données privées - Inférence statistique	202
5.9 Les bases de données in-memory	204
5.10 Le marché des SGBD relationnels	205
5.11 Pour en savoir plus	206

PARTIE 2

UTILISATION DES BASES DE DONNÉES

CHAPITRE 6 • LE LANGAGE SQL DDL	207
6.1 Introduction	207
6.2 Le langage SQL DDL	208
6.3 Création d'un schéma	208
6.4 Les noms SQL	209

6.5	Création d'une table	209
6.6	Les colonnes et leurs types	209
6.7	Caractère obligatoire/facultatif d'une colonne	211
6.8	Les identifiants	212
6.9	Les clés étrangères	212
6.10	Forme synthétique des contraintes	215
6.11	Exemple complet	215
6.12	Suppression d'une table	216
6.13	Ajout, retrait et modification d'une colonne	216
6.14	Ajout et retrait d'une contrainte	216
6.15	Les structures physiques	217
6.16	Que retenir ?	218
6.17	Pour en savoir plus	218
 CHAPITRE 7 • LE LANGAGE SQL DML (1)		219
7.1	Introduction	219
7.2	Requêtes élémentaires	221
7.2.1	Extraction simple	221
7.2.2	Extraction de lignes sélectionnées	222
7.2.3	Lignes dupliquées dans le résultat	223
7.2.4	Des conditions de sélection plus complexes	224
7.2.5	Expressions composées	226
7.3	Données extraites et données dérivées	227
7.3.1	Les fonctions SQL	227
7.3.2	Les registres du système	229
7.4	Ordre des lignes du résultat	230
7.5	Les fonctions agrégatives (ou statistiques)	231
7.6	Les sous-requêtes	233
7.6.1	Condition d'association par sous-requête	234
7.6.2	Sous-requête et clé étrangère composite	235
7.6.3	Attention aux conditions d'association négatives	235
7.6.4	Références multiples à une même table	237
7.7	Les quantificateurs ensemblistes	239
7.8	Exercices	241

CHAPITRE 8 • LE LANGAGE SQL DML (2)	245
8.1 Introduction	245
8.2 Extraction de données de plusieurs tables (jointure)	246
8.2.1 La jointure de plusieurs tables	246
8.2.2 Conditions de jointure et conditions de sélection	247
8.2.3 Lignes célibataires et jointures externes	248
8.3 Les opérateurs ensemblistes	248
8.4 Le produit relationnel	250
8.5 Variantes syntaxiques de la jointure	251
8.6 Les tables statiques	253
8.7 Requêtes sur des structures cycliques	253
8.8 Compléments sur les jointures	258
8.8.1 Sous-requête ou jointure ?	258
8.8.2 Valeurs dérivées dans une jointure	260
8.8.3 Les jointures généralisées	260
8.8.4 Interprétation du résultat d'une jointure	262
8.9 Extraction de données groupées	265
8.9.1 Notion de groupe de lignes	265
8.9.2 Sélection de groupes et sélection de lignes	266
8.9.3 Groupes et jointures	267
8.9.4 Composition de la liste select	268
8.9.5 Composition du critère de groupement	269
8.9.6 Attention aux groupements multiniveaux	270
8.9.7 Peut-on éviter l'utilisation de données groupées ?	271
8.10 Extension de la structure des requêtes SFW	271
8.10.1 Extension de la clause select	271
8.10.2 Extension de la clause from	272
8.11 Limitation de la taille d'un résultat	273
8.12 Interprétation d'une requête	273
8.13 Modification des données	274
8.13.1 Ajout de lignes	274
8.13.2 Suppression de lignes	276
8.13.3 Modification de lignes	276
8.13.4 À propos de la gestion des verrous	277
8.13.5 Mise à jour et contraintes d'unicité	277
8.13.6 Mise à jour et contraintes référentielles	278
8.13.7 Déterminisme des requêtes SQL	278
8.14 Modification des structures de données	280
8.15 Que retenir ?	281

8.16	Pour en savoir plus	282
8.17	Exercices	282
CHAPITRE 9 • LE LANGAGE SQL AVANCÉ		287
9.1	Introduction	287
9.2	Le contrôle d'accès	287
9.3	Les vues SQL	290
9.3.1	Principes et objectifs des vues	290
9.3.2	Définition et utilisation d'une vue	290
9.3.3	Les vues comme interface pour des besoins particuliers	291
9.3.4	Les vues comme mécanisme de contrôle d'accès	291
9.3.5	Les vues comme mécanisme d'évolution de la base de données	292
9.3.6	Les vues comme aide à l'expression de requêtes complexes	292
9.3.7	Mise à jour des données via une vue	293
9.4	Les Common Table Expressions (CTE)	294
9.5	Les requêtes récursives	294
9.6	Les extensions objet de SQL3	297
9.6.1	Types de données complexes (<code>row</code> et <code>array</code>)	298
9.6.2	Type défini par l'utilisateur (TDU)	300
9.6.3	Table typée	300
9.6.4	Hiérarchie de types	301
9.6.5	Hiérarchie de tables typées	302
9.6.6	Références entre tables	304
9.6.7	Remarques finales	305
9.7	Les prédicats (<code>check</code>)	305
9.8	Les procédures SQL (<code>stored procedures</code>)	307
9.9	Les déclencheurs (<code>triggers</code>)	308
9.10	Le catalogue	313
9.11	Les interfaces SQL des programmes d'application	316
9.11.1	L'interface SQL statique classique	318
9.11.2	SQLJ : une interface statique pour Java	321
9.11.3	L'interface SQL dynamique classique	323
9.11.4	JDBC : une interface CLI-SQL dynamique pour Java	326
9.11.5	Comparaison des modèles d'interaction	328
9.11.6	Un problème de sécurité : l'injection de code SQL	329
9.12	SQL et l'information incomplète	331
9.12.1	Introduction	331
9.12.2	La valeur <code>null</code> de SQL	331
9.12.3	La logique ternaire de SQL	332

9.12.4 La propagation de <code>null</code> en SQL	332
9.12.5 La propagation de <code>unknown</code> en SQL	332
9.12.6 Les problèmes de l'information incomplète en SQL	334
9.12.7 Deux recommandations	339
9.13 Que retenir ?	340
9.14 Pour en savoir plus	341
 CHAPITRE 10 • LES BASES DE DONNÉES NON RELATIONNELLES	 343
10.1 Introduction	343
10.2 Les bases de données NoSQL	345
10.2.1 Modèles clé-valeur	346
10.2.2 Modèles orientés documents	347
10.2.3 Modèles orientés colonnes	347
10.3 Les autres modèles non relationnels	348
10.4 Les architectures de distribution	349
10.4.1 La cohérence finale	349
10.4.2 Les protocoles à quorum	350
10.4.3 Le théorème CAP	351
10.5 Un SGBD orienté colonnes : Cassandra	351
10.6 Un SGBD orienté documents : MongoDB	355
10.7 Les SGBD NewSQL : l'exemple de VoltDB	358
10.8 Conclusions sur les SGBD NoSQL	359
10.9 Les chaînes de blocs	361
10.9.1 Principes généraux	362
10.9.2 Le protocole BitCoin	366
10.9.3 Le protocole Ethereum	370
10.10 Pour en savoir plus	373

PARTIE 3

DÉVELOPPEMENT DES BASES DE DONNÉES

CHAPITRE 11 • CONSTRUCTION D'UNE BASE DE DONNÉES	375
11.1 De l'artisanat à la méthode	375
11.2 Les acteurs et les métiers	376
11.3 Deux parcours	377
11.4 Une démarche simplifiée	377
11.5 Les outils	378

CHAPITRE 12 • LE MODÈLE ENTITÉ-ASSOCIATION DE BASE	379
12.1 Introduction	379
12.2 Les types d'entités	380
12.3 Les attributs	381
12.4 Les types d'associations	382
12.4.1 Rôles d'un type d'associations	382
12.4.2 Propriétés d'un type d'associations	383
12.4.3 Classe fonctionnelle d'un type d'associations	383
12.4.4 Type d'associations (ou rôle) obligatoire ou facultatif	385
12.4.5 Cardinalités d'un type d'associations	386
12.4.6 Type d'associations cyclique	386
12.4.7 Représentation graphique des populations	387
12.5 Une question de vocabulaire	387
12.6 Les identifiants	388
12.6.1 Identifiants hybrides	389
12.6.2 Composition des identifiants	390
12.6.3 Identifiants minimaux et identifiants implicites	391
12.6.4 Importance du concept d'identifiant	392
12.7 Les contraintes d'intégrité	393
12.7.1 Contraintes de base	393
12.7.2 Contraintes additionnelles	393
12.8 Contenu informationnel d'un schéma	393
12.9 Exemples	395
12.9.1 Une structure administrative	395
12.9.2 Gestion d'une bibliothèque	396
12.9.3 Voyages en train	397
12.10 Que retenir ?	398
12.11 Pour en savoir plus	399
12.12 Dessiner un schéma conceptuel à l'aide d'un outil	399
12.13 Exercices	400
 CHAPITRE 13 • ÉLABORATION D'UN SCHÉMA CONCEPTUEL	 403
13.1 Introduction	403
13.2 Un premier exemple	404
13.3 Décomposition de l'énoncé	406
13.3.1 Notion de proposition élémentaire	406
13.3.2 Cardinalités	407
13.3.3 Propositions générales et propositions particulières	408

13.3.4	Attention aux propositions complexes irréductibles !	408
13.3.5	Les propositions non binaires réductibles	411
13.4	Pertinence d'une proposition	411
13.5	Représentation d'une proposition	412
13.5.1	Nouveau type d'entités et son attribut	412
13.5.2	Type d'associations entre types d'entités existants	413
13.5.3	Attribut d'un type d'entités existant	413
13.5.4	Nouveau type d'entités et ses deux attributs	414
13.5.5	Restructuration pour intégration	414
13.5.6	Propriétés multivaluées	417
13.5.7	Recommandations pratiques	420
13.6	Non-redondance des propositions	421
13.6.1	Redondance explicite	421
13.6.2	Variantes d'expression	421
13.6.3	Redondance indirecte	421
13.6.4	Redondance apparente	422
13.6.5	Synonymes et homonymes	422
13.7	Non-contradiction des propositions	423
13.8	Les contraintes d'intégrité	424
13.9	Normalisation du schéma	425
13.9.1	Simplification du schéma	425
13.9.2	Élimination des redondances internes	425
13.10	Documentation du schéma	427
13.11	Complétude du schéma	428
13.12	Que retenir ?	428
13.13	Pour en savoir plus	429
13.14	Exercices	429
CHAPITRE 14 • PRODUCTION DU SCHÉMA DE LA BASE DE DONNÉES		433
14.1	Introduction	433
14.2	Représentation des types d'entités	434
14.3	Représentation des attributs	434
14.4	Représentation des types d'associations	434
14.4.1	Types d'associations un-à-plusieurs (1:N)	434
14.4.2	Types d'associations un-à-un (1:1)	436
14.4.3	Types d'associations plusieurs-à-plusieurs (N:N)	438
14.4.4	Types d'associations cycliques	439
14.5	Représentation des identifiants	439

14.6	Traduction des noms	440
14.7	Synthèse des règles de traduction	441
14.8	Les structures physiques	442
14.9	Traduction des structures en SQL	444
14.10	Que retenir ?	445
14.11	Pour en savoir plus	446
14.12	Produire un schéma de base de données à l'aide d'un outil	446
14.13	Exercices	447
 CHAPITRE 15 • MÉTHODOLOGIE DES BASES DE DONNÉES		449
15.1	Introduction	449
15.2	Méthodes, processus et produits	450
15.3	Méthodes de développement d'une base de données	451
15.4	Les modèles de développement	456
15.5	Techniques transformationnelles	458
15.6	Rôle des outils	458
 CHAPITRE 16 • LE MODÈLE ENTITÉ-ASSOCIATION ÉTENDU		459
16.1	Introduction	459
16.2	Le modèle de base - rappel	460
16.3	Sémantiques	461
16.4	Modèle, schéma et population	462
16.5	Les stéréotypes	464
16.6	Schéma de base de données	465
16.7	Attributs	465
16.7.1	Attributs composés	465
16.7.2	Attributs multivalués	465
16.7.3	Attributs complexes	466
16.7.4	Domaine et type d'un attribut	466
16.7.5	Cardinalité d'un attribut	468
16.8	Types d'associations	468
16.8.1	Types d'associations fonctionnels	468
16.8.2	Types d'associations n-aires	468
16.8.3	Types d'associations à rôles polymorphiques	469
16.8.4	Attributs de type d'associations	470
16.8.5	Généralisation des cardinalités des rôles	470
16.8.6	Taille de la population d'un type d'associations	470

16.9	Types d'entités	472
16.9.1	Relation de généralisation-spécialisation (relation is-a)	472
16.9.2	Propriétés et contraintes ensemblistes des sous-types	473
16.9.3	L'héritage	474
16.9.4	Les sous-types non disjoints sont-ils vraiment utiles ?	476
16.9.5	Surtypes et héritage multiples	477
16.9.6	Contraintes portant sur des composants hérités	479
16.9.7	Répartitions multiples	479
16.9.8	Règle de répartition en sous-types	479
16.10	Identifiants	481
16.10.1	Identifiants d'un type d'entités	481
16.10.2	Identifiants d'un type d'associations	483
16.10.3	Identifiants d'un attribut complexe	486
16.11	Contraintes d'intégrité	486
16.11.1	Contraintes de domaine	487
16.11.2	Contraintes entre valeurs	487
16.11.3	Contraintes d'existence	487
16.11.4	Dépendances fonctionnelles	489
16.12	Contraintes d'intégrité additionnelles	490
16.12.1	Expression des contraintes	490
16.12.2	Remarques sur les langages d'expression de contraintes	491
16.12.3	Cycles et contraintes cycliques	492
16.12.4	Propriétés des graphes d'instances	493
16.12.5	Contraintes de redondance	494
16.13	Les noms	494
16.13.1	Règles de dénomination	494
16.13.2	Dénomination des attributs	495
16.13.3	Dénomination des types d'associations	496
16.14	Classes d'objets génériques	497
16.14.1	Type d'associations de composition	497
16.14.2	Type d'associations de matérialisation	498
16.15	Exemple de schéma Entité-association	500
16.16	Métadonnées et métaschémas	500
16.16.1	Une première approche : le catalogue d'une base de données	501
16.16.2	Métaschéma du modèle Entité-association	502
16.17	Sémantique interne du modèle Entité-association	503
16.17.1	Types d'entités et relations is-a	503
16.17.2	Types d'associations	503
16.17.3	Attributs	504
16.17.4	Attributs et rôles obligatoires	504

16.17.5Identifiants	504
16.17.6Autres contraintes	505
16.17.7Application de la théorie relationnelle au modèle Entité-association	505
16.18 Que retenir ?	506
16.19 Pour en savoir plus	506
16.20 Créer un schéma conceptuel à l'aide d'un outil (compléments)	508
16.21 Exercices	508
CHAPITRE 17 • LES DIAGRAMMES DE CLASSES UML	513
17.1 Introduction	513
17.2 UML et les bases de données	514
17.3 Les mécanismes d'extension	515
17.4 Les classes d'objets	516
17.5 Les associations binaires	516
17.6 Les associations n-aires	517
17.6.1 Multiplicité maximale (max)	518
17.6.2 Multiplicité minimale (min)	518
17.7 Les attributs	519
17.8 Les opérations	519
17.9 Les associations qualifiées	520
17.10 Les classes associations	522
17.11 Les associations d'agrégation et de composition	522
17.12 Les contraintes	523
17.13 Le langage de contraintes OCL	524
17.14 Une question de vocabulaire	527
17.15 Exemple de schéma de classes en UML	527
17.16 Le modèle de classes d'UML revisité (DB-UML)	527
17.17 Sémantique du modèle de classes DB-UML	529
17.18 Que retenir ?	531
17.19 Pour en savoir plus	531
17.20 Créer un schéma conceptuel UML à l'aide d'un outil	532
CHAPITRE 18 • ANALYSE CONCEPTUELLE DU DOMAINE D'APPLICATION	533
18.1 Introduction	533
18.2 Analyse des énoncés textuels	536
18.2.1 Proposition existentielle	536
18.2.2 Proposition relationnelle binaire	537

18.2.3 Proposition relationnelle n-aire	540
18.2.4 Intégration d'une proposition dans un schéma	544
18.3 Les patrons de conception	545
18.4 Analyse des documents structurés	547
18.4.1 Analyse des documents de l'organisation	547
18.4.2 Analyse d'un écran de saisie de données	549
18.4.3 Analyse d'un document papier	550
18.5 Dénomination des objets d'un schéma	551
18.6 Processus de correction d'un schéma	552
18.6.1 Constructions syntaxiquement incorrectes	553
18.6.2 Constructions non satisfiables et incohérentes	554
18.7 Processus de normalisation du schéma conceptuel	558
18.7.1 Simplification des constructions non minimales	560
18.7.2 Explicitation des constructions insuffisamment expressives	564
18.7.3 Élimination des redondances structurelles	570
18.7.4 Élimination des redondances internes	572
18.7.5 Autres critères de qualité	575
18.8 Évaluation d'un schéma	575
18.8.1 Paraphrase du schéma	576
18.8.2 Prototypage de la base de données	577
18.8.3 Inférence structurelle	577
18.8.4 Les heuristiques d'erreurs	578
18.8.5 Remarque	578
18.9 Intégration de schémas	579
18.9.1 Scénarios d'intégration	580
18.9.2 Principes de l'intégration binaire de schémas	580
18.9.3 Intégration basée sur les correspondances	581
18.9.4 Repérage des objets en correspondance	586
18.9.5 Unification : le cas des correspondances hétérogènes	587
18.9.6 Normalisation et résolution de conflits sémantiques	588
18.9.7 Compléments	589
18.9.8 Étude de cas	591
18.10 Analyse conceptuelle à l'aide d'un atelier	595
18.10.1 Transformation de schémas	595
18.10.2 Analyse de schémas	597
18.10.3 Intégration de schémas	597
18.11 Que retenir ?	598
18.12 Pour en savoir plus	598
18.13 Exercices	600

CHAPITRE 19 • CONCEPTION LOGIQUE D'UNE BASE DE DONNÉES RELATIONNELLE	605
19.1 Introduction	605
19.2 Le modèle logique relationnel	607
19.2.1 Le modèle relationnel de base	607
19.2.2 Le modèle relationnel enrichi	608
19.3 Représentation des attributs	610
19.3.1 Représentation d'un attribut simple	610
19.3.2 Représentation d'un attribut composé	610
19.3.3 Représentation d'un attribut multivalué	613
19.3.4 Représentation d'un attribut composé multivalué	614
19.3.5 Traduction des types et domaines de valeurs	615
19.3.6 Compléments	616
19.4 Retour aux types d'associations fonctionnels	617
19.5 Représentation des types d'associations complexes	618
19.5.1 Types d'associations n-aires, avec attributs et/ou contraintes	618
19.5.2 Représentation d'un type d'associations à rôle polymorphique	618
19.5.3 Clé étrangère et identifiant absent	620
19.6 Représentation des relations is-a	620
19.6.1 Les trois techniques de base	621
19.6.2 Représentation de relations is-a par matérialisation	622
19.6.3 Représentation de relations is-a par héritage descendant	626
19.6.4 Représentation de relations is-a par héritage ascendant	627
19.6.5 Évaluation comparative	628
19.6.6 Hiérarchies multiples et clés étrangères liées	629
19.7 Compléments	630
19.7.1 Les identifiants facultatifs	630
19.7.2 Les identifiants primaires complexes	631
19.7.3 Traduction des contraintes conceptuelles	632
19.7.4 Noms des objets	632
19.7.5 Delete et update modes des clés étrangères	632
19.7.6 Au sujet des rôles de cardinalité [1-N]	633
19.8 Traduction d'un schéma conceptuel	634
19.8.1 Choix des représentations privilégiées	634
19.8.2 Construction du plan de transformation	636
19.8.3 Limitations d'un plan de transformation	636
19.8.4 Validation d'un plan de transformation	637
19.9 Définition des vues	638
19.10 Conception logique : un exemple	638

19.11	Exploitation des constructions SQL3	640
19.11.1	Le modèle relationnel objet	642
19.11.2	Représentation des attributs complexes	642
19.11.3	Représentation des hiérarchies is-a de types d'entités	642
19.11.4	Conception logique relationnelle objet : un exemple	643
19.11.5	Extension de la méthode	643
19.12	Produire un schéma logique à l'aide d'un outil	644
19.12.1	Transformation de schémas	644
19.12.2	Analyse de schémas	646
19.13	Que retenir ?	646
19.14	Pour en savoir plus	647
19.15	Exercices	648
 CHAPITRE 20 • CONCEPTION PHYSIQUE D'UNE BASE DE DONNÉES RELATIONNELLE		653
20.1	Introduction	653
20.2	Notion d'efficacité d'une base de données	655
20.3	Conception physique d'une base de données en 15 règles	656
20.4	Détermination des index	656
20.4.1	Index induits par les structures logiques	656
20.4.2	Index induits par l'utilisation des données	657
20.4.3	Index à écarter	658
20.4.4	Maintenance du jeu d'index	659
20.5	Technologie des index	660
20.6	Sélection des espaces de stockage	661
20.7	Optimisation avancée	661
20.7.1	Restructurations sans redondance	662
20.7.2	Redondances structurelles	664
20.7.3	Dénormalisation	665
20.8	Estimation des volumes et des temps de traitement	666
20.9	Complément : modèle de choix d'un index	666
20.10	Que retenir ?	667
20.11	Pour en savoir plus	668
20.12	Exercices	668
 CHAPITRE 21 • PRODUCTION DU CODE D'UNE BASE DE DONNÉES		671
21.1	Introduction	671
21.2	Expression déclarative des structures	672
21.3	Les concepts SQL d'expression de contraintes	673

21.4	Les comportements non standard	675
21.5	Les contraintes d'intégrité de base	677
21.6	Les contraintes d'intégrité additionnelles	678
21.7	Les contraintes dérivées du modèle Entité-association	681
21.8	Les contraintes d'intégrité dynamiques	682
21.9	Traduction des relations is-a	683
21.9.1	Gestion à partir des tables de base	683
21.9.2	Gestion via des vues modifiables	685
21.10	Traduction des vues	688
21.11	Que retenir ?	689
21.12	Génération du code à l'aide d'un outil	689
21.13	Exercices	690
CHAPITRE 22 • RÉTRO-INGÉNIERIE D'UNE BASE DE DONNÉES		693
22.1	Introduction	693
22.2	Méthode générale de rétro-ingénierie	697
22.3	Extraction physique	698
22.4	Reconstruction du schéma logique	699
22.4.1	Les sources et leurs techniques d'analyse	699
22.4.2	Recherche des constructions implicites	703
22.5	Conceptualisation du schéma logique	705
22.6	Que retenir ?	709
22.7	Pour en savoir plus	710
22.8	Rétro-ingénierie à l'aide d'un outil	710
CHAPITRE 23 • ÉTUDE DE CAS		711
23.1	Les animaux du zoo	711
23.1.1	Énoncé	711
23.1.2	Construction du schéma conceptuel	712
23.1.3	Production du schéma de tables et du code SQL	713
23.1.4	Exploitation	715
23.2	Autres études de cas	718
CHAPITRE 24 • BIBLIOGRAPHIE		719
CHAPITRE 25 • INDEX		727

ANNEXES

Les annexes sont disponibles en ligne sur le site de l'ouvrage, à l'adresse
<http://www.info.fundp.ac.be/libd> > Ouvrages
ou sur le site www.dunod.com sur la page dédiée à cet ouvrage

ANNEXE A1 • MOTIVATION ET INTRODUCTION

ANNEXE A2 • CONCEPTS DES BASES DE DONNÉES

ANNEXE A3 • MODÈLE RELATIONNEL ET NORMALISATION

ANNEXE A4 • ÉLÉMENTS DE TECHNOLOGIE DES BASES DE DONNÉES

ANNEXE A5 • LES SYSTÈMES DE GESTION DE BASES DE DONNÉES

ANNEXE A6 • LE LANGAGE SQL DDL

ANNEXE A7 • LE LANGAGE SQL DML (1)

ANNEXE A8 • LE LANGAGE SQL DML (2)

ANNEXE A9 • LE LANGAGE SQL AVANCÉ

ANNEXE A10 • LES BASES DE DONNÉES NON RELATIONNELLES

ANNEXE A11 • CONSTRUCTION D'UNE BASE DE DONNÉES

ANNEXE A12 • LE MODÈLE ENTITÉ-ASSOCIATION DE BASE

ANNEXE A13 • ÉLABORATION D'UN SCHÉMA CONCEPTUEL

ANNEXE A14 • PRODUCTION DU SCHÉMA DE LA BASE DE DONNÉES

ANNEXE A15 • MÉTHODOLOGIE DES BASES DE DONNÉES

ANNEXE A16 • LE MODÈLE ENTITÉ-ASSOCIATION ÉTENDU

ANNEXE A17 • LES DIAGRAMMES DE CLASSES UML

ANNEXE A18 • ANALYSE CONCEPTUELLE DU DOMAINE D'APPLICATION

ANNEXE A19 • CONCEPTION LOGIQUE D'UNE BASE DE DONNÉES RELATIONNELLE

ANNEXE A20 • CONCEPTION PHYSIQUE D'UNE BASE DE DONNÉES RELATIONNELLE

ANNEXE A21 • PRODUCTION DU CODE D'UNE BASE DE DONNÉES

ANNEXE A22 • RÉTRO-INGÉNIERIE D'UNE BASE DE DONNÉES

ANNEXE A23 • ÉTUDES DE CAS

ANNEXE A24 • SQL, LES ENSEMBLES ET LA LOGIQUE

ANNEXE A25 • TRANSFORMATIONS DE SCHÉMAS

ANNEXE A26 • CONCEPTION LOGIQUE D'UNE BASE DE DONNÉES RELATIONNELLE OBJET

ANNEXE A27 • APPLICATIONS AVANCÉES DES BASES DE DONNÉES

ANNEXE A28 • INTRODUCTION À SQL`fast`

Avant-propos

Cet ouvrage est le fruit de plus de trente ans d'enseignement, de recherche et de pratique, tant à l'université que dans les entreprises et les administrations. J'y ai rassemblé et structuré les concepts, les techniques et les méthodes du domaine des bases de données qui me sont apparus les plus *utiles* à l'informaticien, depuis l'étudiant jusqu'au praticien en passant par l'utilisateur motivé.

Le lecteur ne trouvera ici que ce qu'il est susceptible de rencontrer dans sa pratique.¹ Même les sections un peu plus théoriques ont leur utilité, soit pour résoudre un problème concret de manière rigoureuse, soit pour mieux comprendre l'un ou l'autre concept délicat ou controversé. En revanche, le lecteur intéressé par les détails techniques propres à un SGBD particulier trouvera ceux-ci dans l'abondante littérature qui leur est consacrée, tant sur le web que dans les librairies.

J'ai donc résisté à la tentation de l'*encyclopédisme*, qui voudrait que je traite ici, comme il est souvent d'usage dans un ouvrage généraliste, de thèmes tels que les dépendances multivaluées, la quatrième forme normale, le calcul relationnel (ou Datalog), l'optimisation des requêtes, les bases de données orientées objet, déductives, XML ou multimédias. Malgré leur incontestable intérêt je les ai écartés de la table des matières dans la mesure où ils restent marginaux par rapport au cœur de métier de l'informaticien.

Avant de décrire plus en détail les objectifs et le mode d'emploi de l'ouvrage, je voudrais remercier tous ceux qui, jadis ou aujourd'hui, de loin ou de près, peu ou prou, ont contribué à son élaboration. Certains ont été, ou sont encore, des collaborateurs de recherche. D'autres sont des collègues avec lesquels j'ai à l'occasion des discussions sur des points particuliers techniques ou théoriques.

Je pense en particulier, dans le désordre, à Véronique Goemans, Yves Delvaux, Lysiane Goffaux, Jean-Pierre Thiry, Alain Delcourt, Benoît Vanhoute, Carine Charlot, Bertrand Mignon, Catherine Tonneau, Muriel Chandelon, Michel Joris,

1. À une exception près, que le lecteur est invité à identifier !

Mario Cadelli, Bernard Decuyper, Olivier Marchand, Jean Henrard, Jean-Marc Hick, Vincent Englebert, Didier Roland, Alain Gofflot, Pierre Delvaux, Anne-France Brogneaux, Arnaud Deflorenne, Stéphane Bodart, Majid Chougrani, Philippe Thiran, Virginie Detienne, Denis Zampunieris, Christine Delcroix, Aurore François, Fabrice Estiévenart, Frédéric Vesentini, Jean-Roch Meurisse, Julien Vilz, Ravi Ramdoyal, Anthony Cleve, Jonathan Lemaitre, Karin Becker, Yannis Tzitzikas, Rokia Bendaoud, Cédric Libert, Gilles Perrouin, François Bodart, Patrick Heymans, Wim Vanhoof et Yves Pigneur. Une pensée toute particulière à Anne-France, Vincent et Anthony, pour leur aide dans le décryptage de SQL lorsqu'on tente de le pousser dans ses derniers retranchements. De nombreuses notes de cet ouvrage leur sont dues.

Je remercie tous les étudiants auxquels j'ai enseigné dans les facultés d'informatique, d'économie, de gestion et de droit, cette discipline particulièrement riche et passionnante qu'est celle des bases de données. Leurs questions, leurs suggestions et leurs difficultés sont à l'origine d'une partie significative de cet ouvrage. Celui-ci leur est tout naturellement dédié. Qu'ils y pensent lorsque, en fin d'année, ils seront tentés de le vouer, lui et son auteur, aux gémonies !

Merci et bravo à ReveR et à ses aventuriers, qui ont à ce point cru en la valeur des résultats des recherches de notre laboratoire qu'ils n'ont pas hésité à fonder en 2003, avec l'aide d'autres illuminés de la même eau, une société commerciale unique au monde. Ils méritent toute mon admiration.

Merci enfin aux maisons d'édition Masson, InterEditions et Dunod pour leur aide et la confiance qu'elles m'ont accordée au fil des années.

Objectifs et structure de l'ouvrage

L'ouvrage a pour objectif d'offrir au lecteur un texte d'introduction aux bases de données à la fois pratique, moderne et détaillé. Les concepts et les problèmes qu'il aborde sont ceux que le lecteur, quel que soit son niveau de départ et quelles que soient ses attentes (il s'adresse aussi bien à l'*étudiant* qu'au *praticien*) rencontrera sur le terrain. Sa structure générale suit une approche rationnelle : pour apprendre à *utiliser* les bases de données il faut au préalable en *comprendre* les concepts et pour apprendre à *construire* des bases de données il faut savoir les *utiliser*. L'ouvrage est par conséquent structuré en trois parties :

- La première, intitulée ***Concepts des bases de données***, est constituée des quatre chapitres qui décrivent les concepts fondamentaux des bases de données : les structures de données, leur implémentation, les systèmes de gestion de bases de données et, pour les lecteurs plus curieux, des éléments du modèle relationnel et de la normalisation. Cette partie est complétée d'un chapitre sur les bases de données non relationnelles, y compris les SGBD dits NoSQL, avec une extension vers les *chaînes de blocs*.
- La deuxième partie, ***Utilisation des bases de données***, décrit les différents aspects du langage SQL, depuis les formes et fonctions élémentaires jusqu'aux fonctions avancées du modèle relationnel objet. Cette partie comprend quatre chapitres.

- La troisième partie, *Développement des bases de données*, décrit de manière détaillée les techniques et les méthodes de construction de bases de données relationnelles et relationnelles objet. On y trouve également une introduction à la rétro-ingénierie des bases de données. Cette partie constituée de treize chapitres est suivie d'une étude de cas complètement résolue et implémentée.

Chaque chapitre est prolongé par une annexe qui propose des exercices additionnels ainsi que des matériaux complémentaires. D'autres annexes transversales contiennent des exposés de mise à niveau, des applications et des manuels pour les outils utilisés dans cet ouvrage. Ces annexes et ces outils, ainsi que des matériaux complémentaires, destinés à l'étudiant, à l'enseignant et au praticien, sont disponibles en ligne sur le site de l'ouvrage.

Les options de cet ouvrage

L'ouvrage a été structuré et rédigé sur la base de quelques principes suggérés par l'expérience et l'étude des besoins actuels en matière de bases de données.

Une approche pratique et pédagogique. Destiné à un public diversifié et évolutif, l'ouvrage adopte une démarche progressive et propose plusieurs parcours de lecture. Le contenu de l'ouvrage est articulé autour de problèmes et de questions effectivement rencontrés en pratique. Les aspects théoriques sont proposés dans la mesure où ils permettent de résoudre aisément et efficacement ces problèmes. Le texte inclut de nombreux exemples, exercices et études de cas, accompagnés de leurs solutions. Les concepts sont illustrés par des outils disponibles gratuitement (SQLfast et DB-Main) que le lecteur est invité à utiliser. La plupart des chapitres se clôturent par une synthèse (*Que retenir ?*), par des pistes d'approfondissement et un état de l'art (*Pour en savoir plus*) et par un jeu d'exercices corrigés.

Un contenu et une approche modernes. L'ouvrage est destiné aux lecteurs confrontés aux problèmes que posent les systèmes d'information et les technologies d'aujourd'hui. Il inclut des résultats récents issus de la recherche et de l'industrie. Ainsi, des méthodes pratiques relatives au *calcul des identifiants*, à la *normalisation de schémas conceptuels*, à la construction de *bases de données relationnelles objet*, aux *bases de données NoSQL*, aux *chaînes de blocs* et à la *rétro-ingénierie* ont-elles été incluses dans cet ouvrage. Les méthodes d'ingénierie qu'il décrit sont basées sur des approches transformationnelles et peuvent être mises en œuvre à l'aide d'outils (les AGL).

Une approche rigoureuse. L'ouvrage ne se contente pas de décrire les concepts et les méthodes à la manière d'une encyclopédie. Il construit progressivement un édifice cohérent et raisonné. Les techniques et les méthodes de résolution de problèmes sont présentées, non comme des recettes prêtes à l'emploi, mais comme le résultat naturel de raisonnements objectifs et rigoureux. Le lecteur peut ainsi comprendre la rationalité de ces techniques et méthodes et en appréhender les limites. Il peut dès lors se les approprier et les adapter à ses propres besoins. Certaines parties délicates de l'exposé sont soutenues par des éléments

plus théoriques qui en garantissent la validité. Ainsi en est-il de la théorie relationnelle et de la normalisation (chapitre 3).

Une approche critique. Les concepts, les technologies et les méthodes ne sont pas livrées comme allant de soi. Leur histoire, leurs points forts et leurs faiblesses sont largement discutés. Pour citer quelques exemples, les normes SQL2 et SQL3, les valeurs null, le modèle relationnel objet, les modèles NoSQL, le modèle Entité-association, les diagrammes de classes d'UML et les options choisies par les implémenteurs de SGBD sont analysés et critiqués, souvent de manière détaillée.

Publics cibles et parcours personnalisés

Cet ouvrage est destiné à la fois aux *étudiants et enseignants*, et aux *professionnels* de l'informatique, auxquels il propose deux parcours différents :

- **Parcours introductif.** Le lecteur ou l'étudiant débutant se contentera dans un premier temps de la suite des neuf chapitres suivants :
 - *Concepts* : chapitres 1 et 2.
 - *Utilisation* : chapitres 6, 7 et 8 consacrés au langage SQL.
 - *Développement* : chapitres 11, 12, 13 et 14. Étude de cas : chapitre 23.

Ces chapitres constituent un parcours d'initiation autonome couvrant les aspects fondamentaux du domaine (figure 1). Il correspond à un module d'introduction aux bases de données de 30 heures, travaux pratiques non compris.

- **Parcours complet.** Il sera proposé au lecteur ou à l'étudiant à la recherche d'un exposé plus approfondi et plus technique du domaine. Les trois parties de l'ouvrage sont constituées comme suit :
 - *Concepts* : chapitres 1 et 2 + chapitres 3, 4, 5 et 10
 - *Utilisation* : chapitres 6, 7, 8 et 9
 - *Développement* : chapitres 15, 16, 17, 18, 19, 20 et 21. Étude de cas : chapitre 23.

Ces chapitres constituent un parcours de lecture complet couvrant en détail l'ensemble du domaine des bases de données (figure 2). Il correspond à un module approfondi de 60 heures, travaux pratiques non compris, ou encore à un module de 45 heures pour l'étudiant qui a suivi le parcours introductif.

En outre, il est possible de définir des parcours plus spécifiques et d'exploiter des matériaux additionnels.

- L'annexe A24 (*SQL, les ensembles et la logique*) constitue une mise à niveau utile pour aborder les chapitres 3, 16 et 18. De même, l'annexe A25 détaille l'approche transformationnelle utilisée informellement dans les chapitres 13, 14, 16, 18 et 19. L'annexe A28 est une introduction à SQLfast, outil dont nous parlerons plus loin.
- Les exercices et les applications sont essentiels pour maîtriser les concepts et les techniques étudiés. On les trouvera à la fin de chaque chapitre ainsi que dans leurs annexes respectives. En outre, les annexes A23 et A27 présentent des applications et des études de cas de complexité variable.

- Sur le plan de la méthodologie (troisième partie), les chapitres 15, 16, 18 et 22 (*rétro-ingénierie*) apportent aux *concepteurs*, *consultants* et *administrateurs de données* une spécialisation en modélisation des données. Celle-ci pourra être complétée par l'étude critique d'UML (chapitre 17).

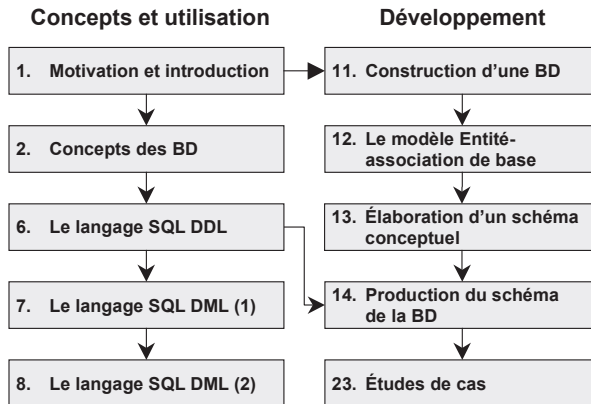


Figure 1 - Parcours de lecture introductif

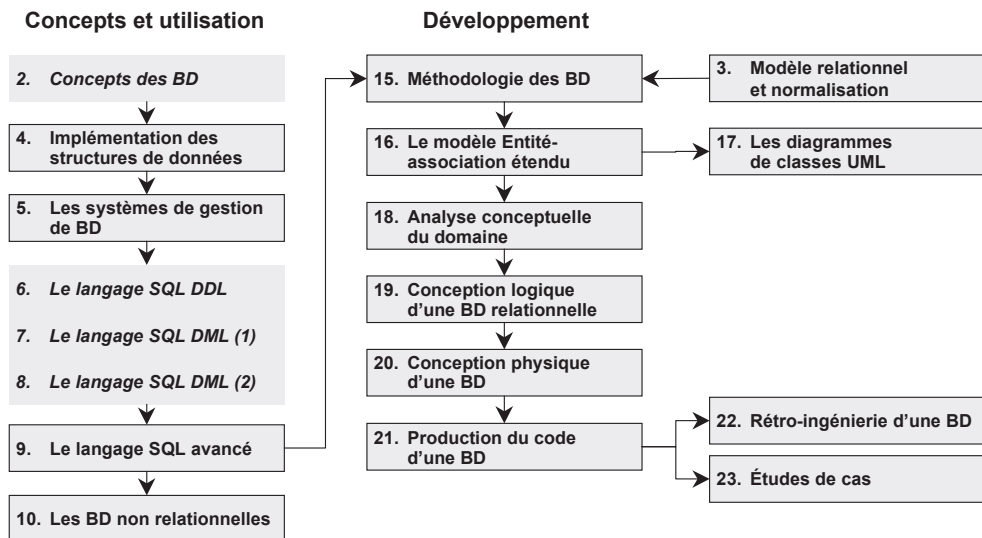


Figure 2 - Parcours de lecture complet

Cet ouvrage est accompagné et complété d'un site web dont le contenu est en libre accès. Celui-ci met en ligne les annexes A01 à A28 et des matériaux complémentaires : tutoriels, logiciels recommandés dans l'ouvrage, matériaux didactiques, documents d'approfondissement et errata (que le lecteur attentif consultera régulièrement). Ces matériaux sont disponibles à l'adresse <http://www.info.fundp.ac.be/libd>

(onglet *Books/Ouvrages*) ou sur le site de Dunod (<http://www.dunod.com/contenus-complementaires/9782100727063>) sur la page dédiée à l'ouvrage.

Au sujet de la 4^e édition

Cette édition constitue une révision majeure de l'édition précédente. Outre de nombreuses mises à jour, corrections et améliorations de présentation, elle inclut de nouveaux matériaux conformes à l'état de l'art en 2018.

- *Chapitre 3.* Les relations en N1NF et les opérateurs *nest* et *unnest*.
- *Chapitre 4.* Les index sur expression. Quand créer un index ? Les prédicteurs à hachage (filtres de Bloom). La signature à hashage (*secure hashing*). Identifiants universels (UUID). Futurologie : les mémoires ADN, les mémoires atomiques, les bases de données quantiques..
- *Chapitre 5.* Le phénomènes des données instables (*dirty data*). Les niveaux d'isolation. La technique de *write-ahead logging*. Les fichiers différentiels.
- *Chapitre 7.* Les conditions sur n-uplets. Les opérateurs logiques *ou exclusif*, *implication*, *équivalence*. Notion de collation.
- *Chapitre 8.* *Zero-table select*. *update* par assignation de n-uplet. *select for update*. Nouveaux cas de non déterminisme.
- *Chapitre 10.* Les chaînes de blocs. BitCoin. Ethereum.
- *Chapitre 18.* Un générateur de prototype de base de données est disponible dans la distribution de SQLfast.
- *Chapitre 20.* Affinement de la procédure de conception du schéma physique (index augmentés, ordre de déclaration des colonnes)
- *Chapitre 22.* Des outils d'analyse d'une base de données existante est disponible dans la distribution de SQLfast.
- *Chapitre 23.* Extension de l'étude de cas "Les animaux du zoo". Une application de gestion complète est développée (disponible dans la distribution de SQLfast). La seconde étude de cas a été reportée en annexe.

Outre l'atelier **DB-MAIN** proposé en support des activités de conception, une nouvelle version de l'outil **SQLfast** est associé à l'ouvrage pour l'apprentissage du langage SQL et pour le développement d'applications interactives de bases de données. Ne nécessitant aucune installation, cet interpréteur, rédigé en Python, inclut le SGBD SQLite et une interface graphique conviviale. Il permet de rédiger et d'exécuter des scripts SQLfast interactifs de complexité quelconque. Il inclut une série de tutoriels inspirés de cet ouvrage. L'annexe A28 est consacrée à une introduction à SQLfast. **DB-MAIN** et **SQLfast** sont disponibles gratuitement sur le site de l'ouvrage.

Jean-Luc Hainaut
Namur, le 31 mars 2018

Chapitre 1

Motivation et introduction

Ce chapitre introduit de manière intuitive le lecteur au concept de *donnée*. Il montre que le contenu des documents que nous produisons et utilisons quotidiennement sont naturellement structurés d'une manière qui suggère leur introduction dans un ordinateur. Ce processus d'analyse conduit à la notion de *base de données*. On identifie les principales questions qui se posent en pratique dans le monde des bases de données et on rappelle quelques étapes de leur histoire.

1.1 DES DONNÉES ET DES HOMMES

Il n'est pas d'interactions avec nos collègues, avec des commerçants ou des administrations lors desquelles nous n'échangeons pas des **informations** sous forme de **données**. La consultation du site d'un fournisseur d'équipement multimédia, le suivi de notre commande de matériel informatique, la déclaration de naissance du petit dernier, la lecture inquiète des notes obtenues en fin d'année scolaire (les nôtres ou celles de notre fainéant de fiston), nos missions dans WoW, la consultation des films proposés ce samedi dans notre quartier, une visite chez notre médecin, nos communications téléphoniques, le téléchargement de l'épisode 3 de la saison 2 ou l'usage d'un GPS n'en sont que quelques exemples parmi les plus évidents.

Où vont les données que nous fournissons et d'où viennent celles qui nous sont livrées ? Elles sont évidemment stockées dans un ordinateur. Dès que ces données présentent un certain degré de complexité, qu'elles sont soumises à des contraintes de qualité, qu'elles doivent résister à des incidents ou à des attaques, qu'elles sont partagées par plusieurs usagers ou qu'elles sont utilisées dans plusieurs départe-

ments pour des fonctions différentes, il est hautement probable qu'elles sont stockées dans une **base de données**.

C'est désormais une évidence, nous sommes quotidiennement, en tant que simples usagers ou dans nos activités professionnelles, et souvent sans en être conscients, en relation étroite avec des milliers de bases de données que nous alimentons et que nous consultons.

Dans les entreprises, ces données, dites permanentes, constituent le matériau de base à partir duquel vont s'élaborer la plupart des applications informatiques quelle qu'en soit la taille. À tel point d'ailleurs que certaines de celles-ci se réduisent pratiquement à la gestion et la consultation de 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.

Les données nécessaires aux applications informatiques sont le plus généralement réparties 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 une ressource commune qu'on appelle une **base de données**. Gérer de telles données n'est pas une mince affaire. Garantir leur qualité (retrouve-t-on ce qu'on y 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 sans conflits, 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 sites web. Ils existent sur toutes les plates-formes, depuis les gros ordinateurs (*mainframes* ou serveurs de grande puissance) jusqu'au PC portable et au *smartphone*, à la carte à puce et aux décodeurs TNT USB.

1.2 UNE PREMIÈRE APPROCHE DES DONNÉES

Sur votre bureau, là à gauche, la pile des bons de commande de la semaine que vos clients vous ont envoyés par courrier postal (figure 1.1). En face de vous l'ordinateur que vous venez d'acquérir grâce aux premiers bénéfices de votre petite entreprise de

matériaux de construction. Votre intention est d'introduire celle-là (la pile) dans celui-ci (l'ordinateur).

Commande N° : 30188		Date : 2/1/2009		
Numéro client B512				
Nom GILLET				
Adresse 14, r. de l'Eté				
Localité Toulouse				
N° PRODUIT	LIBELLE PRODUIT	PRIX	QUANTITE	SOUS-TOTAL
CS464	CHEV. SAPIN 400x6x4	220	180	39600
PA45	POINTE ACIER 45 (20K)	105	22	2310
PA60	POINTE ACIER 60 (10K)	95	70	6650
PH222	PL. HETRE 200x20x2	230	92	21160
TOTAL COMMANDE				69720

Figure 1.1 - La pile de bons de commande de la semaine

Vous pensez dans un premier temps à **scanner** ces documents, mais vous réalisez bien vite que ce procédé ne vous apporterait pas grand-chose d'utile, le résultat ne se prêtant pas à des manipulations ultérieures, telles que l'édition de factures, la gestion des stocks ou votre comptabilité.

L'idée de les copier dans un **document Word** ne vous semble guère plus sérieuse : mise en page laborieuse et exploitation des données pratiquement impossible.

Vous pourriez introduire leur contenu dans une **feuille Excel**, mais la structure des données vous paraît trop complexe à traduire dans un simple tableau. Et puis comment cette solution supportera-t-elle la croissance rapide que vous prévoyez pour votre entreprise ?

Vous n'y échapperez pas : il faudra vous résoudre à introduire ces données dans une **base de données** ! A ce stade de la lecture de ce chapitre, vous savez déjà que les données s'organisent en *fichiers*, constitués chacun d'*enregistrements*.

Prenons le premier document pour en étudier la structure (figure 1.2). Nous y repérons immédiatement un bloc de données relatives au client : *Numéro client*, *Nom*, *Adresse* et *Localité*. La commande elle-même est caractérisée par un *Numéro*, une *Date* (en haut) et un *Total Commande* (en bas). Le tableau de la moitié inférieure regroupe les détails de la commande : pour chacun on y trouve le *Numéro du produit*, son *Libellé*, son *Prix (à l'unité)*, la *Quantité (commandée)* et le *Sous-total*. Nous avons identifié dans le bon de commande trois catégories de données, relatives respectivement aux *clients*, aux *commandes* et aux *détails* de commande.

Commande N° : 30188		Date : 2/1/2009		
Numéro client		B512		
Nom		GILLET		
Adresse		14, r. de l'Eté		
Localité		Toulouse		
N° PRODUIT	LIBELLE PRODUIT	PRIX	QUANTITE	SOUS-TOTAL
CS464	CHEV. SAPIN 400x6x4	220	180	39600
PA45	POINTE ACIER 45 (20K)	105	22	2310
PA60	POINTE ACIER 60 (10K)	95	70	6650
PH222	PL. HETRE 200x20x2	230	92	21160
TOTAL COMMANDE				69720

données du client

données de la commande

données d'un détail

Figure 1.2 - Analyse des données du premier bon de commande

Nous décidons logiquement de ranger ces données dans trois fichiers qui accueilleront respectivement les données sur le *client*, les données sur la *commande* et les données sur les *détails* (figure 1.3). Nous procéderons à l'encodage des autres bons de commande selon cette même distribution des données.

données de la commande			données du client			
NCOM	DATECOM	TOTAL-COMMANDE	NCLI	NOM	ADRESSE	LOCALITE
30188	2/1/2009	69720	B512	GILLET	14, r. de l'Eté	Toulouse

données des détails				
NPRO	LIBELLE	PRIX	QCOM	SOUS-TOTAL
CS464	CHEV. SAPIN 400x6x4	220	180	39600
PA45	POINTE ACIER 45 (20K)	105	22	2310
PA60	POINTE ACIER 60 (10K)	95	70	6650
PH222	PL. HETRE 200x20x2	230	92	21160

Figure 1.3 - Extraction des données du premier bon de commande

On avance bien sûr, mais à y regarder de plus près, le résultat n'est pas tout à fait satisfaisant :

1. D'une part, certaines données sont *calculées*, de sorte qu'il n'est pas absolument nécessaire de les stocker. Nous pouvons ainsi supprimer les champs SOUS-TOTAL (= PRIX x QCOM) et TOTAL-COMMANDE (= somme des valeurs de SOUS-TOTAL).
2. D'autre part, nous observons qu'il est impossible de reconstituer les documents initiaux. Comment en effet repérer le client d'une commande, puisque nous avons extrait et rangé ailleurs le fragment décrivant ce client ? De même, comment identifier la commande de laquelle nous avons extrait un détail ? Il est évident qu'il faut prévoir dans les fragments relatifs aux commandes et aux détails des *données de référence* nécessaires à ces liaisons.

Nous complétons donc la structure des données en introduisant dans le fichier des commandes un champ NCLI indiquant le numéro du client et dans le fichier des détails un champ NCOM spécifiant la commande de chaque détail (figure 1.4). Nous en profitons pour introduire également les données de la commande 30179.

Remarquons en passant que nous avons fait l’hypothèse qu’on pouvait *identifier* un client par son numéro NCLI et une commande par son numéro NCOM, ce qui est bien naturel. Pour les détails, nous réexaminerons cette question plus tard.

données des commandes			données des clients			
NCOM	NCLI	DATECOM	NCLI	NOM	ADRESSE	LOCALITE
30188	B512	2/1/2009	B512	GILLET	14,r. de l'Eté	Toulouse
30179	C400	22/12/2008	C400	FERARD	63,r. du Tertre	Poitiers

données des détails				
NCOM	NPRO	QCOM	LIBELLE	PRIX
30188	CS464	180	CHEV. SAPIN 400x6x4	220
30188	PA45	22	POINTE ACIER 45 (20K)	105
30188	PA60	70	POINTE ACIER 60 (10K)	95
30188	PH222	92	PL. HETRE 200x20x2	230
30179	CS262	60	CHEV. SAPIN 200x6x2	75
30179	PA60	20	POINTE ACIER 60 (10K)	95

Figure 1.4 - Fichiers de données améliorés

Le résultat semble prometteur, mais il reste un malaise : pour tous les détails qui référencent le même produit (ceux qui ont la même valeur du champ NPRO), on reprend dans le fichier des détails le LIBELLE et le PRIX de ce produit, créant ainsi d’importantes redondances non seulement inutiles mais dangereuses. Quelle garantie en effet avons-vous que tous les détails d’un même produit indiqueront exactement le même *Libellé* et le même *Prix* ? Il serait plus intéressant de construire un quatrième fichier destiné à recueillir les données spécifiques aux produits. Nous retirons donc celles-ci du fichier des détails (en y laissant cependant une copie du champ NPRO pour ne pas retomber dans le piège précédent) et nous les rangeons dans ce nouveau fichier. C’est ce qui est proposé à la figure 1.5.

données des commandes			données des clients			
NCOM	NCLI	DATECOM	NCLI	NOM	ADRESSE	LOCALITE
30188	B512	2/1/2009	B512	GILLET	14,r. de l'Eté	Toulouse
30179	C400	22/12/2008	C400	FERARD	63,r. du Tertre	Poitiers

données des détails			données des produits		
NCOM	NPRO	QCOM	NPRO	LIBELLE	PRIX
30188	CS464	180	CS464	CHEV. SAPIN 400x6x4	220
30188	PA45	22	PA45	POINTE ACIER 45 (20K)	105
30188	PA60	70	PA60	POINTE ACIER 60 (10K)	95
30188	PH222	92	PH222	PL. HETRE 200x20x2	230
30179	CS262	60	CS262	CHEV. SAPIN 200x6x2	75
30179	PA60	20			

Figure 1.5 - Les données de bons de commande distribuées dans quatre fichiers

1.3 UTILISER UNE BASE DE DONNÉES

Nous venons de construire une base de données en bonne et due forme, respectant les bonnes pratiques en la matière. La figure 2.8, que le lecteur est invité à consulter, montre ce qu'on obtiendrait après avoir enregistré les données de quelques commandes supplémentaires. Ainsi mises en forme, ces données permettent de répondre simplement à une infinie variété de questions. C'est à l'aide du langage SQL (*Structured Query Language*) que nous étudierons dans cet ouvrage, que nous pourrions interagir avec cette base de données, c'est-à-dire créer, consulter, modifier et supprimer les données. Par exemple, l'extraction des données sur les *clients de Toulouse* s'obtiendra par l'instruction suivante, dont la compréhension à ce stade ne pose pas de problèmes :

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

On y spécifie les données désirées (NCLI, NOM, ADRESSE), la table dont elles sont extraites (CLIENT) et la condition que doivent vérifier les lignes qui nous intéressent (LOCALITE = 'Toulouse'). La recherche les *commandes des clients de Toulouse* est un peu plus délicate mais reste simple (elle sera étudiée dans la section 7.6)¹ :

```
select NCOM
from   COMMANDE
where  NCLI in (select NCLI
                from   CLIENT where LOCALITE = 'Toulouse')
```

Ces données vont pouvoir servir à d'innombrables applications, y compris certaines auxquelles nous ne pensons pas aujourd'hui. Citons quelques exemples.

- Vérifier les commandes du jour : le client est-il déjà enregistré ? Son adresse n'a-t-elle pas changé ? Les produits référencés existent-ils ?
- Produire les factures.
- Calculer votre chiffre d'affaires hebdomadaire, mensuel, annuel.
- Préparer votre déclaration de revenu.
- Étudier la répartition géographique de votre chiffre d'affaires.
- Identifier les produits qui se vendent le mieux.
- Produire le chiffre d'affaires de chaque produit par localité et par quadrimestre.

Avec un peu d'imagination et en acceptant d'ajouter quelques données indispensables (la quantité en stock de chaque produit par exemple), voire une ou deux nouvelles tables, vous pourriez ajouter à cette liste quelques dizaines d'applications supplémentaires toutes utiles au fonctionnement et à la gestion de votre entreprise.

1. Un mot d'explication cependant. Cette requête demande d'extraire (**select**) les valeurs de NCOM des lignes de la table COMMANDE dont la valeur de NCLI apparaît dans un ensemble défini comme suit : l'ensemble des valeurs de NCLI extraites des lignes de CLIENT dont la colonne LOCALITE a la valeur 'Toulouse'.

On pourrait penser que ces applications vont exiger la rédaction de programmes volumineux et complexes, rédigés dans des langages ésotériques réservés à des experts informaticiens. La vérité est heureusement plus nuancée. Voici par exemple à quoi ressemblerait la requête rédigée dans le langage SQL qui calcule la répartition de votre chiffre d'affaires par localité et par produit :

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

Ce programme ne comporte que quatre lignes ! Bien sûr, à ce stade, vous trouvez ce texte obscur et vous vous sentez incapable d'en faire autant. Détrompez-vous : c'est l'objectif de la deuxième partie de cet ouvrage (en particulier la section 8.9) que de vous amener à pouvoir résoudre en SQL ce type de problèmes et bien d'autres plus complexes encore. Sachez que de nombreuses applications indispensables à votre entreprise peuvent se satisfaire de quelques lignes rédigées en SQL.

Observons enfin qu'à aucun moment de l'élaboration de cette base de données nous n'avons pris en compte les applications auxquelles ces données vont servir. Nous avons simplement cherché à décrire des objets de votre environnement à partir de leur représentation naturelle. Si l'analyse a été bien conduite, alors le résultat devra satisfaire les besoins en information, actuels et futurs, de toutes les activités de votre petite entreprise.

1.4 PREMIÈRES CONCLUSIONS

Que retirer de cette expérience ? Nous avons sans en avoir l'air rencontré quelques-uns des principes les plus importants des bases de données. Avant de les énumérer, il faut savoir qu'on utilise en général le terme de *table* au lieu de celui de *fichier* et celui de *ligne* au lieu de celui d'*enregistrement*. C'est ce que nous ferons désormais :

1. Une *base de données* est constituée d'un ensemble de *tables*.
2. Chaque *table* contient les données relatives à des *entités* de même nature.
3. Chaque *ligne* d'une table reprend les données relatives à une entité.
4. Chaque *colonne* d'une table décrit une propriété commune des entités.
5. Les lignes d'une table sont *distinctes*. C'est sur une (voire plusieurs) colonne(s) que se joue cette unicité. Le jeu de colonnes dont les valeurs sont uniques constitue un *identifiant* de la table.
6. Les lignes d'une table peuvent faire *référence* chacune à une ligne d'une autre table, où des informations sur une entité associée peuvent être obtenues. Les colonnes de référence s'appellent *clé étrangère* vers cette autre table.
7. On évite d'enregistrer des données qu'il est possible de *calculer* à partir d'autres données enregistrées.
8. Par application du principe 2, on ne conserve pas dans une même table des

données relatives à des entités de nature différente. On provoquerait en effet des *redondances* nuisibles à plus d'un titre.

9. Le langage SQL permet de rédiger de manière simple et concise des *requêtes* d'extraction et de calcul de données de complexité quelconque.
10. On ne construit pas une base de données pour satisfaire les besoins immédiats d'une application particulière. Une base de données est conçue pour *représenter* le plus *fidèlement* possible les informations relatives à une activité humaine, socio-économique, culturelle ou technique, mais elle est indépendante de ce qu'on fera de ces données, d'autant plus que les applications futures, que la base de données devra aussi satisfaire, ne sont pas nécessairement connues.
11. La construction d'une base de données exige une *analyse* rigoureuse et attentive. Il est cependant possible à tout moment d'ajouter de nouvelles tables et de nouvelles colonnes afin de suivre l'évolution des besoins des utilisateurs.

1.5 LES SYSTÈMES DE GESTION DE BASES DE DONNÉES

On conçoit aisément que la gestion d'une base de données, sa consultation et, d'une manière générale, la manipulation des données qu'elle contient, constituent des opérations complexes. C'est la raison pour laquelle on fera appel à des logiciels spécialisés appelés **systèmes de gestion de bases de données** (SGBD). Le lecteur aura peut-être entendu parler d'Access, Oracle, DB2, SQL Server ou MySQL, qui sont, parmi les nombreux SGBD du marché, les plus représentatifs. Ces logiciels offrent un ensemble de fonctions permettant la définition, l'exploitation et la gestion de tables et de leur contenu. Mentionnons quelques fonctions offertes par les SGBD relationnels (elle seront étudiées plus en détail, notamment au chapitre 5) :

- **Organisation des données.** Le SGBD organise les données d'une base de données sous la forme de tables et de colonnes. Il vérifie les propriétés d'unicité (pas deux lignes de la *table des produits* ayant la même valeur de NPRO) et de référence (à toute valeur de NPRO dans la *table des détails* doit correspondre une ligne dans la *table des produits*) de certaines colonnes.
- **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'unicité et de référence qui ont été déclarées.
- **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.
- **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.

- **Autres fonctions.** Les SGBD offrent d'autres fonctions telles que la protection contre les incidents, la gestion des accès concurrents et le contrôle des accès.

1.6 LES DÉFIS DES BASES DE DONNÉES D'AUJOURD'HUI

Les données sont au cœur des grandes applications informatiques mais elles sont aussi essentielles à la plupart de nos activités et des interactions que nous avons, comme professionnels ou simples citoyens, avec tous nos partenaires et correspondants. Citons quelques défis qui se posent dès aujourd'hui :

- **Multiplicité des types de données.** Les données prennent aujourd'hui des formes très variées, souvent bien éloignées des simples formats que nous avons observés précédemment. Pensons par exemple aux données multimédias (images, vidéo, musique), aux données géographiques (GPS), aux données historiques, aux données semi-structurées (textes, documents, formulaires).
- **L'information incorrecte et incomplète.** Nous sommes submergés d'informations, souvent de piètre qualité. Cette observation est également vraie pour les bases de données, dont le contenu comporte souvent de nombreuses données erronées. Comment les exploiter avec rigueur et précision. Comment produire de l'information correcte à partir de données imprécises, voire incorrectes ?
- **Les volumes et les performances.** Les plus grandes bases de données comportaient en 1969 500.000 enregistrements [Bachman, 1969]. Elles en contiennent aujourd'hui plusieurs centaines de milliards, ont un taux de croissance journalier de plusieurs téra-octets et elles sont réparties sur des milliers de serveurs (Google et Facebook par exemple). Comment mémoriser, gérer et exploiter ces données, en toute sécurité et en offrant aux utilisateurs des temps de réponse acceptables ?
- **Accès aisé par des non-informaticiens.** L'explosion des demandes exige que l'utilisateur lui-même puisse accéder aux données, sans passer par l'intermédiaire d'un informaticien. Quelles technologies, quelles interfaces et quelles compétences vont amener ces utilisateurs à l'autonomie nécessaire ?
- **Maintenance et évolution.** Comme tous les composants des systèmes d'information, les bases de données évoluent, au gré de l'évolution des besoins des utilisateurs. Ce processus est complexe. En effet, une base de données est utilisée par plusieurs milliers de programmes, de sorte que toute modification de la structure des données exigera la modification de milliers de programmes et d'écrans d'interface avec les utilisateurs. Le manque de documentation décrivant les formats et la signification des structures de données constitue ici un défi majeur.
- **Les données distribuées et nomades.** Le modèle d'une base de données sous la forme d'un système unique localisé sur une machine n'est plus approprié aux exigences actuelles. Les données sont aujourd'hui distribuées sur plusieurs machines communicantes, qui vont des gros serveurs aux *smartphones* et tablettes. Les techniques de gestion et d'exploitation centralisées ne sont plus

suffisantes, d'autant que certaines parties de la base de données sont nomades, intermittentes, dupliquées et partiellement autonomes.

- **Les bases de données et le web.** Le web, on l'a assez dit, constitue une extraordinaire base de données. On est loin cependant des concepts classiques des bases de données qui nous sont familiers : mode de stockage, formats de données, SGBD, délocalisation anonyme du stockage et des traitements (*cloud computing*), langages de consultation et de gestion, intégrité, sécurité, résistance aux pannes, tout est fondamentalement différent. Comment unifier web et bases de données ? Comment interfacer le web avec les bases de données ?
- **Les données décisionnelles.** Aux bases de données traditionnelles, celles que nous allons étudier dans cet ouvrage, s'ajoutent les bases de données d'aide à la décision, souvent dénommées *entrepôts de données*. Ces bases de données, spécialement conçues pour l'analyse de données, sont souvent de très grande taille (d'où le terme de *BigData*), sont alimentées à partir de sources diverses, dont les bases de données classiques, et font l'objet de traitements statistiques extrêmement lourds.
- **Les O/RM.** L'émergence des outils de développement de programmes dits *Object/Relational Mapping* (tels EJB et Hibernate) incitent naturellement à modéliser les besoins des utilisateurs sous la forme de classes d'objets. De celles-ci, les outils sont capables de dériver de manière automatique les structures d'une base de données permettant de stocker les données persistantes des objets de ces classes. Cette pratique entraîne souvent de multiples problèmes de qualité et d'évolutivité de la base de données².

Nous ne pourrions bien sûr pas, dans un ouvrage introductif tel que celui-ci, aborder tous ces thèmes de manière détaillée. Nous voudrions cependant présenter de manière pragmatique mais rigoureuse les concepts et les techniques qui permettent au lecteur de comprendre les technologies et les défis des bases de données d'aujourd'hui et ainsi de les maîtriser. À lui ensuite de prolonger son étude sur la base les nombreuses ressources de la littérature et du web.

1.7 UN PEU D'HISTOIRE

Nous brosserons un bref historique des systèmes de bases de données dans lequel on situera la naissance et l'évolution des bases de données relationnelles. Nous décrirons leur positionnement par rapport aux autres modèles en vigueur aux différentes époques, depuis IMS jusqu'aux modèles NoSQL. On précisera l'utilité et les apports

2. On assiste en fait à une inversion de paradigme : l'idée d'une base de données stable au service d'un ensemble évolutif de programmes est remplacée par celle d'une base de données spécifique aux besoins d'une application et évoluant avec celle-ci. C'est cette dernière situation, dont les inconvénients ont été largement identifiés et analysés, qui a donné naissance il y a 40 ans au concept, alors novateur, de base de données.

de l'approche relationnelle dans un contexte plus large que celui des bases de données relationnelles au sens strict.

Le modèle relationnel a été publié pour la première fois en 1970 [Codd, 1970] dans un contexte où la multiplication des logiciels de gestion de données entraînait une égale multiplication des modèles de structure de données, parfois dédiés à un type d'application spécifique³. La nécessité d'une standardisation apparaissait de manière critique. Deux tenants du titre s'opposaient : les SGBD CODASYL et le système IMS d'IBM.

1.7.1 Le modèle collectiviste : CODASYL DBTG

Le groupe CODASYL (*Conference on Data System Languages*) créé en 1959, regroupait des utilisateurs et des fournisseurs de logiciels, dont IBM, et avait dès 1960 développé la recommandation COBOL. Le sous-groupe DBTG (*Database Task Group*) de CODASYL émettait en 1968 un projet de norme concernant la définition et la manipulation de données à structure complexe, norme qui sera publiée sous forme de recommandation en 1971.

Basée sur le SGBD IDS d'Honeywell-Bull (1963), développé par Ch. Bachman, cette proposition incluait un modèle de données sous la forme d'un graphe dont les nœuds sont des types d'enregistrements (dont les champs ont une structure arborescente, à la COBOL) et les arcs des types d'associations un-à-plusieurs. Le schéma d'une base de données était défini via le langage Schema-DDL⁴ et les sous-schémas (on parlerait aujourd'hui de *vues*) via un Subschema-DDL spécifique au langage d'application (COBOL, Fortran ou PL/1 par exemple). La manipulation des données elles-mêmes s'effectuait par le langage DML⁵, qui était constitué de primitives de navigation et de modification des données. De nombreux SGBD s'inspirant directement de cette recommandation ont été développés. Certains sont encore largement utilisés aujourd'hui. On citera par exemple IDS2 de Bull, IDMS de Culliname (racheté par Computer Associates), DBMS de Digital, actuellement chez Oracle et UDS de Siemens. On concède aux SGBD CODASYL d'excellentes performances, une grande sobriété en ressources machine et une programmation simple, mais leur rigidité (difficulté de faire évoluer un schéma), l'absence de langage prédicatif natif tel que SQL (toute nouvelle requête doit se traduire par un nouveau programme) ont souvent poussé leurs utilisateurs à les remplacer par un SGBD relationnel.

Ils ont cependant subi de multiples évolutions telles que l'adjonction d'une couche relationnelle qui permet d'accéder à leurs données via une interface SQL (moteur CODASYL et interface SQL). On trouvera dans [Hainaut, 2003] et [Hainaut, 2009a], disponibles sur le site de l'ouvrage, une introduction aux SGBD CODASYL accompagnée d'une bibliographie.

3. Tel que DBOM d'IBM, dédié à la gestion de nomenclatures de produits (**B**ill of **M**aterial). Cette problématique est toujours d'actualité. Nous y reviendrons en particulier à la section 8.7.

4. *Data Description Language*

5. *Data Manipulation Language*

1.7.2 La pensée unique : IMS

À l'origine, IMS⁶ était un logiciel de gestion de données de structure arborescente sérialisées sur bande magnétique développé dès 1965 par IBM pour North American Rockwell, dans le cadre du projet de vol humain vers la lune. IMS a ensuite été commercialisé comme produit indépendant dès 1969 sous le nom complet de IMS/360. Il a été rapidement étendu par des fonctions liées à l'implantation des données sur disque magnétique, et en particulier l'accès direct aux données et la possibilité de modification aléatoire (non séquentielle). Système propriétaire, irrégulier et complexe, mais extrêmement puissant par ses fonctionnalités, sa robustesse et ses performances, IMS a subi diverses évolutions (*logical child*, *secondary index*) qui n'ont fait que compliquer le modèle et son utilisation.

Les données se présentent sous la forme d'arbres, qu'on parcourt de la racine vers les feuilles. Des structures additionnelles, au demeurant obéissant à des règles complexes et peu intuitives, permettent de naviguer d'arbre en arbre⁷. D'aucuns ont fait remarquer, non sans raison, que les structures sous-jacentes au langage XML présentaient d'étranges analogies avec le modèle IMS.

Jadis vedette des bases de données sur les *mainframes* (ordinateurs centraux) IBM, IMS a progressivement, tout au long des années quatre-vingt, été concurrencé par DB2, le gros moteur relationnel que promouvait avec prudence le même constructeur. IMS est encore utilisé, en particulier pour gérer de très grosses bases de données interactives à haute performance et haute fiabilité, notamment dans le monde bancaire, mais il est en perte de vitesse. À sa complexité, son manque de souplesse et sa lourdeur, s'ajoute la difficulté de trouver aujourd'hui des informaticiens désireux de se former à cette technologie perçue comme obsolète⁸.

Il est intéressant, mais peu surprenant, de constater qu'IBM, pourtant membre de CODASYL, n'ait jamais développé de SGBD selon cette dernière norme⁹. On trouvera dans [Hainaut, 2009b] une brève présentation d'IMS ainsi qu'une bibliographie. On consultera bien sûr aussi le site <http://www.ibm.com/ims>.

1.7.3 La naissance du modèle relationnel

Celle-ci a été particulièrement discrète. Paru dans les CACM, revue scientifique de bon niveau à cette époque, l'article de E. F. Codd¹⁰ [Codd, 1970] a laissé le monde

6. *Information Management System*. Souvent dénommé DL/1, du nom de son langage DDL/DML.

7. Dans le monde des bases de données, on *navigue* en effet dans les arbres. Dès lors qu'on trouve naturel que les arbres aient la racine en haut et les feuilles en bas, plus rien ne doit étonner.

8. Les SGBD CODASYL, plus simples et plus réguliers, exigent beaucoup moins d'efforts de formation.

9. En toute rigueur, il ne s'agit pas d'une véritable norme, puisqu'aucun organisme officiel de normalisation (comme l'ISO au niveau international ou l'ANSI, nord-américaine) n'a pris ces recommandations en charge. Une tentative a existé dans les années quatre-vingts, sous le nom NDL (*Network Data Language*), mais la vague SQL l'a rejetée aux oubliettes. Seule a subsisté la norme RDL (*Relational Data Language*), qui, à la fin de la décennie, a fait l'objet d'un coup de force de la part d'IBM, sifflant la fin de la récréation et imposant sa propre norme dérivée du SGBD SQL/DS. De là sont nées les normes ANSI SQL/87 et SQL/89.

industriel parfaitement indifférent. Même si l'auteur suggérait qu'un tel modèle pourrait conduire à terme à des réalisations concrètes, sa proposition apparaissait comme un essai purement théorique qui ne menaçait en rien la suprématie des *majors* CODASYL et IMS, ni la multiplication frénétique de logiciels de gestion de bases de données propriétaires. L'idée de structurer des données exclusivement sous la forme de domaines (ensembles de valeurs) et de relations (sorte de tables), et de les manipuler via des opérateurs algébriques ou des langages prédicatifs, sans aucun souci de performance, n'a pas réellement convaincu l'industrie. Les chercheurs ont donc travaillé sur les fondements théoriques de ces concepts avant d'envisager leurs applications pratiques. Ce n'est que quelques années plus tard que sont apparues les premières offres commerciales, notamment Oracle en 1979.

1.7.4 Apparition d'un nouvel acteur : l'utilisateur final

Les objectifs de Codd sont clairement énoncés : il propose un modèle simple, rigoureux et complet. *Simple*, en ce qu'il ne comporte que deux constructions élémentaires : le domaine et la relation, et un petit nombre d'opérateurs de dérivation, par opposition aux modèles dominants, qui se composaient de dizaines de concepts irréguliers et hétéroclites et d'un langage de manipulation souvent complexe. *Rigoureux* puisque, simple variante de la théorie des ensembles et des relations, il relève des mathématiques discrètes. *Complet* puisqu'on a pu montrer, d'une part, que ces structures étaient suffisantes pour représenter tout ce que les modèles concurrents pouvaient offrir, et que d'autre part, les opérateurs algébriques formaient un système de dérivation équivalent à la logique du premier ordre. On pouvait donc aisément *tout décrire* et *tout manipuler* via ce simple modèle, qui de surcroît, jouissait d'une interprétation mathématique rigoureuse.

En outre, et grâce à ces qualités, ce modèle strictement indépendant de toute technologie, était accessible tant au développeur qu'à l'utilisateur final. Les SGBD de l'époque n'offraient pas d'interface pour les utilisateurs, lesquels s'en remettaient aux informaticiens pour satisfaire leurs demandes en information, que ces derniers traduisaient, dès que leur planning le leur permettait, en programmes d'application.

1.7.5 La guerre des modèles

Ainsi donc, un nouveau souci apparaissait dans ce capharnaüm de modèles, celui de représenter de manière *naturelle* les informations complexes contenues dans les bases de données. Alors même qu'apparaissaient les premiers balbutiements du modèle relationnel, des groupes de travail s'attelaient au nettoyage des modèles CODASYL et IMS pour en dégager des structures plus simples, indépendantes des techniques d'implémentation. Assez rapidement, le modèle relationnel s'est érigé en

10. Si Codd n'est pas à proprement parler l'(unique) *inventeur* du modèle relationnel (par exemple, une "*information algebra*" avait été publiée en 1962 par le groupe CODASYL [Bosack, 1962]), il a le mérite de l'avoir consolidé, mis en forme et publié de manière convaincante. N'a-t-on pas dit qu'Einstein n'avait pas inventé les théories de la relativité, mais que son génie avait été de rassembler, de synthétiser et de diffuser des matériaux existants mais éparés ?

concurrent. S'est alors déroulé durant quelques années (1972-1976) un débat particulièrement animé durant lequel chaque modèle était proposé, sur base d'arguments plus irréfutables les uns que les autres¹¹, comme la meilleure manière de structurer l'information avant même de l'implémenter sous la forme d'une base de données. L'idée d'un niveau de modélisation abstrait, qu'on qualifiera de *conceptuel* dès 1974 [Deheneffe, 1974], mais surtout en 1976 [Chen, 1976], était née. Le handicap des modèles opérationnels propres aux SGBD était précisément que, dérivant de systèmes implémentés depuis plusieurs années, il souffraient d'irrégularités et de limitations que les tentatives de nettoyage n'avaient pas réussi à faire disparaître¹². Dans cette course, le modèle relationnel, largement poussé par la communauté scientifique, apparaissait comme le favori.

1.7.6 La chute d'Icare

En fait, aucun des prétendants ne remporta la victoire. Ils furent tous supplantés par un outsider inattendu, le modèle Entité-association. Héritier de divers courants de réflexion sur la sémantique des données, au travers de modèles théoriques, mais aussi d'une inspiration plus pragmatique issue des modèles de certains SGBD¹³, ce modèle proposait de modéliser l'univers du discours au moyen de mécanismes de classification et d'agrégation. Il structurait le monde en ensembles d'entités, dotées de propriétés et en association les unes avec les autres. Le modèle relationnel, qui fondamentalement ne propose que le mécanisme d'*agrégat de valeurs*, apparaissait trop pauvre pour constituer un mode d'expression capable de satisfaire les besoins tant des analystes que des utilisateurs non-techniciens. L'article de Chen [Chen 1976] faisait cependant appel au modèle relationnel, mais essentiellement pour préciser la sémantique des constructions du modèle Entité-association, idée que nous reprendrons au chapitre 16.

Il y eut encore une dernière tentative en 1979 [Codd, 1979] consistant à enrichir le modèle relationnel de concepts tels que la représentation explicite d'entités qui ne fait pas appel à des attributs significatifs (mais à des identifiants d'objet appelés *surrogates*) et les relations de généralisation. Cette proposition, par ailleurs relativement complexe, arriva trop tard et n'eut guère de suite.

1.7.7 Le modèle relationnel : la suite de l'histoire

S'il n'a pas pu s'imposer comme formalisme conceptuel universel, le modèle relationnel a cependant eu un impact considérable sur les outils et les pratiques relatifs aux bases de données et à leur ingénierie. Nous pouvons identifier trois domaines d'intense activité qui ont livré des résultats d'une importance capitale.

11. . . . et réciproquement.

12. Deux exemples : (1) en CODASYL, un type d'associations ne peut être cyclique ni plusieurs-à-plusieurs, (2) en IMS, les données se présentent systématiquement sous la forme d'arbres.

13. Il est indubitable que le modèle CODASYL, mais aussi celui du SGBD Socrate (développé par l'université de Grenoble pour la CII, puis commercialisé par Bull et SYSECA) ont influencé les chercheurs à l'origine du modèle Entité-association.

- **Approfondissement du modèle.** Comme nous le verrons au chapitre 3, les travaux de recherche ont mis en lumière des structures particulières des données telles que les diverses sortes de dépendances et les formes de normalisation qui en découlent. Le modèle relationnel a été étendu pour couvrir des domaines importants tels que les BD déductives, les BD actives, les données spatiales et temporelles, les données textuelles et multimédia et les objets. Il a aussi servi de base à l'étude des BD distribuées et de l'interopérabilité. Il a permis d'étudier avec une certaine rigueur la question de l'information incomplète et imprécise.
- **Langages.** Les différents langages et opérateurs développés dans le cadre de la théorie relationnelle ont fait l'objet de réalisations concrètes : langages algébriques (PRTV d'IBM, RDO de DIGITAL), calcul relationnel (QUEL), hybrides (SEQUEL, puis SQL), graphiques (QBE, partiellement repris notamment par Access de Microsoft).
- **SGBD.** Si les premières propositions étaient de nature théorique, il n'a pas fallu attendre longtemps pour que des prototypes de systèmes de gestion apparaissent, du moins dans les laboratoires. Les premières réalisations significatives furent celles d'IBM (System/R) et de l'université de Berkeley (Ingres)¹⁴. Le premier proposait un langage hybride inspiré à la fois de l'algèbre et du calcul (SEQUEL), tandis que le second offrait le langage QUEL, proche du calcul relationnel. Malgré la supériorité du second en régularité et en simplicité, c'est le premier qui s'est imposé sous le nom de SQL. Ces deux systèmes ont été commercialisés dans les années quatre-vingts. Le premier SGBD relationnel commercial est cependant sorti plus tôt, en 1979 sous le nom d'Oracle. IBM a ensuite proposé son premier système en 1982 sous le nom de SQL/DS, suivi de peu de DB2, lui-même constitué d'une version simplifiée du System/R. Depuis cette date, plus d'une centaine de SGBD relationnels ont été commercialisés, dont quelques dizaines subsistent encore aujourd'hui. Tous les SGBD non relationnels qui sont encore en activité à l'heure actuelle (tels que les SGBD CODASYL) ont été complétés d'une interface SQL.

1.7.8 Cronos dévore ses enfants

Dès que la technologie relationnelle a commencé à s'implanter dans le monde industriel, de nouvelles approches ont été développées, ouvrant ce qu'on a appelé, pour chacune, une ère dite *post-relationnelle*. Nous évoquerons trois d'entre elles.

Les bases de données orientées objet

Imaginées dans les années quatre-vingts, les bases de données orientées objet avaient pour objectif de réduire le conflit de paradigmes, ou fondements conceptuels, entre les langages de programmation orientés objet et les bases de données relationnelles. La manipulation d'un objet dans un programme obéit à des règles différentes, selon qu'il s'agisse d'un objet interne au programme ou d'un objet (ou

14. Ancêtre de PostgreSQL, SGBD Open Source très populaire.

ligne d'une table) de la base de données. Par exemple, la création d'un objet par la méthode de classe `new` et l'insertion d'une ligne dans une table par la requête SQL `insert` appartiennent à des logiques très éloignées l'une de l'autre¹⁵. C'est au programme de gérer la persistance d'un objet via les services d'un SGBD classique. Les bases de données orientées objet avaient pour premier but d'assurer de manière transparente le stockage et la gestion des objets des programmes dont on désirait garantir la persistance. D'autres objectifs ont été aussi pris en compte : une modélisation plus riche des données (un objet comporte un arbre de valeurs et non une simple suite de valeurs atomiques), l'expression explicite de liens entre les objets (par association, par inclusion ou par référence) et des langages de requêtes en général inspirés de SQL. On en dira quelques mots au chapitre 10.

Malgré de nombreux projets de recherche et de développement, et l'apparition sur le marché de quelques SGBD orientés objet dans les années quatre-vingt-dix, force est de constater qu'il ne reste aujourd'hui que peu de traces de ces efforts. En fait, ce mouvement s'est dissout dans la technologie relationnelle, sous le nom de *relationnel objet*, qu'il était chargé de supplanter. La norme SQL:1999 (dite SQL3) a incorporé quelques constructions (section 9.6), s'inspirant des principes de l'orientation objet. Oracle par exemple les a adoptées dès la version 8.

Les bases de données XML

La seconde tentative, plus récente, concerne les SGBD dits XML. Il est parfaitement possible de stocker des documents XML dans une base de données relationnelle, et ce, de multiples manières. Il est cependant tentant d'imaginer un SGBD dont le modèle de données natif serait celui d'XML, dont le langage de définition serait tout simplement celui des *DTD* ou des *XML Schemas*, et offrant un langage de manipulation opérant directement et explicitement sur des structures XML sans qu'il soit nécessaire de passer par une conversion en SQL¹⁶. Plusieurs propositions commerciales de SGBD XML ont vu le jour. Il faut cependant observer que les SGBD relationnels ont offert depuis lors des interfaces XML destinées à capter le marché des utilisateurs exigeants de données XML complexes, avec pour résultat d'étouffer les SGBD XML natifs. On évoquera aussi les SGBD XML natifs au chapitre 10.

Les bases de données NoSQL

La prise en compte de données massives, largement distribuées et répliquées, a mis en lumière les limites des SGBD relationnels, qui privilégient la cohérence et la protection des données au détriment de leur accessibilité. Les SGBD dits NoSQL, nés dans les années 2000, ont pour vocation de combler ces lacunes. Ils seront étudiés au chapitre 10. On observe, une fois de plus, que les SGBD relationnels classiques intègrent progressivement les fonctions des SGBD NoSQL.

Ainsi, tel Cronos, la technologie relationnelle dévore-t-elle ses enfants au fur et à mesure qu'ils menacent ses parts de marché !

15. Ce qu'on a appelé *impedance mismatch* (http://en.wikipedia.org/wiki/Impedance_mismatch).

16. Il est intéressant de noter que les modèles orientés objet des années quatre-vingt avaient souvent comme domaine de prédilection la gestion et la manipulation de documents complexes.

Chapitre 2

Concepts des bases de données

Ce chapitre décrit les premières notions des bases de données. Les données se présentent sous la forme de tables formées de lignes et de colonnes. Une 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 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 (clés étrangères). Le comportement du SGBD lors des opérations de modification des données est analysé. On étudie aussi le phénomène de redondance interne, qui conduira au concept de table normalisée.

2.1 INTRODUCTION

Dans ce chapitre, nous allons aborder plus en détail les structures de données que nous avons découvertes informellement dans la section 1.2. La représentation des données sous forme de tableaux est simple et intuitive. On peut cependant imaginer qu'une base de données de plusieurs centaines, voire plusieurs milliers de tables n'ira pas sans quelques problèmes qui forceront à respecter une certaine discipline. D'une part, les SGBD proposent d'associer aux tables des propriétés qui vont nous permettre de structurer plus finement les données. D'autre part, nous construirons nos bases de données de manière systématique, comme nous le verrons dans la troisième partie de l'ouvrage. L'objectif de ce chapitre est de décrire de manière détaillée les caractéristiques des structures de données offertes par les SGBD relationnels ainsi que leurs conséquences dans l'extraction et la modification des données.

2.2 TABLES, LIGNES ET COLONNES

Une base de données est organisée 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 **valeurs**, chacune 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 appellerons parfois une **entité** ou un **fait**.

La figure 2.1 représente une table comportant huit lignes¹, décrivant chacune un client. On trouve dans chaque ligne 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 1 700.

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 chacune des colonnes NOM, ADRESSE, LOCALITE et COMPTE.

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 2.1 - Structure et contenu de la table CLIENT

L'ensemble des valeurs de même type correspondant à une même propriété des entités constitue donc une **colonne**. '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). Préciser la nature des valeurs d'une colonne consiste à spécifier son **type** ou son **domaine de valeurs**. Nous n'en dirons pas plus à ce stade.

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-

1. Leur ordre est indifférent: on ne peut donc en principe parler de la première, de la troisième ou de la dernière ligne d'une table. 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.

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 2.2 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 table, nommée PIECE, décrit des pièces de rechange, caractérisées par leur numéro (NUMP) et leur type (TYPE). La troisième table, 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 2.2 - Un exemple de base de données

2.3 LA VALEUR null

Affirmer que chaque ligne possède une valeur pour chaque colonne de la table implique que cette valeur existe et soit connue au moment d'enregistrer cette ligne dans la table. Tel n'est pas toujours le cas. Des circonstances particulières peuvent faire qu'il faille procéder à cet enregistrement alors que certaines données ne sont pas connues. Citons-en trois :

- l'information est pertinente mais n'existe pas pour l'entité (le fait) correspondant à la ligne (numéro minéralogie du véhicule d'une personne qui ne possède pas de voiture),
- l'information n'est pas pertinente pour les entités telles que celle qui correspond à la ligne (le code NACE n'existe pas pour un client qui est une personne physique),
- l'information existe mais n'est pas connue à ce moment (date de naissance d'une personne).

L'absence de valeur d'une colonne dans une ligne, quelle qu'en soit la raison, se signalera par l'affectation de la valeur conventionnelle **null** à cette colonne. Ainsi, si l'offre du fournisseur 81 (Dumont) pour le type de pièces 57 (ECROU) existe mais que son prix n'est pas encore connu, on inscrira la ligne (81, 57, null) dans la table OFFRE (figure 2.2). Nous verrons à plusieurs occasions que les valeurs null sont

déliçates à manipuler et qu'il est préférable de les éviter lorsque la chose est possible. La section 9.12 étudiera plus en détail la question de l'*information incomplète* et la manière dont le langage SQL la traite (ou la maltraite !) En particulier, on y verra que null n'est pas une valeur à proprement parler, mais un *marqueur* indiquant l'absence de valeur.

On peut imposer l'interdiction d'assigner la valeur null à une colonne. Une telle colonne sera appelée **colonne obligatoire**. Si la valeur null est autorisée, cette colonne sera dite **facultative**. Le SGBD garantit le caractère obligatoire d'une colonne. Toute tentative d'insérer une ligne qui ne posséderait pas de valeur pour une colonne obligatoire serait automatiquement rejetée.

2.4 LES IDENTIFIANTS

Nous avons vu que 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.

L'une des fonctions essentielles qui seront associées à une colonne est la désignation univoque de l'entité représentée, c'est-à-dire son identification. Une colonne constitue un **identifiant**² de sa table si, à tout instant, il ne peut exister plus d'une ligne possédant une valeur déterminée de cette colonne. Tel est le cas de NUMF pour la table FOURNISSEUR et NUMP pour la table PIECE. Déclarer que la colonne NUMF forme l'identifiant de la table FOURNISSEUR, c'est imposer qu'à tout instant les lignes de cette table aient des valeurs distinctes de NUMF.

Cette définition est valable également pour un *groupe de colonnes* (on parle d'**identifiant composite**). Si un fournisseur ne peut faire qu'une seule offre par type de pièces, alors les colonnes {NUMF, NUMP} constituent un identifiant composite de la table OFFRE.

On peut définir **plusieurs identifiants** dans une table. Par exemple, une table qui reprendrait 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.

Identifiant primaire et identifiant secondaire

Parmi les identifiants d'une table, l'un est choisi comme le plus représentatif et sera dès lors déclaré **identifiant primaire**³. Les autres sont alors des identifiants **secondaires**⁴ de la table. En règle générale, toute table possède un identifiant primaire et zéro, un ou plusieurs identifiants secondaires⁵. Les lignes d'une table qui possède au moins un identifiant forment un *ensemble* et sont donc distinctes.

2. La terminologie standard des bases de données relationnelles propose le terme de *clé* ou *key* pour désigner ce concept. Étant donné le grand nombre de significations différentes assignées à ce terme dans le domaine des bases de données, nous lui préférons celui d'identifiant. Nous en reparlerons dans la section 3.3.

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

4. Le terme *secondaire* n'est pas standard.

Identifiant et contrainte d'unicité

Pour jouer correctement ce rôle d'identification, il est donc nécessaire que l'identifiant possède des valeurs distinctes, propriété qu'on appelle **contrainte d'unicité**. Celle-ci est garantie par le SGBD, qui rejettera automatiquement toute tentative d'insertion d'une ligne dont la valeur de l'identifiant est déjà présente dans la table.

Composition de l'identifiant primaire

Un identifiant primaire doit être constitué de colonnes obligatoires. C'est ce qu'on appelle l'**intégrité d'entité** (*entity integrity*). L'idée est que l'essence même d'une entité se concentre dans l'identifiant primaire de sa table.

Notion d'identifiant minimal

Tout ensemble de colonnes qui comprend un identifiant est encore un identifiant : (NUMF, NOMF) est aussi 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*. Sauf circonstances particulières, on cherchera à ne définir que des identifiants minimaux.

2.5 LES CLÉS ÉTRANGÈRES

Un examen attentif de la figure 2.2 montre que les deux colonnes NUMFL et NUMPL jouent dans la table OFFRE un rôle de *référence*. En effet, chacune de leurs valeurs référence une ligne d'une autre table. La valeur 152 de la colonne NUMFL de la ligne (152, 14, 62) de la table OFFRE désigne la ligne (152, MERCIER, Tours) de la table FOURNISSEUR.

Une colonne, ou un ensemble de colonnes, d'une table (dite *table source*) dont le rôle est de référencer une ligne dans une autre table (dite *table cible*) est dénommé **clé étrangère** (*foreign key*). Celle-ci est une copie d'un identifiant (dit *identifiant cible*) de la table cible. Elle est donc formée de colonnes en même nombre et de mêmes types que l'identifiant cible. Une clé étrangère dont l'identifiant cible est composite est elle-même composite.

C'est par les 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).

Le nom d'une colonne d'une clé étrangère est indépendant de celui de l'identifiant qu'elle référence. On peut donner à une clé étrangère le nom de l'identifiant cible (NUMP dans OFFRE), ou celui de la table référencée (PIECE), ou encore celui du rôle que jouent les lignes référencées (PIECE_OFFERTE).

5. 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 identifiant, possibilité que nous ignorerons dans ce chapitre.

Clé étrangère et contrainte référentielle

Pour jouer correctement ce rôle de référence, il est nécessaire que l'ensemble des valeurs d'une clé étrangère soit un sous-ensemble des valeurs de l'identifiant cible, et ceci à tout instant. Cette propriété est appelée **contrainte référentielle**. Elle est garantie par le SGBD pour autant qu'on ait explicitement déclaré les clés étrangères. Toute opération qui conduirait à violer cette contrainte serait automatiquement rejetée. Par exemple, l'insertion de la ligne (174, 97, 125) dans la table OFFRE serait rejetée puisqu'il n'existe pas de ligne dans la table PIECE dont le numéro est 97.

Une dernière remarque

La cible d'une clé étrangère peut être tout identifiant, primaire ou secondaire, de la table référencée. On convient, si possible, de se limiter à l'identifiant primaire.

2.6 IDENTIFIANTS ET CLÉS ÉTRANGÈRES FACULTATIFS

La présence de composants facultatifs dans un identifiant ou une clé étrangère complique quelque peu leur interprétation.

a) Identifiant secondaire et colonnes facultatives

Un identifiant peut être constitué d'une colonne **facultative**. La contrainte d'unicité ne porte alors que sur l'ensemble des lignes dont la colonne possède une valeur non *null*, sauf pour certains SGBD (SQL Server par exemple), qui n'admettent qu'une seule ligne possédant une valeur *null*, ce qui est une erreur très gênante.

Si l'identifiant est **composite** et si toutes ses colonnes (soient A1 et A2) sont facultatives, alors ces colonnes sont en principe *coexistantes*, c'est-à-dire qu'à tout instant, pour toute ligne de la table, (1) soit A1 et A2 ont une valeur non *null*, (2) soit A1 et A2 ont une valeur *null*. La contrainte d'unicité ne porte que sur les lignes dont les colonnes de l'identifiant possèdent toutes une valeur non *null*.

Le cas suivant est plus délicat. L'identifiant est **composite** et au moins une de ses colonnes est facultative. Que devient une ligne dont l'identifiant ne respecte pas la règle de coexistence ci-dessus et est **en partie null** (A1 a une valeur et A2 est *null*). Il existe plusieurs interprétations de la contrainte d'unicité dans ce cas :

- La contrainte d'unicité ne s'applique pas à cette ligne (observé dans Oracle, MySQL, PostgreSQL, InterBase, Firebird et SQLite).
- La contrainte d'unicité est satisfaite s'il n'existe pas d'autre ligne ayant même valeur de A1 (Informix, SQL Server, DB2).

b) Clés étrangères et colonnes facultatives

Si une clé étrangère est constituée d'une colonne facultative, la contrainte référentielle ne porte que sur les valeurs non *null* de cette colonne

Si la clé étrangère est **composite** et si toutes ses colonnes (appelons-les B1 et B2) sont facultatives, alors ces colonnes sont en principe *coexistantes*, c'est-à-dire qu'à tout instant, pour toute ligne de la table, (1) soit B1 et B2 ont une valeur non *null*, (2)

soit B1 et B2 ont une valeur *null*. La contrainte référentielle ne porte que sur les lignes dont les colonnes de la clé étrangère possèdent une valeur non *null*.

L'existence de lignes pour lesquelles **certain**s composants d'une clé étrangère possèdent une valeur tandis que les autres ont une valeur *null* est cependant autorisée par certains SGBD mais leur interprétation est plus complexe (voir section 6.9).

2.7 SCHÉMA 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 2.2, nous obtenons le schéma (encore partiel à ce stade) de la figure 2.3.

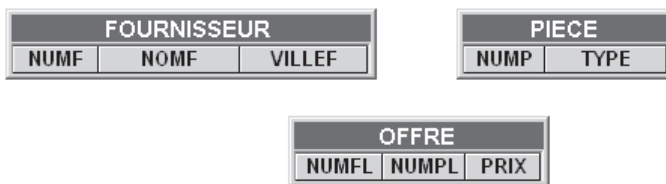


Figure 2.3 - Schéma (partiel) de la base de données de la figure 2.2

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 (de 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 du domaine d'application à représenter évolue.

2.8 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 2.4).

- Une *table* et ses *colonnes*⁶ 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.

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

- 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 (clé étrangère composite) sera représentée par une accolade.
- Une contrainte référentielle sera représentée par une flèche qui part du nom de la colonne de la clé étrangère (ou de l'accolade en cas de groupe de colonnes) et qui pointe vers le cartouche de la table référencée.

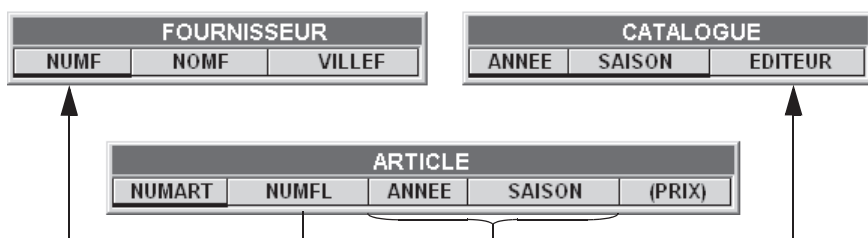


Figure 2.4 - Représentation des identifiants et des clés étrangères

Cette notation de représentation est simple et naturelle. Elle ne conviendra cependant pas pour dessiner des schémas plus volumineux et plus complexes tels que ceux que nous rencontrerons dans la suite de l'ouvrage.

On pourrait adopter une disposition telle que celle d'Access de Microsoft (figure 2.5), 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).

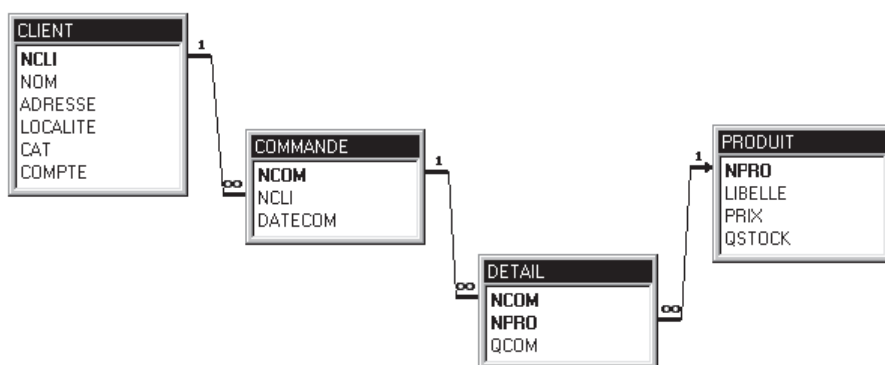


Figure 2.5 - Représentation graphique des schémas selon Access

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.

La notation d'Access ne permet cependant pas de représenter de manière graphique certaines constructions qui nous seront utiles⁷, ce qui nous amènera à

utiliser, lorsque cela sera nécessaire, la notation plus complète illustrée dans la figure 2.6, et qui est propre à l'atelier d'ingénierie DB-MAIN dont nous reparlerons plus loin. Les conventions de cette notation sont les suivantes :

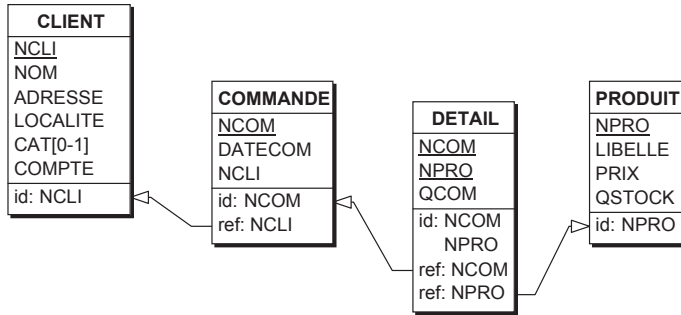


Figure 2.6 - Représentation graphique plus complète (atelier DB-MAIN)

- 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 ;
- une colonne facultative est caractérisée par le symbole [0-1] qui indique qu'une ligne possède de 0 à 1 valeur pour cette colonne⁸ ;
- 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 pour l'instant) ;
- un groupe de colonnes qui forme à la fois un identifiant et une clé étrangère est simultanément noté "id:" (ou "id:") et "ref".

2.9 UN EXEMPLE DE BASE DE DONNÉES

La base de données qui nous servira d'exemple principal dans cet ouvrage est structurée selon les schéma des figures 2.5, 2.6 et 2.7. Elle dérive du petit exercice illustré à la figure 1.5. Son contenu pourrait, à un instant donné, se présenter comme indiqué à la figure 2.8. On donnera une brève définition de la signification des tables et des colonnes.

- Table **CLIENT** : chaque ligne décrit un client ; les colonnes décrivent successivement le numéro du client (NCLI), son nom (NOM), son adresse (ADRESSE),

7. Les colonnes facultatives, les identifiants secondaires, les clés étrangères cycliques, les clés étrangères multiples entre deux mêmes tables, des identifiants et/ou clés étrangères non disjoints, les contraintes d'existence, les structures objet-relationnelles, etc.

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

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.

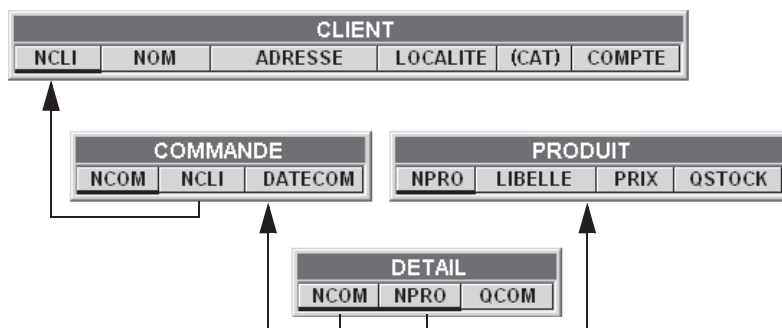


Figure 2.7 - Schéma de la base de données exemple

- Table **PRODUIT** : chaque ligne décrit un produit en précisant son numéro (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 ; on précise 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 en précisant le numéro de la commande de ce détail (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.

2.10 MODIFICATION ET CONTRAINTES D'INTÉGRITÉ

Les constructions du schéma, et notamment les colonnes obligatoires, les identifiants et les clés étrangères, imposent aux données des contraintes qui doivent être satisfaites à tout instant. Ces contraintes vont donc agir lors de toute tentative de modification des 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 contraintes seront toujours respectées par les données après ces opérations. Si ces contraintes sont violées, on dira que les données ont perdu leur intégrité. D'où le terme de **contraintes d'intégrité** sous lequel on les désigne généralement. Nous examinerons brièvement l'impact de ces contraintes sur les opérations de modification du contenu d'une base de données.

2.10.1 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, à l'exclusion de la valeur null. Par exemple, il est permis de créer une ligne de CLIENT sans valeur pour CAT, mais pas sans valeur pour LOCALITE.

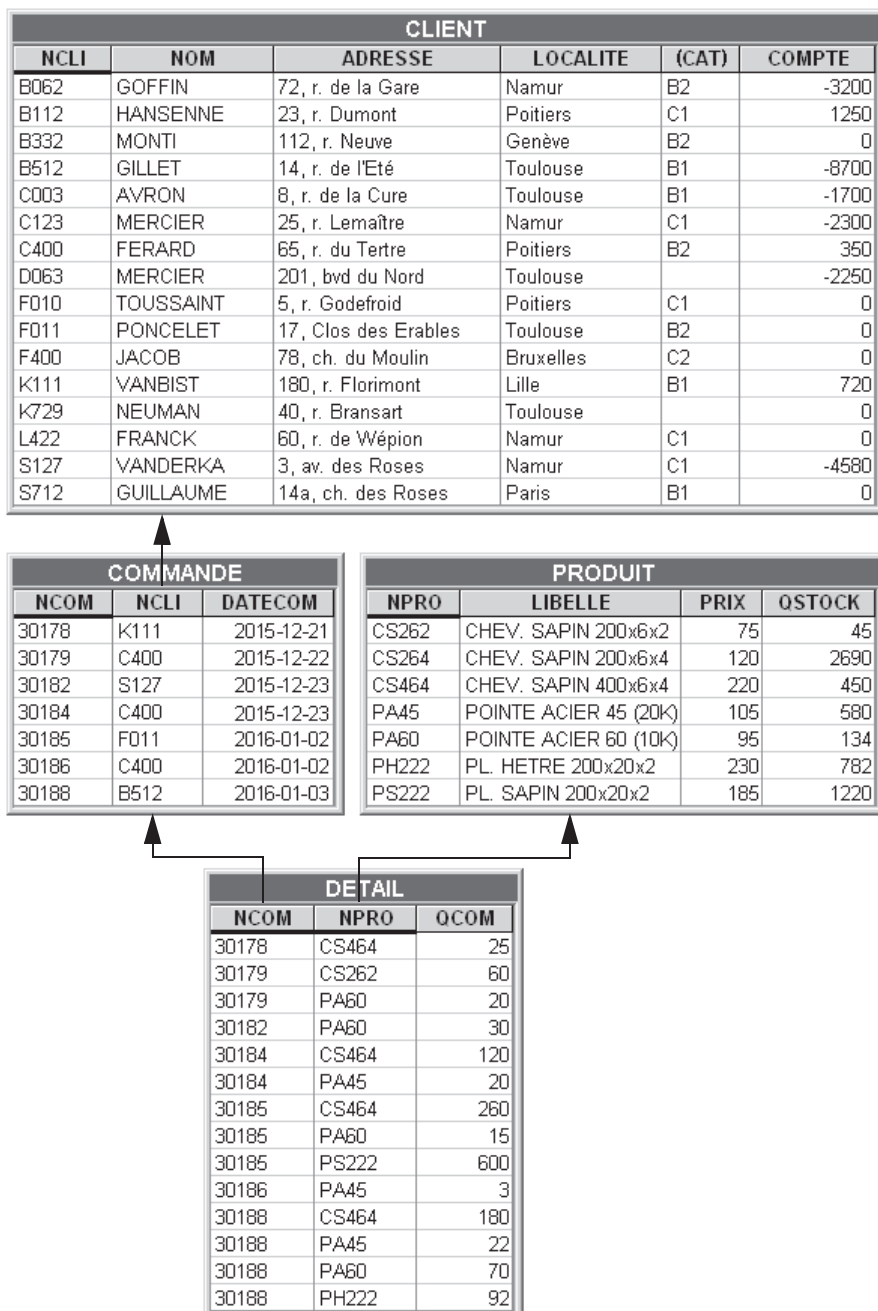


Figure 2.8 - Contenu typique de la base de données d'exemple

2.10.2 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 par exemple exister plus d'une ligne de **CLIENT** ayant la même valeur de **NCLI**.

- *Création d'une ligne* : autorisée s'il n'existe pas de ligne possédant la même valeur de l'identifiant.
- *Suppression d'une ligne* : pas de contrainte.
- *Modification de l'identifiant d'une ligne* : autorisée s'il n'existe pas de ligne possédant la nouvelle valeur de l'identifiant.

2.10.3 Les contraintes référentielles (clés étrangères)

Une contrainte référentielle précise que les colonnes d'une table qui constituent une clé étrangère doivent à tout instant, pour chaque ligne, contenir des valeurs qu'on retrouve comme identifiant primaire d'une ligne dans une autre table⁹. 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 2.9), 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*).

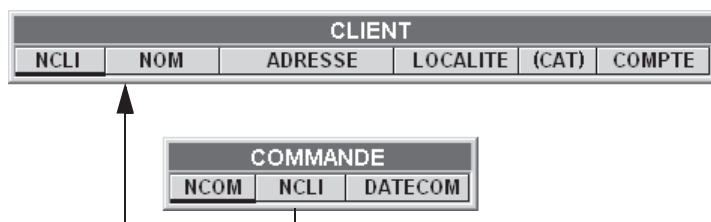


Figure 2.9 - Schéma d'étude de l'intégrité référentielle

*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 null).

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

Effectuée sans restriction¹⁰.

9. Il peut d'ailleurs s'agir de la même table. On dira alors que la clé étrangère est cyclique.

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 mettre la valeur de NCLI à null).

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. Il existe plusieurs comportements possibles qui laissent la base de données dans un état correct. On cite les trois principaux (d'autres seront décrits au chapitre 6) :

- **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*. Ce mode sera appelé en SQL **no action**.
- **Propagation** ou **cascade.** Le deuxième comportement consiste à supprimer non seulement la ligne de CLIENT, mais aussi toutes les lignes de COMMANDE qui la référencent¹¹. Ce mode sera appelé en SQL **cascade**.
- **Indépendance.** Le troisième comportement est possible lorsque la clé étrangère est constituée de colonnes facultatives. Il consisterait ici, si NCLI de COMMANDE avait été déclarée facultative, à mettre à null 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. Ce mode sera appelé en SQL **set null**.

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 (*blocage*, **no action**), modification des valeurs de clés étrangères qui référencent cette ligne (*propagation*, **cascade**) de manière à préserver les relations entre les lignes concernées, effacement des valeurs de clés étrangères qui référencent cette ligne (*indépendance*, **set null**).

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. Tous cependant, garantissent la cohérence des données.

10. 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.

11. L'effet réel de ces suppressions dépend du comportement défini pour la clé étrangère NCOM de DETAIL. On reviendra sur ces interférences au chapitre 9.

2.11 REDONDANCES INTERNES DANS UNE TABLE

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 2.10, 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	14/10/2008	F3
1030	L'humanité perdue	Finkelkraut A.	2 02 033300 7	14/10/2008	F3
1032	Mercure	Nothomb A.	2 253 14911 X	14/10/2008	G5
1045	Eva Luna	Allende I.	2 253 05354 6	22/2/2009	F3
1067	Mercure	Nothomb A.	2 253 14911 X	24/2/2009	G5
1022	Mercure	Nothomb A.	2 253 14911 X	3/10/2008	G6

Figure 2.10 - La table LIVRE enregistre les informations sur les livres disponibles dans une bibliothèque

2.11.1 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.

Cette représentation semble naturelle mais elle pose un problème qui apparaît clairement : 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, les données redondantes occupent un espace inutile.
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. Comment garantir que les valeurs en double seront orthographiées exactement de la même manière ?
4. Si l'enregistrement d'un premier exemplaire d'un livre peut se faire librement, celui des exemplaires suivants doit être conforme aux informations déjà présen-

tes, ce qui complique considérablement la procédure.

5. 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 que les valeurs de TITRE et d'AUTEUR *dépendent* directement, ou sont *fonction*, de celle d'ISBN. On a donné à cette contrainte le nom de **dépendance fonctionnelle**. Elle sera notée :

ISBN $\longrightarrow$ TITRE, AUTEUR

ISBN est le *déterminant* de la dépendance tandis que TITRE, AUTEUR en sont les *déterminés*. Il existe bien sûr, dans chaque table, une dépendance fonctionnelle entre l'identifiant de cette table et chacune des colonnes de celles-ci :

NUMERO $\longrightarrow$ TITRE, AUTEUR, ISBN, DATE_ACHAT, PLACE

L'origine de la redondance que nous avons observée est assez simple à identifier : les valeurs de TITRE et d'AUTEUR peuvent apparaître en plusieurs exemplaires parce que les valeurs de ces deux colonnes dépendent de celles d'une colonne (ISBN) **qui n'est pas un identifiant de la table**. Comme il est normal qu'une même valeur d'ISBN apparaisse plusieurs fois dans la table, il est tout aussi normal que certaines valeurs de TITRE et AUTEUR apparaissent également en plusieurs exemplaires, constituant ainsi une redondance.

On en tire une conclusion pratique : il faut éviter de structurer une table de telle sorte qu'elle soit le siège d'une dépendance fonctionnelle problématique, c'est-à-dire dont le déterminant n'est pas un identifiant de la table.

Reste une question à résoudre : que faire lorsqu'il existe dans une table une telle dépendance fonctionnelle ? C'est ce que nous allons examiner.

2.11.2 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 *Mercur* d'A. Nothomb (ISBN 2 253 14911 X) des trois exemplaires de celui-ci que constituent les livres 1032, 1067 et 1022. Les données spécifiques à un ouvrage sont communes à tous ses exemplaires et sont donc enregistrées autant de fois que cet ouvrage a d'exemplaires.

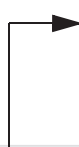
La résolution de ce problème passe par la **décomposition** de la table LIVRE en deux tables distinctes, auxquelles nous donnerons de nouveaux noms, pour éviter toute ambiguïté (figure 2.11). La première, OUVRAGE, contient la description des ouvrages. On y reprend le code ISBN, le titre et l'auteur de 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.

La description précise de cette décomposition sera étudiée au chapitre 3. Nous pouvons cependant tirer de notre petite expérience trois règles qui seront suffisantes pour l'instant :

1. La table OUVRAGE est constituée de toutes les colonnes impliquées dans la dépendance fonctionnelle problématique (ISBN, TITRE, AUTEUR) ; son *identifiant* est constitué du *déterminant* de cette dépendance fonctionnelle, ISBN.
2. La table EXEMPLAIRE est la table d'origine de laquelle on a retiré toutes les colonnes apparaissant en partie droite (*déterminé*) de la dépendance fonctionnelle problématique (TITRE, AUTEUR).
3. La colonne (ou les colonnes) du déterminant de la dépendance fonctionnelle problématique (ISBN) se retrouve(nt) dans les deux tables ; elle constitue dans EXEMPLAIRE une clé étrangère vers la nouvelle table OUVRAGE.

Ce processus de décomposition en vue d'éliminer les redondances internes porte le nom de *normalisation*. Les tables OUVRAGE et EXEMPLAIRE de la figure 2.11 sont dites *normalisées*, alors que la table LIVRE ne l'est pas.

Il est clair que la décomposition n'a pas entraîné de perte d'information. Il est en effet aisé de reconstituer le contenu exact de la table LIVRE à partir des contenus des tables OUVRAGE et EXEMPLAIRE. Ces deux bases de données ont donc le même contenu en information.



ISBN	TITRE	AUTEUR
2 02 033300 7	L'humanité perdue	Finkelkraut A.
2 253 14911 X	Mercure	Nothomb A.
2 253 05354 6	Eva Luna	Allende I.

NUMERO	ISBN	DATE_ACHAT	EMPL
1029	2 02 033300 7	14/10/2005	F3
1030	2 02 033300 7	14/10/2005	F3
1032	2 253 14911 X	14/10/2005	G5
1045	2 253 05354 6	22/2/2006	F3
1067	2 253 14911 X	24/2/2006	G5
1022	2 253 14911 X	3/10/2005	G6

Figure 2.11 - 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 2.10 sont ainsi éliminées

La seconde cependant a résolu les problèmes que posait la première. En effet, on observe que :

1. chaque information étant présente une seule fois, le volume des données est minimisé ;
2. la modification d'un titre ou d'un auteur ne réclame la modification que d'une seule ligne de la table OUVRAGE ;
3. l'introduction d'un exemplaire, qu'il s'agisse du premier ou d'un suivant, se traduit par la simple insertion d'une ligne, pour autant que l'intégrité référentielle soit préservée ;

- 4. un ouvrage peut être répertorié sans qu'il y corresponde d'exemplaire.

Le chapitre 3 sera consacré à une étude plus approfondie de la redondance interne ainsi que des techniques de résolution. À ce stade, il nous suffit d'observer ce phénomène pour en comprendre les causes.

2.12 LES STRUCTURES PHYSIQUES

À la forme tabulaire des données correspond sur le support (un disque magnétique généralement) 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 correcte est exécutable par le SGBD, qu'il existe ou non des index permettant d'accélérer l'accès et le tri des données.

- Ce terme *index* désigne un mécanisme dont l'objectif est comparable à celui de **l'index d'un ouvrage** tel que celui-ci. Pour un mot significatif donné, l'index nous donne la liste des numéros de page où ce mot apparaît, sans qu'il soit nécessaire de parcourir l'intégralité de l'ouvrage pour retrouver ces pages. En outre, l'index est ordonné de manière telle qu'on puisse rapidement y repérer le mot recherché. Dans le schéma de la figure 2.12, un index est représenté par un groupe de noms de colonnes préfixé du symbole **acc** (*access key* ou *clé d'accès*) 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, qui seront étudiées en détail au chapitre 4. 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.

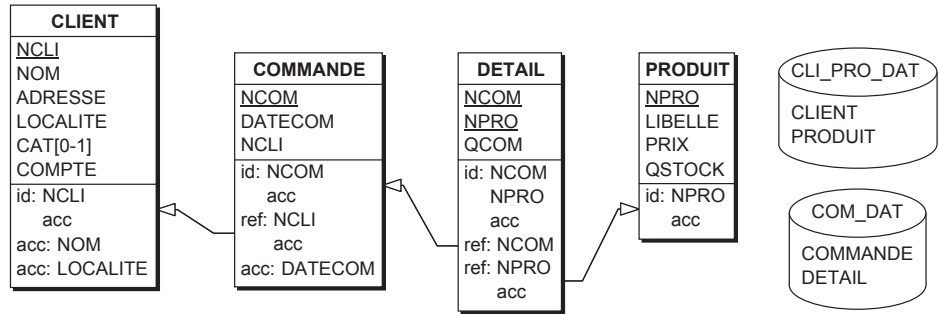


Figure 2.12 - Schéma physique d'une base de données spécifiant les index et les espaces de stockage

- Les lignes des tables sont rangées dans des fichiers, appelés plus spécifiquement **espaces de stockage** (*dbspace, tablespace, space*), qui constituent des *conteneurs* implantés dans une mémoire secondaire (disque magnétique, SSD, carte mémoire, CD-ROM, DVD). Un tel conteneur est découpé en *pages* de taille fixe (typiquement 4.096 octets) et peut accueillir des lignes de plusieurs tables. Dans un disque magnétique, le temps de lecture d'une page quelconque est de l'ordre de 12 millisecondes. Nous noterons graphiquement un espace de stockage par un cylindre comprenant la liste des noms des tables (figure 2.12).
- Un **agrégat physique** (souvent dénommé *cluster*) est formé d'un ensemble de lignes qui sont souvent extraites simultanément lors de l'exécution des requêtes. Les lignes formant un agrégat sont stockées 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.

2.13 QUE RETENIR ?

Une base de données est constituée d'une collection de **tables**. Dans une table, les données sont présentées sous la forme de lignes et de colonnes. Chaque **ligne** regroupe les données relatives à une **entité** ou un **fait** du **domaine d'application** (la partie du monde à laquelle on s'intéresse). Chaque **colonne** représente une propriété commune à ces entités ou à ces faits. Si, pour une ligne particulière, l'information sur une propriété est manquante ou inconnue, on marquera sa colonne de l'indicateur **null**, improprement appelé valeur null.

Une colonne acceptant les valeurs null est dite **facultative**. Dans le cas contraire, elle sera **obligatoire**. Un groupe d'une ou plusieurs colonnes tel qu'il n'existe jamais plus d'une ligne qui possède des valeurs déterminées de ces colonnes constitue un **identifiant** de la table. Parmi les identifiants d'une table, l'un est déclaré **identifiant primaire**, les autres étant secondaires. Un groupe d'une ou plusieurs colonnes d'une table tel que leurs valeurs pour chaque ligne réfèrent une ligne dans une autre table constitue une **clé étrangère** vers cette dernière table. L'identifiant impose une **contrainte d'unicité** et la clé étrangère une **contrainte référentielle**. Le caractère obligatoire des colonnes, les contraintes d'unicité et les contraintes référentielles constituent les principales **contraintes d'intégrité** d'une base de données.

Une base de données est définie par son **schéma**, qui spécifie la liste des tables, et pour chacune son nom, la liste de ses colonnes (nom, domaine de valeurs, caractère obligatoire), ses identifiants et ses clés étrangères. Quant aux données stockées dans les tables, elle constituent le **contenu** de la base de données.

La **modification** du contenu d'une base de données s'effectue par l'insertion de nouvelles lignes, la suppression de lignes existantes ou la modification de lignes existantes. Toute modification du contenu doit garantir la préservation des

contraintes d'intégrité associées à la base de données. L'intégrité référentielle requiert un soin particulier. Par exemple, la suppression d'une ligne référencée par d'autres lignes sera exécutée selon une des différentes politiques possibles (appelées *delete mode*) : **blocage** (suppression refusée), **cascade** (suppression accompagnée de celle des lignes référençantes) ou **indépendance** (suppression accompagnée par la mise à null des colonnes de la clé étrangère des lignes référençantes).

Une table peut inclure des **redondances internes**, par lesquelles certains faits peuvent être enregistrés plusieurs fois. Une telle duplication pose plusieurs problèmes, notamment lors de la mise à jour des données. Cette redondance apparaît lorsqu'on enregistre dans une table des données relatives à des faits de différentes natures. On cherchera à éliminer ces redondances par un procédé de **normalisation**, qui consiste en une **décomposition** adéquate de la table problématique.

Les données sont stockées sur des supports externes, le plus souvent des disques magnétiques. Ceux-ci sont structurés en **espaces de stockage** (ou *fichiers*) dans lesquels les lignes des tables sont enregistrées. Des **index** permettent d'accéder rapidement aux lignes qui possèdent des valeurs déterminées de certaines colonnes. Les lignes qui sont fréquemment demandées simultanément peuvent être stockées sous forme d'**agrégats** minimisant leur accès conjoint.

Les bases de données sont gérées par des logiciels complexes dénommés *systèmes de gestion de bases de données* (**SGBD**) offrant divers services tels que la création de structures de données, la modification et l'extraction de données et le contrôle d'accès.

2.14 POUR EN SAVOIR PLUS

Le lecteur trouvera au chapitre 3 une approche plus rigoureuse des structures de base de données, et en particulier des contraintes d'intégrité. Il y trouvera également un traitement plus complet du processus de normalisation. Les chapitres 4 et 5 aborderont de manière plus approfondie certains aspects techniques de la gestion des données, notamment la structure physique des données et les systèmes de gestion de bases de données.

La deuxième partie de cet ouvrage est consacrée à l'utilisation des concepts présentés dans ce chapitre, notamment au travers du langage SQL. Elle permettra de comprendre en profondeur et de maîtriser les concepts qui viennent d'être présentés.

2.15 PRATIQUER LES BASES DE DONNÉES

Cet ouvrage a avant tout une vocation pratique. Au-delà de la connaissance et de la compréhension des concepts, il est important que le lecteur puisse manipuler ceux-ci de manière concrète. La première mesure consiste à se connecter à un SGBD offrant une interface SQL ou à en installer un sur un ordinateur personnel. Cette dernière option est sans doute la plus pratique. Nous suggérons quelques pistes.

1. Le logiciel **SQLfast** permet d'exécuter très simplement des requêtes SQL ou des scripts (suite de requêtes) de complexité quelconque sans connaissances particu-

lières en programmation. SQLfast est développé en Python, exploite le SGBD SQLite et est disponible actuellement sous Windows, ou Wine sous Linux. Son installation, simple et rapide, est décrite à l'annexe A28, qui comprend également un tutoriel du langage SQLfast. Le logiciel peut être téléchargé gratuitement à partir du site de l'ouvrage. C'est cette approche que nous recommandons.

La figure 2.13 montre un exemple élémentaire mais complet de script SQLfast, qui ouvre une base de données (**openDB**), demande à l'utilisateur (**ask**) le nom d'une ville (figure 2.14), exécute une requête qui extrait (**select**) et affiche à l'écran des informations sur les clients de cette ville¹² (figure 2.15), puis ferme la base de données (**closeDB**).

```
openDB CLICOM.db;
ask ville = Nom de ville;
select NCLI,NOM from CLIENT where LOCALITE = '$ville$';
closeDB;
```

Figure 2.13 - Un script SQLfast

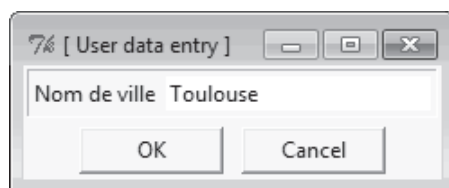


Figure 2.14 - Exécution du script : saisie d'un nom de ville

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

Figure 2.15 - Résultat de l'exécution du script à l'écran (format standard n° 1)

- La suite Office de Microsoft inclut le SGBD relationnel MS Access. Ce dernier constitue un *bac à sable* très convivial pour expérimenter les concepts des bases de données. Le langage SQL n'y est pas immédiatement visible mais nécessite quelques manipulations.
- On trouvera sur le web des SGBD légers et gratuits, faciles à installer et à utiliser. On recommandera par exemple **Firebird** (www.firebirdsql.org/), **InterBase 6** (www.ibprovider.com/eng) ou **SQLite** (www.sqlite.org).

12. **ville** est le nom d'une variable du script. L'instruction **ask** lui assigne la valeur introduite par l'utilisateur. L'expression **\$ville\$** désigne la valeur de la variable **ville**.

- 4. On envisagera aussi une des vedettes Open source : **MySQL** (www.oracle.com/us/sun/index.htm, www.mysql.com) ou **PostgreSQL** (www.postgresql.org/).
- 5. Enfin, les SGBD commerciaux offrent souvent des versions légères gratuites pour évaluation ou usage privé. **Oracle**, **DB2** d'IBM et **SQL Server** de Microsoft sont souvent plus délicats à installer mais constituent les *leaders* du marché.

2.16 EXERCICES

- 2.1 On considère le bon de commande papier de la figure 2.16, qu'on se propose d'encoder sous la forme de données à introduire dans la base de données de la figure 2.8. Qu'en pensez-vous ?

Commande N° : 30186		Date : 30/2/2009		
Numéro client B516				
Nom ASSRAN				
Adresse				
Localité Cassis				
N° PRODUIT	LIBELLE PRODUIT	PRIX	QUANTITE	SOUS-TOTAL
PA45	POINTE ACIER 45 (20K)	105	un	105
PA45	POINTE ACIER 45 (25K)	95	trois	285
TOTAL COMMANDE				422

Figure 2.16 - Un bon de commande curieux

- 2.2 Vérifier si le schéma suivant est normalisé. Si nécessaire, le décomposer en tables normalisées.

VENTE						
NPRO	CLIENT	DATE	QUANTITE	ADRESSE	DELEGUE	REGION

CLIENT —> ADRESSE, DELEGUE
DELEGUE —> REGION

- 2.3 Décomposer si nécessaire la table ci-dessous.

COMMANDE					
NCOM	NCLI	NOM	DATE	NPRO	LIBELLE

NCLI —> NOM
NPRO —> LIBELLE

Exercices corrigés

- **Exercice 2.1.** Les données de ce bon de commande présentent plusieurs anomalies qui en empêcheront l'introduction dans la base de données. (1) *Numéro de commande déjà présent dans la BD* : violation d'une contrainte d'unicité. (2) *Date de commande invalide* : violation du domaine de valeurs. (3) *Numéro de client inexistant* : violation d'une contrainte référentielle. (4) *Adresse du client manquante* : violation du caractère obligatoire d'une colonne. (5) *Deux détails référencent le même produit* : violation d'une contrainte d'unicité (identifiant de DETAIL). (6) *Les quantités sont exprimées en caractères* : violation du domaine de valeurs. (7) *Le produit PA45 possède deux libellés et deux prix* : violation de dépendances fonctionnelles. (8) *Le montant total est incorrect* : sans importance, il s'agit d'une donnée calculée non enregistrée.
- **Exercice 2.2.** La colonne REGION dépend d'une colonne qui n'est pas un identifiant. La table n'est pas normalisée. On la décompose en deux tables VENTE(NPRO, CLIENT, DATE, QUANTITE, ADRESSE, DELEGUE) et REP(DELEGUE, REGION). Ensuite, dans la nouvelle table VENTE, les colonnes ADRESSE et DELEGUE dépendent d'une colonne qui n'est pas un identifiant. Par décomposition, on obtient le schéma ci-dessous :

VENTE(NPRO, CLIENT, DATE, QUANTITE)

CLI(CLIENT, ADRESSE, DELEGUE)

REP(DELEGUE, REGION)

Deux clés étrangères : VENTE.CLIENT vers la table CLIENT et CLI.DELEGUE vers la table REP. Remarque importante : on traite d'abord les dépendances externes, dont le déterminé n'est pas un déterminant.

- **Exercice 2.3.** La colonne NOM dépend d'une colonne qui n'est pas un identifiant. La table n'est pas normalisée. On la décompose en deux tables COMMANDE(NCOM, NCLI, DATE, NPRO, LIBELLE) et CLIENT(NCLI, NOM). Ensuite, dans la nouvelle table COMMANDE, la colonne LIBELLE dépend d'une colonne qui n'est pas un identifiant. Par décomposition, on obtient le schéma suivant :

COMMANDE(NCOM, NCLI, DATE, NPRO)

CLIENT(NCLI, NOM)

PRODUIT(NPRO, LIBELLE)

Deux clés étrangères : COMMANDE.NCLI vers la table CLIENT et COMMANDE.NPRO vers la table PRODUIT.

Des exercices et solutions additionnels sont disponibles à l'annexe A2. L'installation et l'utilisation de SQLfast sont décrites à l'annexe A28.

Chapitre 3

Modèle relationnel et normalisation

Ce chapitre étudie un problème fréquent de redondance interne dans les bases de données. Ce phénomène est resitué dans le cadre du modèle relationnel et plus particulièrement celui de la théorie des dépendances fonctionnelles et de la normalisation. On propose des procédures de calcul des identifiants d'une table et de décomposition de tables sièges de redondances internes.

3.1 LE PHÉNOMÈNE DE REDONDANCE INTERNE

La section 2.11 nous a permis de mettre en évidence la présence de redondances internes dans une table (LIVRE) qui regroupe des données relatives à deux concepts différents (les ouvrages et leurs exemplaires). Nous avons également observé que cette redondance disparaissait lorsque, par une décomposition appropriée de la table d'origine, on répartissait les données dans deux tables OUVRAGE et EXEMPLAIRE dédiées chacune à l'un des deux concepts.

Le but de ce chapitre est d'étudier de manière rigoureuse le problème de la redondance interne au sein d'une table et les techniques qui permettent de l'identifier et de l'éliminer sans perte d'information. Nous situerons l'analyse du problème dans le cadre du *modèle relationnel* de base de données dont l'un des objectifs est l'étude et l'élimination de la redondance interne. Nous réexaminerons la notion d'identifiant puis nous introduirons celle de dépendance fonctionnelle, qui lui est très proche.

L'intérêt de ces matériaux va bien au delà de la question de la redondance dans les bases de données relationnelles. Ils seront en effet très utiles dans d'autres modèles

de données (chapitre 15), dans différents processus de la construction d'une base de données (chapitres 17 et 18) et en rétro-ingénierie (chapitre 21).

3.2 LE MODÈLE RELATIONNEL DE BASES DE DONNÉES

Le concept de *base de données* comme collection sur ordinateur des données d'une entreprise a émergé au début des années soixante. Cependant, l'histoire des bases de données relationnelles a véritablement démarré avec la publication en 1970 dans les *Communications of the ACM* d'un article d'Edgar F. Codd, chercheur d'IBM, proposant une organisation des données particulièrement simple, sous la forme de constructions mathématiques élémentaires dégagées de toutes considérations technologiques [Codd, 1970]. Nous ferons du modèle relationnel une présentation simplifiée suffisante pour l'étude de la redondance interne. La lecture préalable de l'annexe A3 pourrait être utile afin de rafraîchir les notions d'ensemble et de logique.

3.2.1 Concepts de base

Le modèle propose une structuration des données à l'aide de deux concepts élémentaires : le domaine et la relation.

Un *domaine* est un ensemble nommé de valeurs donné a priori. Chaque valeur est supposée désigner un objet concret ou abstrait du monde réel : un client, un nom, un prix, une date.

Une *relation* est un ensemble nommé d'agrégats de n valeurs, chacune appartenant à un domaine. Nous appellerons *ligne* un tel agrégat (la théorie préconise le terme de *n-uplet*). Les composants de même rang des lignes forment un *attribut* de la relation. Les attributs d'une relation portent des noms distincts.

On distingue dans une relation, d'une part, son *schéma* ou *intention* constitué de son nom et, pour chaque attribut, de son nom et de celui son domaine¹ et, d'autre part, l'ensemble de ses lignes, qu'on appelle son *extension*.

Quel rapport existe-t-il entre une *relation* et une *table* ? Une table est une représentation matérielle d'une relation, ou à l'inverse, une relation est une abstraction mathématique d'une table. Dans ce chapitre, nous parlerons de *relations* et d'*attributs*, mais on pourrait sans rien y changer remplacer ces termes par ceux de *tables* et de *colonnes*, pour autant que nous admettions les deux restrictions suivantes :

- les lignes de la table sont distinctes (elles forment un *ensemble*). On ne peut donc trouver dans une table deux lignes dont les colonnes ont mêmes valeurs.
- les valeurs *null* sont exclues.

Nous admettrons enfin que les valeurs des domaines sont *élémentaires*. Une valeur d'attribut ne peut donc être un agrégat de valeurs ni un ensemble de valeurs ou d'agrégats. On dira que la relation est *plate* ou en *première forme normale* (1FN).

1. Et, comme nous le verrons plus tard, des contraintes dont la relation est le siège.

Plutôt que la représentation graphique du schéma d'une table adoptée jusqu'ici, nous utiliserons la notation suivante, plus légère, qui indique le nom de la relation ainsi que le nom et le domaine de valeurs de chaque attribut :

OFFRE(CHAINES: char(20), PRODUIT: char(16), PRIX: decimal(6,2))

ou, plus simplement, si on ignore les domaines,

OFFRE(CHAINES, PRODUIT, PRIX)

Remarque. La théorie propose une définition très simple du concept de relation : une relation est une partie du produit cartésien² d'une suite de domaines. Soient $D_1, D_2, \dots, D_n$, des domaines non nécessairement distincts. Une relation (ou plutôt son extension) $R(A_1: D_1, A_2: D_2, \dots, A_n: D_n)$ est définie comme :

$$R \subseteq D_1 \times D_2 \times \dots \times D_n$$

3.2.2 Opérateurs d'extraction de données

Un modèle de données n'a d'utilité que s'il propose un moyen de dériver des données à partir des lignes des relations. Nous étudierons plus loin le langage SQL qui répondra à ce besoin³. Nous nous contenterons ici d'un ensemble réduit d'opérateurs très simples mais suffisants pour notre analyse. Ces opérateurs font partie de ce qu'on appelle l'algèbre relationnelle, qu'on retrouvera en partie dans le langage SQL. Nous n'en retiendrons que la projection, la jointure et la sélection, que nous illustrerons sur la petite base de données de la figure 3.1. La relation OFFRE indique quelles chaînes de magasins offrent un produit déterminé et à quel prix. La relation IMPLANTATION nous informe sur les villes où on trouve un magasin d'une chaîne déterminée.

OFFRE			IMPLANTATION	
CHAINE	PRODUIT	PRIX	CHAINE	VILLE
SUPER-U	Chocolat	6	SUPER-U	LILLE
MAMMOUTH	Sucre	2,2	SUPER-U	ANNECY
MAMMOUTH	Chocolat	5	MAMMOUTH	LYON
MATCH	Sel	1,4	MAMMOUTH	LILLE
MAMMOUTH	Sel	1,4	MATCH	PARIS
			MATCH	LILLE
			MATCH	TOURNAI
			CORA	CHARLEROI

Figure 3.1 - Deux relations

2. Le produit cartésien de deux ensembles E_1 et E_2 est l'ensemble de tous les couples qu'il est possible de construire en prenant un élément de E_1 et un élément de E_2 . Le produit cartésien se généralise à n ensembles.

3. Ce chapitre fait partie du parcours complet, supposé être effectué après le parcours introductif, lequel comprend une introduction à SQL.

a) Projection d'une relation

L'opérateur de projection produit une relation dont les attributs forment un sous-ensemble de ceux de la relation d'origine. Le résultat comporte, en principe, moins d'attributs que la relation d'origine ; elle peut aussi comporter moins de lignes lorsque plusieurs lignes de la relation d'origine ont mêmes valeurs pour les attributs de la projection (élimination des doubles). La projection de la relation OFFRE sur les attributs PRODUIT et PRIX nous donne une relation à deux attributs (figure 3.2) qui indique pour chaque produit à quel(s) prix il est offert, sans qu'on sache dans quelle(s) chaîne(s). On note que l'information selon laquelle le sel est offert au prix de 1,4 n'est reprise qu'une seule fois. La projection se note comme suit :

NIVEAU-des-PRIX = OFFRE[PRODUIT,PRIX]

NIVEAU-des-PRIX	
PRODUIT	PRIX
Chocolat	6
Sucre	2,2
Chocolat	5
Sel	1,4

Figure 3.2 - Résultat de la projection de OFFRE sur les attributs (PRODUIT, PRIX)

La projection s'exprime en SQL par la clause `select`. Attention cependant, il est important d'éliminer les doublons éventuels par le modificateur `distinct`⁴.

```
select distinct PRODUIT,PRIX
from OFFRE
```

b) Jointure naturelle de deux relations

La jointure de deux relations telles que OFFRE et IMPLANTATION sur les attributs CHAÎNE produit une relation dont les lignes sont constituées à partir d'une ligne d'OFFRE et d'une ligne d'IMPLANTATION dont les attributs CHAÎNE, dits *attributs de jointure*, ont mêmes valeurs.

Par exemple, la ligne (MAMMOUTH, Chocolat, 5) d'OFFRE et la ligne (MAMMOUTH, LILLE) d'IMPLANTATION seront *jointes* sous la forme (MAMMOUTH, Chocolat, 5, LILLE)⁵, qui fera donc partie du résultat, lequel est repris à la figure 3.3. On notera cette jointure comme suit :

OFFRE-VILLE = OFFRE(CHAÎNE)*IMPLANTATION(CHAÎNE)

ou, lorsqu'il n'y a pas d'ambiguïté sur les attributs de jointure,

OFFRE-VILLE = OFFRE*IMPLANTATION

4. Nécessaire lorsque la liste des attributs de la projection n'inclut pas tous les composants d'un identifiant de la relation d'origine.

5. Dans la jointure naturelle, les attributs de jointure ne sont repris qu'une seule fois.

La jointure s'exprimera en SQL par la mention des deux relations dans la clause `from` accompagnée d'une condition de jointure (section 8.2) :

```
select O.CHAINE, PRODUIT, PRIX, VILLE
from OFFRE O, IMPLANTATION I
where O.CHAINE = I.CHAINE
```

OFFRE-VILLE			
CHAINE	PRODUIT	PRIX	VILLE
SUPER-U	Chocolat	6	ANNECY
SUPER-U	Chocolat	6	LILLE
MAMMOUTH	Sucre	2,2	LILLE
MAMMOUTH	Sucre	2,2	LYON
MAMMOUTH	Chocolat	5	LILLE
MAMMOUTH	Chocolat	5	LYON
MATCH	Sel	1,4	TOURNAI
MATCH	Sel	1,4	LILLE
MATCH	Sel	1,4	PARIS
MAMMOUTH	Sel	1,4	LILLE
MAMMOUTH	Sel	1,4	LYON

Figure 3.3 - Jointure des relations OFFRE et IMPLANTATION sur l'attribut CHAINE

c) Sélection de lignes

La sélection est un opérateur qui, appliqué à une relation, retourne le sous-ensemble de ses lignes qui satisfont une condition de sélection. La sélection des offres dont le prix est inférieur à 5 s'écrit comme suit :

OFFRE-BUDGET = OFFRE(PRIX < 5)

d) Combinaison d'opérateurs

Puisque le résultat de l'application d'un opérateur de projection, de jointure ou de sélection est une relation, on peut appliquer à celle-ci d'autres opérateurs. L'expression suivante qui demande l'application d'une projection au résultat de la jointure d'une sélection et d'une relation de base, produit une relation qui donne, pour chaque produit dont le prix est inférieur à 5, les villes où il est offert :

PRODUIT-VILLE = (OFFRE(PRIX < 5)*IMPLANTATION)[PRODUIT, VILLE]

Notons encore la composition binaire, qu'on rencontrera souvent dans la suite. Soient $R(A,B)$ et $S(B,C)$. Leur composition, notée $R \circ S$, est égale à $(R(B)*S(B))[A,C]$.

L'équivalent en SQL de ces expressions est évident. Le langage algébrique est très puissant mais ne conduit certainement pas à des expressions aussi lisibles que ce qu'on aurait écrit en SQL. Là n'est évidemment pas son objectif.

e) Autres opérateurs

L'algèbre relationnelle comporte d'autres opérateurs permettant de dériver de nouveaux domaines et de nouvelles relations. Citons en particulier les opérateurs ensemblistes (union, intersection et différence), le produit cartésien, déjà mentionné, et la division. Ces opérateurs ont été étendus aux **multi-ensembles**, collections généralisant les ensembles, formées d'éléments non nécessairement distincts (voir annexe A24).

3.3 IDENTIFIANTS

Un identifiant⁶ d'une relation est un sous-ensemble de ses attributs tel qu'il ne peut exister à aucun moment plus d'une ligne possédant les mêmes valeurs de ces attributs. Examinons quelques propriétés de ce concept.

1. Un identifiant est *minimal* si aucune de ses parties n'est un identifiant. Si au contraire il comporte des attributs surnuméraires, qu'on peut supprimer sans qu'il perde son statut d'identifiant, il est dit *non minimal*.
2. Tout ensemble d'attributs dont une partie est un identifiant est aussi un identifiant. On en déduit qu'en ajoutant un attribut quelconque à un identifiant, on obtient encore un identifiant (non minimal).
3. L'ensemble des attributs d'une relation est un identifiant de celle-ci. En effet, par définition, une relation ne peut contenir deux lignes identiques (3.2.1). Cet identifiant est très souvent non minimal. *Conséquence* : toute relation possède au moins un identifiant.
4. Une relation peut posséder plusieurs identifiants minimaux. Il est habituel de désigner l'un d'entre eux comme l'*identifiant primaire* de la relation.
5. Il est possible qu'un attribut appartienne à plusieurs identifiants minimaux.

On indiquera un identifiant en soulignant ses composants de manière continue. Dans les situations plus complexes, on spécifiera un identifiant par la clause "id:" ou "id(R):". Les exemples suivants illustrent ces principes.

1. VENTE(ARTICLE, MAGASIN, PRIX, ...). Dans le domaine d'application décrit, un article n'est vendu que dans un seul magasin, et à prix fixe.
2. VENTE'(ARTICLE, MAGASIN, PRIX, ...). Ici, un article peut être vendu par plusieurs magasins. Son prix dépend du magasin qui le vend.
3. VENTE"(ARTICLE, MAGASIN, PRIX, ...). Un magasin peut vendre un même article à différents prix (selon une règle qui n'est pas précisée).

6. Contrairement à l'usage, nous n'avons pas retenu le terme *clé* (*key*) pour désigner le concept d'identifiant, afin d'éviter un nom auquel trop d'acceptions ont été associées dans le domaine des bases de données. Citons par exemple : *unique key*, *secondary key*, *superkey*, *foreign key*, *access key*, *record key*, *alternate record key*, *relative key*, *sequence key*, *encryption key*, *user key*, *private key*, *public key*, *master key*, *concatenated key*, *hierarchical key*, *prefix key*, *sort key*, etc.

4. EMPLOYE(MATR, NSS, NOM, ADRESSE). Cette relation est dotée de **deux** identifiants distincts, indiquant par là qu'un employé peut être identifié soit par son matricule (MATR), soit par son numéro de Sécurité sociale (NSS).
5. HORAIRE(PROF, HEURE, LOCAL)
 id: PROF, HEURE
 id: HEURE, LOCAL

La relation HORAIRE indique qu'un professeur fait cours à une heure déterminée dans un local déterminé. Le premier identifiant traduit le fait qu'un professeur, durant une heure déterminée, est présent dans le même local. Le second précise que dans un local, à une heure déterminée, on ne trouve qu'un seul professeur.

3.4 DÉPENDANCES FONCTIONNELLES

Si l'identifiant constitue incontestablement la propriété la plus simple et la plus utile d'une relation, il existe un concept qui lui est proche mais qui représente une propriété plus élémentaire, la *dépendance fonctionnelle*, ou, pour faire court, **DF**

3.4.1 Le phénomène de dépendance fonctionnelle

Nous approcherons le concept de dépendance fonctionnelle par un exemple avant d'en étudier les propriétés.

ACHAT		
CLIENT	PRODUIT	PRIX
André	Sucre	2,2
Marc	Sucre	2,2
Marc	Sel	1,4
Anne	Savon	1,4
Anne	Sel	1,4

Figure 3.4 - La relation ACHAT est le siège d'une dépendance fonctionnelle

Dans la relation ACHAT(CLIENT,PRODUIT,PRIX) de la figure 3.4, chaque ligne (c,p,x) spécifie que le client c a acheté le produit p au prix x. Il faut cependant préciser que le prix du produit ne dépend que de ce dernier et qu'il est donc indépendant du client qui effectue l'achat. On conclut donc que, quel que soit le client qui l'achète, le sucre coûte 2,2, le sel coûte 1,4 et le savon coûte également 1,4. Si un nouvel achat de sel devait être enregistré, on sait déjà que la valeur de PRIX serait 1,4, quel que soit le client.

Si, quel que soit le contenu de ACHAT, la valeur de l'attribut PRIX ne dépend que de celle de PRODUIT et pas des autres attributs (CLIENT), alors on dira qu'il existe une *dépendance fonctionnelle* de PRODUIT vers PRIX, ce qu'on notera :

ACHAT: PRODUIT $\longrightarrow$ PRIX

ou plus simplement, quand le contexte le permet :

PRODUIT $\longrightarrow$ PRIX

À une valeur déterminée de PRODUIT correspond une même valeur de PRIX dans toutes les lignes dans lesquelles cette valeur de PRODUIT apparaît. Si on construit une relation par projection de ACHAT sur PRODUIT et PRIX, alors PRODUIT en sera automatiquement l’identifiant.

Définition 3.1
Dans une relation $R(A,B,C,D)$, il existe une *dépendance fonctionnelle* $A \longrightarrow B$ si, à tout instant, deux lignes de R qui ont même valeur de A ont aussi même valeur de B .

Dans notre exemple, on dira que :

- PRODUIT *détermine* (fonctionnellement) PRIX ;
- PRIX *dépend* (fonctionnellement) de PRODUIT ;
- PRODUIT est le *déterminant* et PRIX est le *déterminé* de la dépendance fonctionnelle.

Le déterminant et le déterminé d’une DF peuvent comprendre plus d’un attribut. On pourrait donc trouver dans une relation $R(A,B,C,D)$ des dépendances telles que :

$B,C \longrightarrow D$ ou $B \longrightarrow C,D$

Sans perte de généralité, et sauf mention contraire, on ne considérera dans la suite que les DF *élémentaires*, dont le déterminé ne comporte qu’un seul attribut.

3.4.2 Un exemple plus complexe

La relation COM suivante est particulièrement complexe. Une ligne (a,b,c,d,e,f,g,h) indique que *le client n° a, de nom b et d’adresse c a passé la commande numéro d, à la date e, spécifiant le produit n° f en quantité g et au prix unitaire h.*

COM(NCLI, NOM, ADRESSE, NCOM, DATE, NPRO, QTE, PRIX-U)		
NCOM $\longrightarrow$ NCLI	<i>toute commande est émise par un client</i>	df1
NCLI $\longrightarrow$ NOM	<i>tout client a un nom</i>	df2
NCLI $\longrightarrow$ ADRESSE	<i>tout client a une adresse</i>	df3
NCOM $\longrightarrow$ DATE	<i>toute commande est passée à une certaine date</i>	df4
NCOM, NPRO $\longrightarrow$ QTE	<i>dans toute commande, il y a une quantité par produit</i>	df5
NPRO $\longrightarrow$ PRIX-U	<i>tout produit a un (et un seul) prix unitaire</i>	df6

Nous verrons plus loin que la complexité de cette relation, dont témoigne sa sémantique alambiquée, est la conséquence d’anomalies multiples induites par certaines dépendances fonctionnelles dont elle est le siège. Pour l’heure, le lecteur est invité à imaginer un schéma relationnel plus simple qui exprime de manière plus claire les mêmes informations.

3.4.3 Graphe ADF d'une relation (attributs et DF)

Il est possible de donner d'une relation une représentation graphique, dénommée *graphe des attributs et des DF* (graphe ADF), qui spécifie les attributs ainsi que les DF dont ils sont le siège (figure 3.5).

On appelle DF **externe** dans un graphe ADF toute DF dont le déterminé n'entre dans la composition d'aucun déterminant. Les autres DF sont dites **internes**.

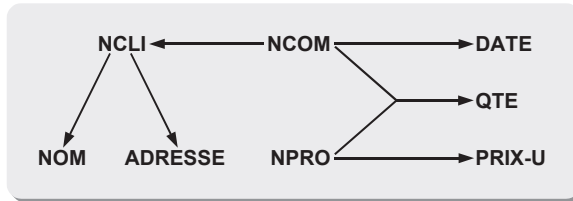


Figure 3.5 - Graphe ADF de la relation COM

La relation COM comporte une seule DF interne ($NCOM \longrightarrow NCLI$), toutes les autres étant externes.

Un attribut **externe** appartient à un déterminé mais à aucun déterminant (NOM, ADRESSE, DATE, QTE, PRIX-U). Un attribut **source** appartient à un déterminant mais pas à un déterminé (NPRO, NCOM). Un attribut **interne** appartient à un déterminant et à un déterminé (NCLI). Un attribut **isolé** n'appartient ni à un déterminant ni à un déterminé. Les attributs sources et isolés forment les attributs **racines**.

3.4.4 Propriétés des dépendances fonctionnelles

Nous étudierons quelques propriétés utiles des dépendances fonctionnelles, et en particulier les mécanismes permettant de construire de nouveaux ensembles de DF à partir d'un ensemble donné.

a) Règles de dérivation des dépendances fonctionnelles

Il existe un jeu de règles, dites *règles d'inférence d'Armstrong*, qui permettent de produire de nouvelles DF à partir de DF connues au sein d'une même relation. Elles sont au nombre de trois, mais il est habituel de les combiner sous la forme de six règles plus intuitives et plus pratiques. A et B désignent des attributs tandis que K, L et M désignent des ensembles d'attributs⁷.

1. *Réflexivité* : Si L est un sous-ensemble non vide de K, alors on a $K \longrightarrow L$. Un ensemble d'attributs détermine chacun de ses sous-ensembles. Cette propriété qui semble évidente produit ce qu'on appelle des *DF triviales*.
2. *Augmentation* : Si $K \longrightarrow B$, alors on a aussi $KA \longrightarrow BA$. Si on ajoute un attribut quelconque à un déterminant et à son déterminé, on obtient encore une DF.

7. On conviendra ici que l'expression KL désigne l'union de K et L ; que A,B désigne l'ensemble de ces deux attributs et que KA désigne l'ensemble formé de A et des attributs de K.

3. *Additivité* : Si $K \longrightarrow A$ et $L \longrightarrow B$, on a aussi $KL \longrightarrow AB$. Si on *additionne* (il s'agit en fait d'une *union*) respectivement les déterminants et les déterminés de deux DF, on obtient encore une DF.
4. *Décomposabilité* : Si $K \longrightarrow LM$, on a aussi $K \longrightarrow L$ et $K \longrightarrow M$. Pour toute DF, il existe une DF entre son déterminant et chaque attribut de son déterminé. Une DF dont le déterminé comprend un seul attribut est dite *élémentaire*.
5. *Transitivité* : Si $K \longrightarrow L$ et $L \longrightarrow M$, on a aussi $K \longrightarrow M$. En clair, la composition de deux DF est encore une DF.
6. *Pseudo-transitivité* : Si $K \longrightarrow L$ et $LM \longrightarrow N$, on a aussi $KM \longrightarrow N$.

Il existe d'autres règles liant des DF de plusieurs relations. Les quatre règles de propagation suivantes sont particulièrement importantes :

7. *Propagation vers la projection* : Si $K \longrightarrow L$ existe dans R, elle existe aussi dans toute projection de R qui inclut les attributs de K et L.
8. *Propagation à partir d'une projection* : Si $K \longrightarrow L$ existe dans une projection de R, elle existe aussi dans R.
9. *Propagation vers la sélection* : Si $K \longrightarrow L$ existe dans R, elle existe aussi dans tout sous-ensemble de R. Malheureusement, l'inverse n'est pas vrai !
10. *Propagation vers la jointure* : Si $K \longrightarrow L$ existe dans R, elle existe aussi dans toute jointure de R avec une relation S quelconque.

Exemple

Considérons la relation :

```
ORDRE (FOURN, NPRO, NCOM, ADR, REGION, QTE)
FOURN  $\longrightarrow$  ADR, NPRO
ADR  $\longrightarrow$  REGION
NCOM, NPRO  $\longrightarrow$  QTE
```

On en infère les DF supplémentaires suivantes dans ORDRE :

```
FOURN, NCOM  $\longrightarrow$  FOURN, NCOM
ADR, NCOM  $\longrightarrow$  REGION
ADR, NCOM, NPRO  $\longrightarrow$  REGION, QTE
FOURN  $\longrightarrow$  NPRO
FOURN  $\longrightarrow$  REGION
FOURN, NCOM  $\longrightarrow$  QTE
```

Dans la projection ORDRE[FOURN, NPRO, REGION] on a aussi :

```
FOURN  $\longrightarrow$  NPRO, REGION
```

La règle d'augmentation conduit à la notion de *DF minimale* : une DF est minimale si elle ne dérive pas d'une autre DF par augmentation. Autrement dit, si on enlève un attribut au déterminant d'une DF minimale, le résultat n'est plus une DF.

b) Dépendances de base, dépendances dérivées

Une DF de base d'une relation est spécifiée explicitement et ne peut se dériver d'autres DF de cette relation. Une DF dérivée est construite à partir d'autres DF par application des règles décrites précédemment, et plus particulièrement la règle de transitivité⁸. Soit la relation COLIS dont le schéma se présente comme suit :

COLIS(NCOL, NCLI, DATE, NOM, ADRESSE)

NCOL $\longrightarrow$ NCLI	df1
NCOL $\longrightarrow$ DATE	df2
NCLI $\longrightarrow$ NOM	df3
NCLI, DATE $\longrightarrow$ ADRESSE	df4

De la composition de *df1* avec *df3* on déduit : $\text{NCOL} \longrightarrow \text{NOM}$. De la composition de *df4* avec l'addition de *df1* et *df2* on déduit : $\text{NCOL} \longrightarrow \text{ADRESSE}$. Les DF *df1* à *df4* sont appelées *de base* tandis que les deux DF déduites sont dites *dérivées*. Lorsqu'on spécifie un schéma relationnel, on ne reprend que les DF de base. La figure 3.6 illustre graphiquement ces six dépendances.

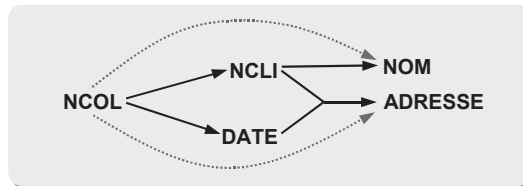


Figure 3.6 - DF de base (traits pleins) et DF dérivées de la relation COLIS (en pointillés)

c) Fermetures et couvertures minimales

À partir d'un ensemble F de DF d'une relation R , il est possible de dériver d'autres ensembles équivalents en ajoutant ou retirant des DF dérivées.

- On appelle *fermeture de F* par les règles d'Armstrong l'ensemble F^+ qui inclut F et qui contient en outre toutes les DF qu'on peut en dériver en appliquant itérativement les règles d'Armstrong. F ne possède qu'une seule fermeture F^+ .
- On appelle *couverture minimale de F* un plus petit ensemble de DF à partir duquel il est possible de reconstituer F par les règles d'Armstrong. F peut admettre plusieurs couvertures minimales. Elles ont la même fermeture que F .

La notion de fermeture existe aussi pour les attributs : si K est un ensemble d'attributs de R , on appelle fermeture de K selon F l'ensemble K^+ des attributs de R déterminés par K . Soit l'ensemble $K = \{\text{NCOM}\}$ de la relation COM. On détermine aisément $K^+ = \{\text{NCLI, NOM, ADRESSE, NCOM, DATE}\}$ (selon les DF de 3.4.2).

8. Dans certains cas, la propriété *de base* ou *dérivée* n'est pas intrinsèque et fait l'objet d'un choix. Par exemple, une relation $R(A,B,C, \dots)$ dans laquelle sont définies les DF $A \longrightarrow B$, $B \longrightarrow C$, $C \longrightarrow A$, admet comme dérivées les DF $B \longrightarrow A$, $C \longrightarrow B$, $A \longrightarrow C$. On pourrait cependant considérer les secondes comme DF de base et les premières comme DF dérivées.

d) Circuit de dépendances fonctionnelles

Un circuit est constitué d'une suite de DF telle que certains attributs intervenant dans ces DF se déterminent indirectement eux-mêmes. L'exemple de la figure 3.7 comporte un circuit formé de deux DF. On observe en effet que les attributs déterminés MATIERE et PROF apparaissent, complètement ou partiellement, dans leurs propres déterminants. Ces deux attributs font partie du circuit alors que ETUDIANT en est exclu.

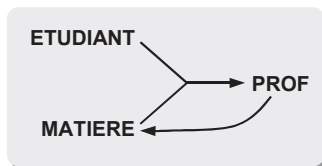


Figure 3.7 - Graphe comportant un circuit

On observe que dans cet exemple toutes les DF et tous les attributs sont internes. Les circuits de DF décrivent des structures intéressantes mais qui vont nous poser quelques problèmes tant dans le calcul des identifiants que dans la normalisation.

3.5 CONTRAINTES D'INCLUSION ET CLÉS ÉTRANGÈRES

Une contrainte d'inclusion précise que les valeurs d'un attribut (ou d'un groupe d'attributs) dans une relation doivent simultanément être des valeurs d'un attribut (ou groupe d'attributs) d'une autre relation (ou de la même relation).

Considérons les deux relations de la figure 3.1 en admettant qu'une offre ne puisse être enregistrée que si la chaîne dispose d'au moins une implantation. Cette propriété se traduit aisément par la contrainte d'inclusion :

$$\text{OFFRE}[\text{CHAINE}] \subseteq \text{IMPLANTATION}[\text{CHAINE}]$$

Lorsque le membre de droite de l'expression est défini sur les attributs d'un identifiant, la contrainte est dite *référentielle* et les attributs du membre de gauche forment une *clé étrangère* de la première relation. L'attribut NCLI de la relation COMMANDE (figure 2.7), qui est une clé étrangère vers la relation CLIENT, est soumise à la contrainte d'inclusion :

$$\text{COMMANDE}[\text{NCLI}] \subseteq \text{CLIENT}[\text{NCLI}]$$

Une contrainte d'inclusion peut prendre la forme d'une contrainte d'égalité. Revenons au schéma de la base de données qui nous sert d'exemple principal (figure 2.7). Les trois clés étrangères induisent les contraintes :

$$\begin{aligned} \text{COMMANDE}[\text{NCLI}] &\subseteq \text{CLIENT}[\text{NCLI}] \\ \text{DETAIL}[\text{NCOM}] &\subseteq \text{COMMANDE}[\text{NCOM}] \\ \text{DETAIL}[\text{NPRO}] &\subseteq \text{PRODUIT}[\text{NPRO}] \end{aligned}$$

Cependant, il serait légitime d'admettre que, s'il ne peut exister de détail sans commande (2^e contrainte), il ne peut non plus exister de commande sans détail. Cette dernière contrainte s'exprime sous la forme de la contrainte inverse :

$$\text{COMMANDE}[\text{NCOM}] \subseteq \text{DETAIL}[\text{NCOM}]$$

De ces deux contraintes inverses, on tire :

$$\text{COMMANDE}[\text{NCOM}] = \text{DETAIL}[\text{NCOM}]$$

Propriétés des contraintes d'inclusion

Tout comme les DF, les contraintes d'inclusion jouissent de propriétés qui permettent de dériver d'autres contraintes. Nous reprendrons cinq propriétés utiles dans le cadre de cet ouvrage. Soient $R(A,B,C)$, $S(A,B,C)$ et $T(A,B,C)$ trois relations dont on ignore les autres attributs.

1. *Réflexivité.* On a toujours : $R[A] \subseteq R[A]$.
2. *Décomposabilité.* Si on a $S[A,B] \subseteq R[A,B]$, on a aussi $S[A] \subseteq R[A]$ et $S[B] \subseteq R[B]$. On notera (à regret !) que la réciproque n'est pas vraie.
3. *Transitivité.* Si on a $T[A] \subseteq S[A]$ et $S[A] \subseteq R[A]$, on a aussi $T[A] \subseteq R[A]$.
4. *Propagation des DF.* Si on a $S[A,B,C] \subseteq R[A,B,C]$ et $R: A \longrightarrow B$, on a aussi $S: A \longrightarrow B$. Il s'agit d'une variante de la propriété 8 des dépendances fonctionnelles.
5. *Propagation vers la projection.* Si on a $S[A,B] \subseteq R[A,B]$ et $S' = S[A,B,C]$, on a aussi $S'[A,B] \subseteq R[A,B]$.
6. *Propagation vers la sélection.* Si on a $S[A,B] \subseteq R[A,B]$ et $S' \subseteq S$, on a aussi $S'[A,B] \subseteq R[A,B]$.
7. *Propagation vers la jointure.* Si on a $S[A,B] \subseteq R[A,B]$ et $S' = S * T$, on a aussi $S'[A,B] \subseteq R[A,B]$.

Ces règles restent valables lorsque A, B et C sont des groupes d'attributs. Ces propriétés nous permettent d'étudier de manière plus précise les structures problématiques mises en évidence dans l'annexe A2, que nous reprenons ci-après. Le schéma de la figure 3.8 comporte une clé étrangère composite {NCLI,NFOURN}. L'identifiant cible est lui-même constitué de deux clés étrangères. On pose la question suivante : peut-on substituer ces dernières à la clé étrangère composite ? En d'autres termes, les schémas 3.8 et 3.9 sont-ils équivalents ?

Les contraintes référentielles du schéma 3.8 s'expriment ainsi :

$$\text{COMPTE}[\text{NCLI}] \subseteq \text{CLIENT}[\text{NCLI}]$$

$$\text{COMPTE}[\text{NFOURN}] \subseteq \text{FOURNISSEUR}[\text{NFOURN}]$$

$$\text{ACHAT}[\text{NCLI},\text{NFOURN}] \subseteq \text{COMPTE}[\text{NCLI},\text{NFOURN}]$$

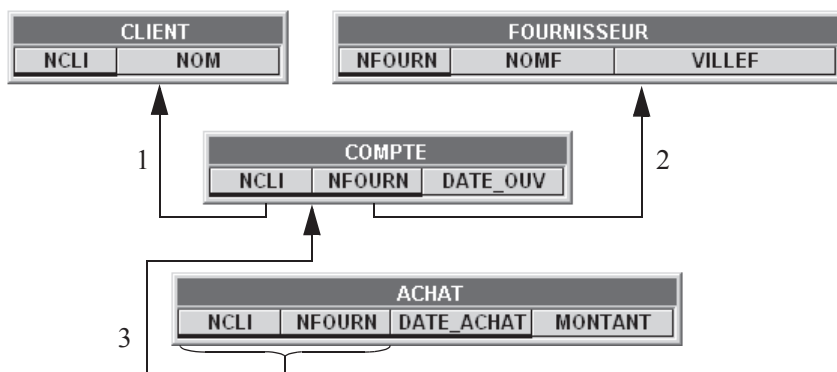


Figure 3.8 - Clé étrangère composite

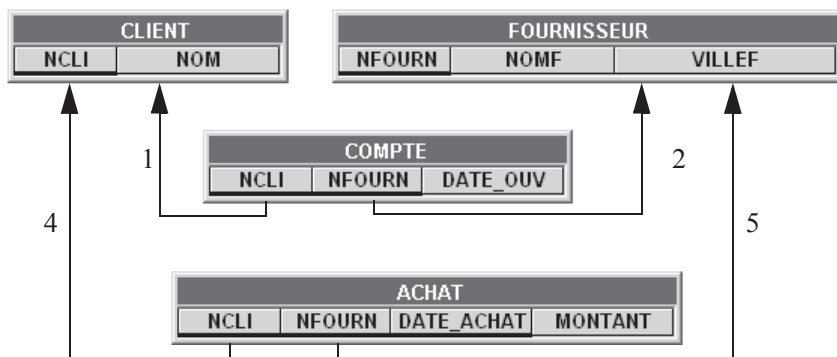


Figure 3.9 - Une version alternative du schéma de la figure 3.8 ?

Par la propriété de décomposabilité (2), on a aussi :

$$\begin{aligned} \text{ACHAT}[\text{NCLI}] &\subseteq \text{COMPTE}[\text{NCLI}] \\ \text{ACHAT}[\text{NFOURN}] &\subseteq \text{COMPTE}[\text{NFOURN}] \end{aligned}$$

Par transitivité (3), on obtient :

$$\begin{aligned} \text{ACHAT}[\text{NCLI}] &\subseteq \text{CLIENT}[\text{NCLI}] \\ \text{ACHAT}[\text{NFOURN}] &\subseteq \text{FOURNISSEUR}[\text{NFOURN}] \end{aligned}$$

qui représentent les contraintes 4 et 5 du schéma de la figure 3.9. En revanche, selon la note de la propriété de décomposabilité, il est impossible à partir des contraintes 4 et 5 de retrouver la contrainte 3, soit

$$\text{ACHAT}[\text{NCLI}, \text{NFOURN}] \subseteq \text{COMPTE}[\text{NCLI}, \text{NFOURN}]$$

du schéma initial. Le schéma 3.8 est donc **plus précis** que le schéma 3.9.

Rappelons enfin qu'une contrainte d'inclusion peut être bidirectionnelle, correspondant à une *contrainte d'égalité* : si $R[B] \subseteq S[B]$ et $S[B] \subseteq R[B]$ alors $R[B] = S[B]$.

Une clé étrangère imposant une contrainte d'égalité est dite *totale* et est notée *equ* plutôt que *ref*. Elle sera étudiée au chapitre 19.

3.6 CALCUL DES IDENTIFIANTS D'UNE RELATION

La normalisation d'une relation va reposer sur les rapports qui existent entre les identifiants et les dépendances fonctionnelles de cette relation.

Dans la plupart des exemples rencontrés jusqu'ici, la question du choix de l'identifiant ne s'est jamais posée explicitement, puisque ce dernier apparaissait comme évident : NCLI est objectivement l'identifiant de CLIENT et NPRO celui de PRODUIT. De même, on n'aura pas trop de peine⁹ à se persuader que l'identifiant de la relation ACHAT (figure 3.4) est {CLIENT, PRODUIT}. Mais comment raisonner dans les cas plus complexes tels que ceux des relations VENTE" ou HORAIRE de la section 3.3 ?

En fait, les identifiants ne se *choisissent* pas, ils *s'imposent*, ou plus exactement ils se *calculent* à partir de l'ensemble des DF dont la relation est le siège. Nous examinerons une procédure intuitive qui permet de résoudre la plupart des cas qui se présentent en pratique.

3.6.1 Relation entre identifiants et dépendances fonctionnelles

Observons d'abord que les identifiants et les dépendances fonctionnelles sont des concepts très proches. En exprimant l'un à partir de l'autre, on constate que :

- tout identifiant d'une relation détermine chaque attribut de cette relation ;
- si un groupe d'attributs détermine chacun des attributs de la relation, alors on peut déclarer ce groupe identifiant de la relation.

De ces observations on tire une règle, simple en apparence, de recherche des identifiants d'une relation : *on recherche tous les groupes qui déterminent chaque attribut de la relation*. Cette règle est malheureusement inefficace. En effet, elle nous obligerait à répertorier tous les groupes d'au moins un attribut et à calculer pour chaque groupe l'ensemble des attributs qu'il détermine directement (par des DF de base) et indirectement (par des DF dérivées). Un ensemble de N éléments admettant $(2^N - 1)$ sous-ensembles non vides distincts, toute recherche exhaustive serait irréaliste. Il est possible d'améliorer cette procédure (inutile par exemple d'examiner tous les sur-ensembles d'un identifiant reconnu), mais elle reste inutilisable en pratique. Il faut donc trouver une autre technique.

Deux propriétés, dont la première est déjà connue, vont nous aider.

1. L'ensemble des attributs d'une relation constitue un identifiant de cette relation. Ainsi, l'ensemble {A,B,C} est un identifiant, peut-être non minimal, de la relation $R(A,B,C)$.

9. Même si certains devront sans doute un peu réfléchir avant de se prononcer !

2. Soit J un identifiant multi-attributs d'une relation. Soit K un sous-ensemble de J et C un attribut de J extérieur à K (figure 3.10). Si $K \longrightarrow C$, alors J , duquel on retire C , c'est-à-dire $J - \{C\}$, est aussi un identifiant de la relation. Ainsi, si $R(\underline{A}, \underline{B}, \underline{C}, D)$ et $B \longrightarrow C$, alors $R(\underline{A}, \underline{B}, C, D)$.

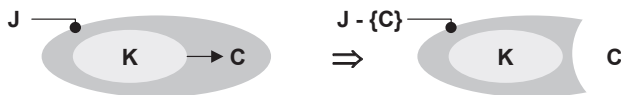


Figure 3.10 - Réduction d'un identifiant

D'où une première idée de procédure (Procédure 3.1). Nous allons affiner cette procédure selon que la relation possède un ou plusieurs identifiants.

Procédure 3.1 : Calcul de l'identifiant d'une relation R (procédure de base)

1. Un premier identifiant J est constitué de l'ensemble des attributs de R
2. On recherche dans J un attribut C externe dans le graphe ADF de J ; on retire C de J
3. On répète l'étape 2 jusqu'à ce qu'il ne soit plus possible de retirer d'attribut à J
4. J est l'identifiant (non nécessairement minimal) de R .

Deux observations

- On n'exige pas que les DF soient exclusivement de base.
- On commence par les attributs externes pour éviter d'inhiber certaines DF, comme dans l'exemple suivant. Soit $R(A, B, C)$; $A \longrightarrow B$; $B \longrightarrow C$. Eliminer C puis B conduit à l'identifiant $\{A\}$ tandis qu'éliminer B d'abord détruit la DF $B \longrightarrow C$ et produit l'identifiant non minimal $\{A, C\}$ ¹⁰.

3.6.2 La relation possède un seul identifiant

Une relation dont le graphe ADF est sans circuit ne possède qu'un seul identifiant minimal. Celui-ci se trouve aisément par la simple application de la procédure de base Procédure 1. On observe qu'il est constitué de tous les attributs racines.

Exemple 1

La relation COM (figure 3.5) se traite comme suit :

1. L'identifiant de départ est l'ensemble $\{NCLI, NOM, ADRESSE, NCOM, DATE, NPRO, QTE, PRIX-U\}$.
2. On en retire les attributs externes DATE, QTE, PRIX-U, NOM et ADRESSE.
3. On en retire l'attribut NCLI, qui est devenu externe.

Il reste l'identifiant $\{NCOM, NPRO\}$ (figure 3.11)

10. Sauf si on travaille avec la fermeture des DF, ce qu'on cherche à éviter.

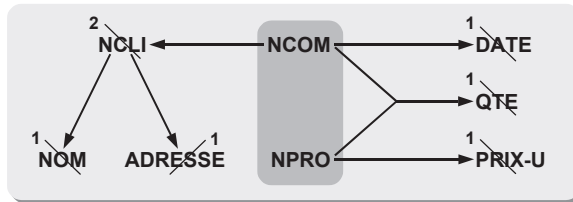


Figure 3.11 - Calcul de l'identifiant de COM en deux vagues

Exemple 2

La relation COLIS, dont le graphe ADF est repris à la figure 3.6, possède un identifiant obtenu comme suit :

1. L'identifiant de départ est l'ensemble {NCOL, NCLI, DATE, NOM, ADRESSE}.
 2. On en retire les attributs externes ADRESSE et NOM.
 3. On retire ensuite les attributs NCLI et DATE qui sont ainsi devenus externes.
- On obtient l'identifiant minimal {NCOL}.

Exemple 3

L'identifiant de la relation ACHAT s'obtient par l'élimination de l'attribut PRIX. Son identifiant est {CLIENT, PRODUIT}.

3.6.3 La relation possède plusieurs identifiants

Si le graphe ADF comporte un circuit, alors la relation possède plusieurs identifiants, ce qui complique quelque peu les choses. L'application de la procédure 3.1 produit un ensemble d'attributs (appelé *noyau irréductible*) parmi lesquels subsistent des *DF internes* dont certaines forment un circuit. Le circuit en effet ne propose plus d'attributs externes. Cet ensemble constitue bien un identifiant, mais celui-ci n'est pas minimal. En effet, il inclut plusieurs identifiants plus réduits.

Le schéma suivant présente un circuit de DF comme le montre son graphe ADF (figure 3.12, gauche).

```

COURS(MATIERE, ETUDIANT, PROF, CLASSE, TEL)
ETUDIANT → CLASSE
PROF → TEL
PROF → MATIERE
ETUDIANT, MATIERE → PROF

```

Les attributs CLASSE et TEL peuvent être retirés de l'identifiant initial (figure 3.12, droite). Il reste alors un noyau d'attributs {MATIERE, ETUDIANT, PROF} qui ne peut plus être réduit selon la procédure 3.1 puisqu'il n'y existe plus d'attributs externes.

L'identifiant obtenu est donc formé de ce noyau. Cependant, un examen plus attentif montre que cette conclusion n'est pas satisfaisante, puisque le sous-ensemble {MATIERE, ETUDIANT} est de toute évidence un identifiant.

Observons que la propriété 2 de la section 3.6.1 veut que l'attribut C qu'on retire de J soit déterminé mais elle n'exige pas qu'il soit *externe* dans son graphe ADF.



Figure 3.12 - Un graphe ADF comportant un circuit. À droite, le noyau irréductible

On peut donc retirer PROF ou MATIERE du noyau sans que le reste, {MATIERE, ETUDIANT} ou {ETUDIANT, PROF}, perde la propriété d'identifiant. On procède dès lors comme suit.

Pour chaque attribut du circuit, on le retire du noyau, ainsi que les DF dans lesquelles il apparaît. On obtient ainsi deux graphes partiels (figure 3.13), auxquels on applique la procédure classique 3.1. On obtient immédiatement les deux identifiants minimaux {MATIERE, ETUDIANT} et {ETUDIANT, PROF}.



Figure 3.13 - Deux sous-graphes extraits du noyau irréductible fournissant les identifiants

En toute généralité, chaque sous-graphe obtenu par le retrait d'un attribut du circuit est susceptible de comporter encore d'autres circuits. La procédure décrite doit donc être appliquée récursivement jusqu'à la réduction de tous les circuits. La procédure complète est développée ci-dessous (Procédure 3.2).

Procédure 3.2 : Calcul de l'identifiant d'une relation R (procédure complète)

1. Un identifiant est obtenu par application de la Procédure 3.1.
2. Si cet identifiant n'est le siège d'aucune DF, la procédure est terminée.
3. Sinon, le graphe ADF résiduel comporte un ou plusieurs circuits. Pour chaque attribut A appartenant à un circuit, on applique l'étape 4.
4. On retire A du graphe ADF ainsi que toutes les DF dans lesquelles il apparaît. On applique au graphe résultant la Procédure 3.2.

3.6.4 Rôle des attributs racines

Un examen attentif des procédures décrites ci-avant conduit à une conclusion des plus intéressantes. On observe que :

- l'identifiant unique d'une relation est formé de ses attributs racines,
- si une relation comporte un ou plusieurs circuits de DF, chaque identifiant contient tous les attributs racines.

3.7 DÉCOMPOSITION D'UNE RELATION

La propriété suivante, établie dans [Delobel, 1973], va nous être particulièrement utile. Soit la relation $R(A,B,C,D)$, dans laquelle A , B , C et D sont des attributs, ou plus généralement des ensembles d'attributs. On a :

$$A \longrightarrow B \Rightarrow R = R[A,B] * R[A,C,D]$$

Que dit cette propriété ? On considère une relation R siège d'une dépendance fonctionnelle $A \longrightarrow B$. On calcule d'une part sa projection sur les attributs du déterminant + ceux du déterminé, soit $R[A,B]$. On calcule d'autre part sa projection sur les attributs du déterminant + tous les autres attributs, qui forment le *résidu*, soit $R[A,C,D]$. On calcule enfin R' , la jointure de ces deux projections. La propriété nous garantit que $R' = R$.

De manière résumée, elle exprime le fait qu'une décomposition selon une DF **préserve les données**. La principale conséquence de cette propriété est que les deux schémas suivants sont équivalents ; on peut donc remplacer le schéma de gauche par celui de droite et inversement :

$R(A,B,C,D)$ $A \longrightarrow B$	$\Rightarrow$	$R1(\underline{A},B)$ $R2(A,C,D)$

La décomposition obéit au canevas suivant, qu'il importe de bien mémoriser :

$R(\text{déterminant, déterminé, résidu})$	$\Rightarrow$	$R1(\underline{\text{déterminant}}, \text{déterminé})$ $R2(\text{déterminant, résidu})$

Une propriété importante de toute décomposition est la propagation des dépendances fonctionnelles. Si une dépendance $K \longrightarrow L$ existe dans la relation d'origine, elle existe aussi dans tout fragment qui reprend les attributs de K et de L (propriété 7, section 3.4.4). C'est ainsi qu'on déduit l'identifiant de $R1$ ci-dessus. Cette propriété est valable tant pour les **DF de base** que pour les **DF dérivées**. On peut donc dériver toutes les DF des fragments à partir des DF de la fermeture de la relation d'origine.

Il reste à spécifier le lien qui existe entre les fragments $R1$ et $R2$. Il est clair que la décomposition a *découplé* des données qui étaient en relation dans R , et qu'on doit donc ajouter une propriété qui recrée le lien perdu. L'expression de ce lien dépendra de l'objectif qu'on assigne à la décomposition.

- Si on considère que $\{R\}$ et $\{R1, R2\}$ sont des bases de données (schéma + données) qui doivent être **strictement équivalentes**, alors on doit exiger que $R1[A] = R2[A]$, ce qui nous conduit à préciser le canevas de décomposition comme suit :

$R(A,B,C,D)$ $A \longrightarrow B$	$\Leftrightarrow$	$R1(\underline{A},B)$ $R2(A,C,D)$ $R1[A] = R2[A]$

- En revanche, si cette décomposition s’inscrit dans un **processus évolutif de conception** d’un schéma auquel n’est encore associée aucune donnée, alors la décomposition constitue une *amélioration* du schéma qui n’exige pas une équivalence stricte. C’est alors la signification des données qui imposera la relation d’inclusion qui convient : $R1[A] = R2[A]$, $R1[A] \subseteq R2[A]$ ou $R1[A] \supseteq R2[A]$. Dans la plupart des cas, nous déclarerons dans la relation R2 l’attribut (ou ensemble d’attributs) A *clé étrangère* vers la relation R1. Le principe de décomposition est schématisé dans la figure 3.14.

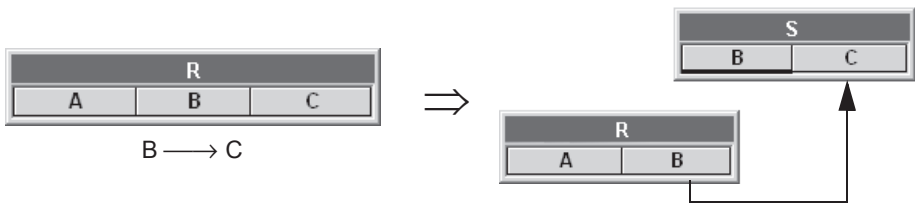


Figure 3.14 - Schéma de décomposition selon une dépendance fonctionnelle

Trois exemples de décomposition valide

ACHAT (CLIENT,PRODUIT,PRIX) PRODUIT —> PRIX	⇒	TARIF (PRODUIT,PRIX) ACHAT (CLIENT,PRODUIT) ACHAT[PRODUIT] ⊆ TARIF[PRODUIT]
ACHAT2 (CLI,PRO,MAG,PRIX) PRO, MAG —> PRIX	⇒	TARIF (PRO,MAG,PRIX) ACHAT2 (CLI,PRO,MAG) ACHAT2[PRO,MAG] ⊆ TARIF[PRO,MAG]
COM(NCOM,CLI,PRO,QTE)	⇒	PASSE (NCOM,CLI) COM (NCOM,PRO,QTE) COM[NCOM] = PASSE[NCOM]

Une décomposition qui ne respecterait pas la règle énoncée serait à priori parfaitement légitime, en ce qu’elle produirait des données correctes. Cependant, on ne pourrait garantir que la jointure des fragments restituerait la relation initiale. L’ensemble des fragments ne serait plus équivalent à la relation d’origine. En reprenant l’exemple de la relation ACHAT, le lecteur se convaincra sans difficulté que :

$$\begin{aligned} \text{ACHAT} &\subseteq \text{ACHAT}[\text{CLIENT, PRODUIT}] * \text{ACHAT}[\text{CLIENT, PRIX}] \\ \text{ACHAT} &\subseteq \text{ACHAT}[\text{PRODUIT, PRIX}] * \text{ACHAT}[\text{CLIENT, PRIX}] \end{aligned}$$

3.8 NORMALISATION D’UNE RELATION

Normaliser une relation consiste à lui appliquer une ou plusieurs décompositions préservant les données afin d’éliminer les problèmes de redondance interne dont elle est éventuellement le siège.

3.8.1 Réexamen du phénomène de redondance interne

Considérons à nouveau la relation ACHAT (figure 3.4) dont on rappelle le schéma :

ACHAT(CLIENT, PRODUIT, PRIX)
PRODUIT $\longrightarrow$ PRIX

L'examen des données montre que les lignes contiennent des représentations multiples de mêmes faits élémentaires. Par exemple, si le fait qu'*André a acheté du sucre* ou celui que *Marc a acheté du savon* sont représentés une seule fois, il n'en va pas de même des faits suivants : *le sucre coûte 2,2* ou *le sel coûte 1,4*, qui sont chacun représentés deux fois. Plus précisément, le fait que *le sucre coûte 2,2* est représenté autant de fois qu'il y a de clients qui en ont acheté.

Ce problème de redondance trouve son origine dans le fait que la relation ACHAT représente deux types de faits indépendants :

- *les clients achètent des produits*, qui est exprimé dans le fragment ACHAT[CLIENT,PRODUIT] ;
- *chaque produit a un prix déterminé*, correspondant au fragment ACHAT[PRODUIT,PRIX].

On observe en outre que le type de faits redondant (*chaque produit a un prix déterminé*) est précisément celui que représente la dépendance fonctionnelle PRODUIT $\longrightarrow$ PRIX explicitement indiquée dans le schéma.

En décomposant ACHAT selon cette DF on découple les deux types de faits en les isolant chacun sous la forme d'une relation autonome. Ainsi que le montre la figure 3.15, les redondances ont disparu. Cette décomposition est valide et préserve les données puisqu'elle obéit au principe de décomposition de la section 3.7.

ACHAT'		TARIF	
CLIENT	PRODUIT	PRODUIT	PRIX
André	Sucre	Sucre	2,2
Marc	Sucre	Sel	1,4
Marc	Sel	Savon	1,4
Anne	Savon		
Anne	Sel		

Figure 3.15 - Décomposition sans perte ni redondance de la relation ACHAT

3.8.2 Analyse du phénomène

La redondance est assez simple à expliquer. Nous avons vu que l'identifiant de la relation ACHAT est {CLIENT, PRODUIT}. Il ne peut donc exister plus d'une ligne possédant des valeurs déterminées de CLIENT et PRODUIT. Comme cet identifiant est minimal, aucun de ses sous-ensembles stricts, tel que {PRODUIT} par exemple, n'est un identifiant. Il existera donc en général plusieurs lignes possédant une valeur déterminée de PRODUIT. En raison de la DF PRODUIT $\longrightarrow$ PRIX, à cette valeur de PRODUIT sera associée la **même** valeur de PRIX. D'où la redondance.

En résumé, la redondance a pour origine la présence d'un attribut (PRIX) dépendant d'un ensemble d'attributs (PRODUIT) qui n'est pas un identifiant de la relation. C'est donc bien la DF PRODUIT $\longrightarrow$ PRIX qui provoque le phénomène litigieux.

3.8.3 Relation normalisée

Cette analyse se traduit comme suit : une DF dont le déterminant est un identifiant n'entraîne pas de redondance (on exclut du raisonnement les DF triviales). Une relation dont toutes les DF satisfont cette propriété est sans redondance et est dite *normalisée*. Dans le cas contraire, cette relation est *non normalisée*. On appellera *DF anormale* une DF non triviale dont le déterminant n'est pas un identifiant.

Définition 3.2 :

Une relation R est *normalisée* si, simultanément :

1. elle est en 1^{re} forme normale
2. le déterminant de chaque DF non triviale de R est un identifiant de R.

Une DF n'est en soi ni bonne ni mauvaise. C'est le rapport qui existe entre son déterminant et les identifiants de la relation qui induit ou non une redondance¹¹.

3.8.4 Normalisation d'une relation

À ce stade, nous sommes à même de diagnostiquer le degré de normalisation d'une relation. Celle-ci est *non normalisée* s'il y existe des DF anormales.

Il reste à définir une procédure qui transforme une relation non normalisée en une collection de relations normalisées équivalentes. Cette procédure est basée sur le principe de décomposition étudié à la section 3.7.

Le principe est simple : si on décompose une relation non normalisée selon une DF anormale, on obtient deux nouvelles relations dans lesquelles la DF ne pose plus de problèmes. Il suffit donc de traiter de la même manière les fragments obtenus de manière itérative jusqu'à ce que les relations ne comprennent plus de DF anormales. Deux questions se posent :

1. *Quel ensemble de DF considère-t-on ?*
Une analyse simple mais que nous ne reprendrons pas ici montre qu'on doit se limiter aux **DF de base**. Il faut donc écarter les DF dérivées.
2. *Dans quel ordre traite-t-on les DF anormales ?*
On peut également montrer qu'une décomposition doit se faire selon une **DF anormale externe**.

À partir du canevas de la section 3.7, on observe que chaque décomposition :

- génère une nouvelle relation R1(déterminant, déterminé) ;
- réduit la relation d'origine R par retrait des attributs du déterminé. Les relations R et R2 ayant même signification, on convient d'attribuer le nom R à R2.

11. On entend trop souvent dire : *la relation n'est pas normalisée parce qu'il y existe des DF !*

Le résultat doit encore être affiné : regroupement (c'est-à-dire *jointure*) éventuel des relations de même identifiant, choix de noms significatifs pour les nouvelles relations, expression des contraintes référentielles. Ces observations suggèrent la procédure intuitive de normalisation d'une relation Procédure 3.3. Cette procédure n'est pas complète mais sera suffisante dans la plupart des situations que nous rencontrons en pratique. Il peut cependant subsister une anomalie, lorsque la dernière version de R comporte encore des DF anormales. Il s'agira de DF internes qui ne peuvent être éliminées par la procédure simplifiée. Nous les traiterons à la section 3.8.5.

Procédure 3.3 : Normalisation d'une relation R (procédure simplifiée)

1. Dessiner le graphe ADF de la relation réduit aux DF de base
2. Calculer les identifiants minimaux
3. Marquer les DF anormales
4. Tant qu'il existe une DF anormale externe $K \longrightarrow L$, générer une relation $R_L(K, L)$ et retirer L de R ; préciser la contrainte référentielle
5. Finalisation : regroupement, noms significatifs, contraintes référentielles.

Exemple

Soit la relation COM dont le graphe ADF est donné à la figure 3.5. Ce graphe ne reprend que les DF de base. L'unique identifiant {NCOM, NPRO} de la relation a déjà été calculé. Les DF anormales sont marquées sur le graphe ADF de la figure 3.16.

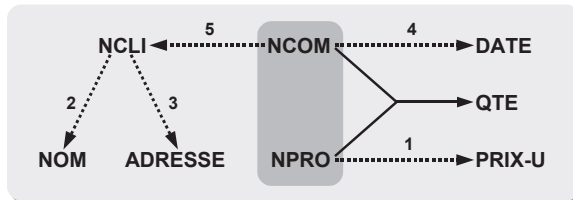


Figure 3.16 - En pointillés, les DF anormales de la relation COM

Les cinq itérations successives de l'étape 4 produisent les résultats suivants :

- *état initial*
COM(NCLI, NOM, ADRESSE, NCOM, NPRO, DATE, QTE, PRIX-U)
- *itération 1*
DF anormale externe : NPRO $\longrightarrow$ PRIX-U
R1(NPRO, PRIX-U)
COM(NCLI, NOM, ADRESSE, NCOM, NPRO, DATE, QTE)
- *itérations 2 et 3*
DF anormales externes : NCLI $\longrightarrow$ NOM; NCLI $\longrightarrow$ ADRESSE
R1(NPRO, PRIX-U)
R2(NCLI, NOM)
R3(NCLI, ADRESSE)
COM(NCLI, NCOM, NPRO, DATE, QTE)

- *itérations 4 et 5*

DF anormales externes : $\text{NCOM} \longrightarrow \text{DATE}$; $\text{NCOM} \longrightarrow \text{NCLI}$;
 R1(NPRO, PRIX-U)
 R2(NCLI, NOM)
 R3(NCLI, ADRESSE)
 R4(NCOM, DATE)
 R5(NCOM, NCLI)
 COM(NCOM, NPRO, QTE)

Finalisation : il semble pertinent de regrouper les relations R2 et R3 d'une part, et R4 et R5 d'autre part, considérant qu'elle ont le même identifiant :

R1(NPRO, PRIX-U)
 R23(NCLI, NOM, ADRESSE)
 R45(NCOM, NCLI, DATE)
 COM(NCOM, NPRO, QTE)

En donnant aux relations R1, R23 et R45 des noms significatifs et en reconstituant les contraintes référentielles, on obtient les relations normalisées :

TARIF(NPRO, PRIX-U)
 CLIENT(NCLI, NOM, ADRESSE)
 COMMANDE(NCOM, DATE, NCLI)
 DETAIL(NCOM, NPRO, QTE)
 COMMANDE.NCLI : *clé étrangère vers* CLIENT
 DETAIL.NCOM : *clé étrangère vers* COMMANDE
 DETAIL.NPRO : *clé étrangère vers* TARIF

Remarque sur la pertinence des regroupements

Le regroupement de deux relations de même identifiant doit être envisagé avec prudence. Qu'en serait-il par exemple des deux relations suivantes, dont l'interprétation est assez évidente ?

RP(MATRICULE, DATE, PROMOTION)
 RM(MATRICULE, DATE, MALADIE)

Le regroupement produirait la relation RPM :

RPM(MATRICULE, DATE, PROMOTION, MALADIE)

. . . dont l'interprétation pourrait être : si $(m,d,p,l) \in \text{RPM}$, alors, à la date **d**, l'employé de matricule **m** a reçu une promotion au grade **p** et a déclaré souffrir de la maladie **l**. Reste à déterminer si, dans cette entreprise, il faut attendre d'être malade pour espérer recevoir une promotion, ou si toute promotion entraîne automatiquement un congé de maladie ! Il est évident qu'ici, le regroupement n'est pas pertinent et que les deux relations RP et RM doivent être conservées telles quelles.

Ce raisonnement intuitif sur les promotions, les maladies et leur simultanéité potentielle peut s'exprimer en termes plus rigoureux. La section 3.7 définit les conditions de *décomposabilité* d'une relation, mais aussi, celles de son inverse, la *joignabilité* : quand deux relations sont-elles équivalentes à leur jointure ? Il faut et il suffit que leurs projections sur les attributs servant à la jointure soient égales, comme l'exprime la règle ci-dessous, lue de droite à gauche :



Nous en savons désormais assez pour justifier la pertinence du regroupement (en fait de la jointure) des deux relations RP et RM de même identifiant. Ces relations ne peuvent être regroupées que si :

$$RP[MATRICULE, DATE] = RM[MATRICULE, DATE]$$

Cette propriété n'étant en aucune manière garantie dans notre exemple, le regroupement projeté s'avère non pertinent. CQFD

3.8.5 Normalisation d'un noyau irréductible

Lorsque le graphe ADF comporte un circuit, la procédure de décomposition proposée ne permet pas de réduire celui-ci. On retrouve en effet ce circuit dans le noyau irréductible. Si ce noyau comporte encore une DF anormale, alors il représente une relation non normalisée¹².

Il n'existe malheureusement pas de représentation unique de ce noyau. Pire, nous allons montrer que parmi les représentations possibles, il n'en existe aucune qui soit totalement satisfaisante. Nous proposerons trois solutions parmi lesquelles on choisira celle dont on accepte les inconvénients. Nous raisonnerons à partir d'un exemple très simple de noyau irréductible, mais qu'il sera aisé de généraliser :

$$R(A, B, C)$$

$$A, B \longrightarrow C$$

$$C \longrightarrow B$$

a) La peste

Le noyau irréductible est traduit en une relation assortie des DF anormales. Cette relation comporte donc des redondances internes qu'il faudra gérer par des mécanismes tels que des *déclencheurs* SQL (voir section 9.9). L'application de cette approche à l'exemple produit la relation suivante, la DF $A, B \longrightarrow C$ s'exprimant par un identifiant de R :

$$R(\underline{A}, \underline{B}, C)$$

$$C \longrightarrow B$$

b) Le choléra

On poursuit la décomposition de R selon la DF anormale $C \longrightarrow B$, ce qui conduit aux deux relations $R1(C,B)$ et $R2(A,C)$. La redondance interne a disparu, mais il n'est plus possible d'exprimer la DF $A, B \longrightarrow C$ puisque ses attributs ont été répartis dans plusieurs relations. La décomposition *préserve les données* mais *ne préserve pas les*

12. Un noyau irréductible comportant un circuit mais aucune DF anormale ne pose pas de problème. Montrer qu'il en est bien ainsi (voir exercice 3.5).

dépendances. On peut bien sûr la redéfinir comme une DF présente dans la jointure des deux fragments, mais sa gestion en SQL sera complexe et inefficace. La solution proposée par cette approche est la suivante :

$R1(\underline{C}, B)$
 $R'(A, C)$
 $R'.C$: clé étrangère vers $R1$
 $R1 * R'$: $A, B \longrightarrow C$

c) La peste et le choléra

La troisième technique est en apparence plus complexe que les deux autres. Elle consiste à conserver la relation $R(A, B, C)$ de l'approche 1 mais aussi la relation $R1(\underline{C}, B)$ de l'approche 2. La relation entre R et $R1$ est exprimée par une contrainte d'inclusion :

$R(A, B, C)$
 $R1(\underline{C}, B)$
 $R[B, C] \subseteq R1[B, C]$

La redondance est ici maximale. En revanche, toutes les DF sont traduites par des identifiants gérables par le SGBD. En outre, la contrainte d'inclusion, en apparence complexe, peut se traduire de manière très élégante. On déclare d'abord $\{C\}$ identifiant primaire et $\{C, B\}$ identifiant secondaire de $R1$. On déclare ensuite dans R $\{C, B\}$ clé étrangère vers $R1$. Cette approche produit une solution qui peut être entièrement traduite en structures SQL classiques sans aucun recours à des mécanismes additionnels tels que les déclencheurs. Cette clé étrangère est qualifiée de *non minimale*. Voir à ce sujet sa gestion dans l'exercice 21.1.

d) Exemple

La relation COURS étudiée à la section 3.6, et dont le schéma légèrement modifié est repris ci-après, comporte les deux identifiants $\{ETUD, MAT\}$ et $\{ETUD, PROF\}$.

$C(MAT, ETUD, PROF, CLASSE, TEL)$
 $ETUD \longrightarrow CLASSE$
 $PROF \longrightarrow TEL, MAT$
 $ETUD, MAT \longrightarrow PROF$

Sa décomposition selon la procédure simplifiée conduit aux relations $R1$ et $R2$ ainsi qu'à un noyau irréductible dont le circuit comporte une DF anormale (figure 3.17).

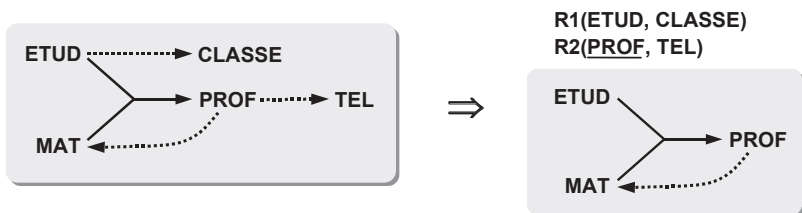


Figure 3.17 - Première décomposition conduisant à un noyau irréductible non normalisé

Ce noyau n'est donc pas normalisé : la matière d'un professeur est répétée pour chaque étudiant qui assiste au cours de ce professeur. Les trois modes de traitement du noyau irréductible produisent les solutions ci-après. La finalisation est laissée à l'initiative du lecteur.

La peste	Le choléra	La peste et le choléra
R1(ETUD, CLASSE) R2(PROF, TEL) C(MAT, ETUD, PROF) PROF $\longrightarrow$ MAT	R1(ETUD, CLASSE) R2(PROF, TEL) R3(PROF, MAT) C(ETUD, PROF) R3*C: MAT, ETUD $\longrightarrow$ PROF	R1(ETUD, CLASSE) R2(PROF, TEL) R3(PROF, MAT) C(MAT, ETUD, PROF) C[PROF, MAT] $\subseteq$ R3[PROF, MAT]

Remarque. Il peut arriver que l'application de la procédure simplifiée Procédure 3.3 laisse un noyau qui comporte encore des DF externes, lesquelles sont forcément saines sinon elles auraient été déjà traitées. Avant d'analyser et de traiter ce noyau, il convient de décomposer le schéma résiduel selon chacune de ces DF externes. On trouvera un exemple de cette situation dans l'exercice 3.6.

3.8.6 SGBD et dépendances fonctionnelles

Les SGBD ignorent les dépendances fonctionnelles, à l'exception de celles dont le déterminant est un identifiant de leur table, de sorte qu'ils sont incapables de gérer les redondances internes d'une table non normalisée (sinon via des *déclencheurs* - section 9.9). Il est donc important de n'admettre que des tables normalisées.

3.9 LES RELATIONS NON EN 1^{re} FORME NORMALE

Les relations étudiées jusqu'ici, dites en *première forme normale*, ou 1FN, sont construites sur des domaines simples. Ce modèle a très tôt été considéré par certains comme trop simpliste pour exprimer naturellement la complexité des données du monde réel. Pour décrire un formulaire, comme un bon de commande, on doit en effet désagréger ses données pour les répartir entre plusieurs relations, alors que ce formulaire est considéré par les utilisateurs comme une entité de données.

Admettons que la valeur d'un attribut puisse être un *agrégat de données*. Formellement, cet agrégat serait une relation, composée de lignes et de colonnes. Ainsi, les données des bons de commandes pourraient être regroupées comme le montre la figure 3.18. La nouvelle relation COMMANDE" possède quatre attributs. NCOM, NCLI et DATECOM sont définis sur des domaines simples. En revanche, chaque valeur de DETAIL est une *relation* possédant deux attributs, NPRO et QCOM. Appelant Detail cette relation auxiliaire servant de domaine, nous pouvons définir comme suit le schéma de la relation COMMANDE":

Detail(NPRO, QCOM)
COMMANDE"(NCOM, NCLI, DATECOM, **DETAIL: Detail***)

ou, plus simplement :

COMMANDE"(NCOM, NCLI, DATECOM, **DETAIL(NPRO,QCOM)***)

L'expression Detail* indique que le domaine de DETAIL n'est pas simplement la relation Detail, mais *l'ensemble des parties* de cette relation, ou, en termes mathématiques, *la puissance* de son extension. On dira que les attributs NCOM, NCLI et DATECOM sont *monovalués* tandis que DETAIL est *multivalué*.

COMMANDE"				
NCOM	NCLI	DATECOM	DETAIL	
			NPRO	QCOM
30178	K111	2015-12-21	CS464	25
30179	C400	2015-12-22	CS262	60
			PA60	20
30182	S127	2015-12-23	PA60	30
30184	C400	2015-12-23	CS464	120
			PA45	20
30185	F011	2016-01-02	CS464	260
			PA60	15
			PS222	600
30186	C400	2016-01-02	PA45	3
....				

Figure 3.18 - Relation N1FN représentant des bons de commandes

Cette relation peut être transformée en 1FN, comme illustré à la figure 3.19, en répétant les valeurs des attributs simples pour chaque ligne de DETAIL. Cette opération d'*aplatissement* est réalisée par un nouvel opérateur algébrique, nommé **unnest**.

COMMANDE'				
NCOM	NCLI	DATECOM	NPRO	QCOM
30178	K111	2015-12-21	CS464	25
30179	C400	2015-12-22	CS262	60
30179	C400	2015-12-22	PA60	20
30182	S127	2015-12-23	PA60	30
30184	C400	2015-12-23	CS464	120
30184	C400	2015-12-23	PA45	20
30185	F011	2016-01-02	CS464	260
30185	F011	2016-01-02	PA60	15
30185	F011	2016-01-02	PS222	600
30186	C400	2016-01-02	PA45	3
....				

Figure 3.19 - Réduction en 1FN de la relation de la figure 3.19

L'opérateur inverse, **nest**, permet de reconstruire la relation en N1FN à partir de COMMANDE'. Une ligne est construite à partir de chaque groupe de COMMANDE' ayant les mêmes valeurs de (NCOM, NCLI, DATECOM), valeurs auxquelles on associe l'ensemble des valeurs de (NPRO, QCOM).

On appliquera les opérateurs **unnest** et **nest** comme suit :

COMMANDE' = unnest(COMMANDE", DETAIL)
 COMMANDE" = nest(COMMANDE', DETAIL(NPRO,QCOM))

Une observation importante, qui justifie l'intérêt de cette forme de relation, est que COMMANDE" ne comporte **pas de redondance** alors son équivalent en 1FN, COMMANDE' comporte des répétitions. Nous y reviendrons.

Les opérateurs **nest** et **unnest** ne sont pas nécessairement réversibles. S'il est toujours vrai, quelle que soit l'extension de T(A,B,C,D), que :

$$\text{unnest}(\text{nest}(T, \text{CD}(\text{C},\text{D})), \text{CD}) = T$$

l'inverse ne l'est pas dans tous les cas, de sorte qu'on a, en toute généralité :

$$\text{nest}(\text{unnest}(\text{S}, \text{CD}), \text{CD}(\text{C},\text{D})) \neq \text{S}$$

Illustrons ce phénomène par l'exemple de la figure 3.20. La relation **S** dont l'attribut CD est défini sur l'ensemble des parties d'une relation, comporte trois lignes distinctes. L'application de l'opérateur **unnest** à l'attribut **CD** produit la relation **T**. Durant l'opération, deux lignes (MERCIER, 2016-01-03, PA60, 35) ont été générées, dont un seul exemplaire a été retenu dans **T**. Appliqué à cette dernière, l'opérateur **nest** regroupe les lignes de mêmes valeurs de (A, B) et leur associe les valeurs de (C, D) pour former une ligne de **S'**. On observe ainsi le regroupement des lignes selon les valeurs (MERCIER, 2016-01-03) formant une seule ligne de **S'**, alors qu'elles étaient réparties en deux lignes dans la relation **S** d'origine. Appliqué à **S**, l'opérateur **unnest** n'est donc pas réversible.

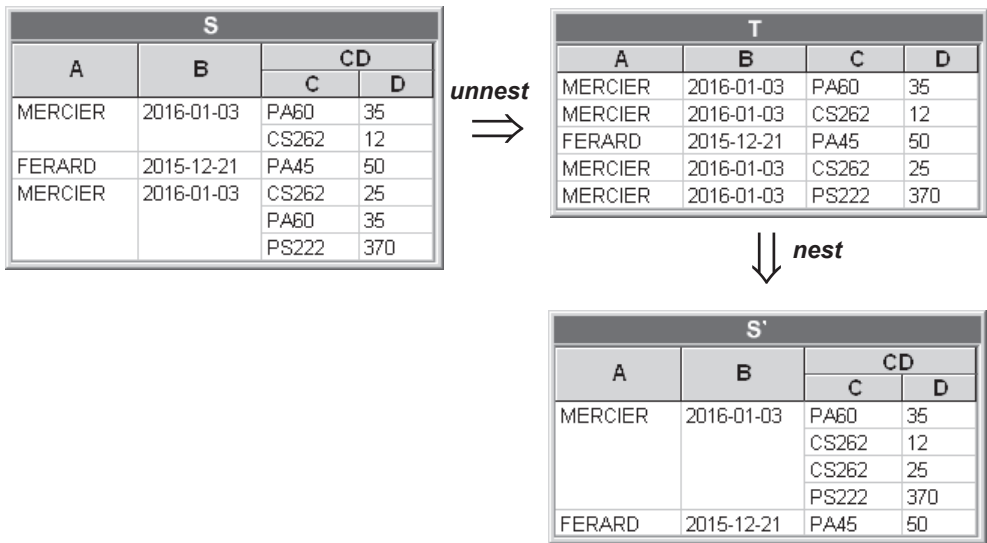


Figure 3.20 - Cas de non réversibilité de l'opérateur unnest

La propriété de réversibilité de l'opérateur **unnest** est liée à la présence d'un identifiant dans la relation en N1FN. Si on se reporte à notre base de données de référence (Figure 2.8), dont la relation de la figure 3.18 dérive, on observe que l'identifiant de COMMANDE" est l'attribut monovalué NCOM. En revanche, la relation **S** ne possède pas d'identifiant exclusivement constitué d'attributs monovalués.

Il existe donc des *formes acceptables* de relations en N1FN, celles qui supportent sans perte d'information l'opérateur unnest. On dira que ces relations sont en **forme normale partitionnée**, définie comme suit :

*Une relation S est en **forme normale partitionnée** ssi (1) S possède au moins un attribut monovalué, (2) S possède au moins un identifiant, éventuellement non minimal, formé de l'ensemble de ses attributs monovalués et (3) toute relation utilisée comme domaine d'un attribut est également en forme normale partitionnée.*

Notes

1. La question de l'*ensemble vide* comme valeur de l'attribut **CD** de **S** introduit la notion de *valeur absente* (ou valeur *null* en SQL) pour les attributs **C** et **D** de **T**. Ce concept, qui a été écarté dans le modèle de base, sera abordé lors de l'étude du langage SQL. Nous n'en dirons pas plus pour l'instant.
2. L'opérateur unnest introduit dans la relation résultante un nouveau type de dépendance, la *dépendance multivaluée*, généralisation de la dépendance fonctionnelle dans laquelle le déterminé n'est plus une valeur élémentaire, mais un ensemble de valeurs. Étant le peu d'intérêt pratique de ce concept, nous n'en dirons pas plus dans cet ouvrage mais il sera étudié dans l'annexe A03.
3. La forme normale partitionnée est indépendante de la *forme normale de Boyce-Codd* étudiée dans ce chapitre, basée sur les dépendances fonctionnelles. Elle aura cependant un lien avec les dépendances multivaluées.

Si le modèle relationnel N1NF est plus expressif que le modèle de base, il entraîne cependant une plus grande complexité à la fois théorique et pratique. On décrira à la section 9.6.1 l'implémentation (approximative) en SQL des attributs complexes. L'extension du modèle relationnel aux attributs complexes a été étudiée dès les années quatre-vingt [Scheck,1986]. Plus tard, de nombreuses publications ont appliqué ce modèle et ses opérateurs aux structures de données XML et JSON.

3.10 QUE RETENIR ?

Selon la théorie relationnelle, une base de données est constituée de **domaines** et de **relations**, formées d'**attributs**, chacun tirant ses valeurs d'un domaine. Des opérateurs permettent de dériver des données à partir de celles d'un ensemble de relations. La **projection** d'une relation T est un opérateur qui produit une relation qui reprend un sous-ensemble des attributs de T. La **jointure** de deux relations T1 et T2 produit une relation dont les lignes sont constituées de l'assemblage de deux lignes extraites chacune de T1 et T2. Ces deux lignes sont assemblées dès qu'elles partagent les mêmes valeurs d'attributs déterminés (les attributs de jointure). La **sélection** produit le sous-ensemble des lignes qui satisfont une condition sur les valeurs d'attribut.

L'**identifiant** d'une relation est un sous-ensemble de ses attributs dont les valeurs sont uniques pour les lignes de cette relation. On s'intéresse en général aux *identifiants minimaux*, c'est-à-dire ceux dont tous les composants sont nécessaires pour garantir l'unicité des valeurs.

Une **dépendance fonctionnelle** (DF) $K \longrightarrow L$ existe dans une relation entre un ensemble K d'attributs (le **déterminant**) et un ensemble L d'attributs (le **déterminé**) si, dans tout couple de lignes qui ont mêmes valeurs de K , les valeurs de L sont aussi les mêmes. Une dépendance fonctionnelle exprime une propriété du domaine d'application. Une DF est **élémentaire** si son déterminé L comporte un seul attribut. Une DF est **minimale** s'il n'existe pas de DF entre une partie stricte de son déterminant et son déterminé.

Un jeu de **règles d'inférence**, dites d'Armstrong, permet de dériver des DF à partir d'un ensemble initial de DF. Ce jeu inclut notamment la règle de **transitivité** selon laquelle la composition de deux DF produit une troisième DF. La **fermeture** d'un ensemble F de DF est l'ensemble (unique) de toutes les DF qu'il est possible de générer à partir de F , en appliquant les règles d'Armstrong. Une DF est **dérivée** si elle peut être reconstruite à l'aide des règles d'Armstrong. Sinon, elle est dite **de base**. Une **couverture minimale** d'un ensemble F de DF est un ensemble de DF de même fermeture que F , mais dont aucun sous-ensemble ne jouit de cette propriété. La **fermeture** d'un ensemble A d'attributs est l'ensemble de tous les attributs dépendant fonctionnellement de A .

La représentation graphique des attributs et des DF constitue le **graphe ADF** de la relation. Si celui-ci comporte un **circuit**, alors la relation possède plusieurs identifiants.

Une **contrainte d'inclusion** précise que les valeurs d'un ou plusieurs attributs présentes dans les lignes d'une relation doivent se retrouver parmi les valeurs des attributs correspondants dans les lignes d'une autre relation. La **clé étrangère** induit une contrainte d'inclusion particulière dans laquelle les attributs forment un identifiant de la seconde relation.

L'identifiant **se calcule** par élimination des attributs déterminés mais non déterminants. En cas de circuit de DF, les différents identifiants sont obtenus suite à la rupture de ce circuit.

Il est possible de **décomposer** une relation en plusieurs fragments sans perte de données pourvu qu'on applique cette décomposition selon une DF. Dans ce cas, l'ensemble des lignes de la relation d'origine est égale à la jointure de ses deux projections respectivement sur les attributs de la DF d'une part, et sur le déterminant et les attributs extérieurs à la DF d'autre part.

Des **redondances internes** peuvent apparaître dans une relation lorsqu'elle est le siège d'une dépendance fonctionnelle dont le déterminant n'est pas un identifiant. Une telle DF est dite **anormale**. On élimine ces redondances en décomposant la relation selon les DF anormales. Ce processus est appelé **normalisation**.

Lorsque le procédé de décomposition ne peut plus être poursuivi malgré la présence de DF anormales, la relation résiduelle constitue le **noyau irréductible**. Il existe **trois formes** finales de ce noyau, dont aucune ne préserve à la fois les données et les DF tout en éliminant des redondances.

Dans la forme **N1FN** de relation, on admet qu'une valeur d'attribut soit elle-même une relation. Cette forme peut être convertie en relation simple par l'opérateur **unnest**. À l'inverse, l'opérateur **nest** transforme une relation simple sous la forme

N1FN. L'opérateur **unnest** est réversible si la relation en N1FN est en **forme normale partitionnée**, c'est-à-dire si elle possède un identifiant exclusivement constitué d'attributs monovalués.

3.11 POUR EN SAVOIR PLUS

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 en avons présenté dans ce chapitre. En particulier, il existe d'autres types de dépendances (dépendances fonctionnelles, multivaluées, de jointure) et d'autres formes normales (notamment 1FN, 2FN, 3FN, FNCE, FNBC, 4FN, 5FN). Celle que nous avons présentée dans la section 3.8.3, qui veut que *tout déterminant soit un identifiant*, est dénommée *forme normale de Boyce-Codd* (FNBC).

Le résultat de la procédure dénommée *La peste* correspond à la *troisième forme normale* (3FN) tandis que celle qui a été qualifiée de *Choléra* est en *forme normale de Boyce-Codd* (FNBC). La troisième solution, *La peste et le choléra* correspond à une variante assez peu connue, dénommée *forme normale à clé élémentaire* ou FNCE (*Elementary Key Normal Form* ou EKNF) [Zaniolo, 1982].

Il existe un autre procédé de normalisation, dit *par synthèse*. En apparence plus simple que le procédé par décomposition, il consiste à former une relation à partir des attributs de chaque DF, puis à regrouper les relations de même identifiant. On ajoute en outre une relation pour chaque identifiant dont tous les attributs ne sont pas repris dans une même relation. Ce procédé semble plus intuitif mais s'avère moins puissant que la décomposition : il conduit à la *FNCE* seulement et il ne prend en compte que les DF.

Le lecteur est renvoyé à l'importante littérature couvrant les différents aspects théoriques et pratiques des bases de données relationnelles. Il consultera, lorsqu'elles sont disponibles, les références historiques [Delobel, 1982], [Maier, 1983], [Ullman, 1988], [Atzeni, 1993] ainsi que les références plus récentes telles que [Bouzeghoub, 1998], [Abiteboul, 2000], [Gardarin, 2003] ou [Date, 2004].

3.12 EXERCICES

3.1 Décomposer si nécessaire la relation ACHAT.

ACHAT(NCOM, NPRO, PRIX)
 NCOM $\longrightarrow$ NPRO
 NPRO $\longrightarrow$ PRIX

3.2 Décomposer si nécessaire la relation COMMANDE.

COMMANDE(NCOM, NCLI, NOM, DATE, NPRO, LIBELLE)
 NCOM $\longrightarrow$ NCLI, DATE, NPRO
 NCLI $\longrightarrow$ NOM
 NPRO $\longrightarrow$ LIBELLE

- 3.3 Décomposer si nécessaire la relation ACHAT2.

ACHAT2(CLI, PRO, MAG, PRIX)
 PRO, MAG $\longrightarrow$ PRIX

- 3.4 Décomposer si nécessaire la relation ACHAT3.

ACHAT3(CLI, PRO, MAG, PRIX)
 CLI, PRO, MAG $\longrightarrow$ PRIX

- 3.5 Décomposer si nécessaire la relation ECRIT (POSITION indique le rang de l'auteur dans la liste des auteurs).

ECRIT(AUTEUR, OUVRAGE, POSITION)
 AUTEUR, OUVRAGE $\longrightarrow$ POSITION
 OUVRAGE, POSITION $\longrightarrow$ AUTEUR

- 3.6 Calculer les identifiants de la relation CINE. Décomposer cette relation si nécessaire.

CINE(FILM, VILLE, SALLE, DISTRIBUTEUR, DELEGUE)
 SALLE $\longrightarrow$ VILLE
 FILM, VILLE $\longrightarrow$ SALLE, DISTRIBUTEUR
 DISTRIBUTEUR $\longrightarrow$ DELEGUE

- 3.7 Démontrer la règle de *pseudo-transitivité*:

$K \longrightarrow L$ et $LM \longrightarrow N \Rightarrow KM \longrightarrow N$

Exercices corrigés

- **Exercice 3.1.** L'identifiant de ACHAT est {NCOM}. La DF NPRO $\longrightarrow$ PRIX est donc anormale. Par décomposition selon cette DF, on obtient le schéma relationnel normalisé :

ACHAT(NCOM, NPRO); PRODUIT(NPRO, PRIX)
 ACHAT[NPRO] $\subseteq$ PRODUIT[NPRO]

- **Exercice 3.2.** L'identifiant de COMMANDE est {NCOM}. Les DF NCLI $\longrightarrow$ NOM et NPRO $\longrightarrow$ LIBELLE sont donc anormales. Par décomposition selon chacune de ces DF, on obtient le schéma relationnel normalisé :

COMMANDE(NCOM, NCLI, DATE, NPRO)
 CLIENT(NCLI, NOM); PRODUIT(NPRO, LIBELLE)
 COMMANDE[NCLI] $\subseteq$ CLIENT[NCLI]
 COMMANDE[NPRO] $\subseteq$ PRODUIT[NPRO]

- **Exercice 3.3.** L'identifiant de ACHAT2 est {CLI, PRO, MAG}. La DF PRO, MAG $\longrightarrow$ PRIX est donc anormale. On obtient par décomposition :

ACHAT2(CLI, PRO, MAG); TARIF(PRO, MAG, PRIX)
 ACHAT2[PRO, MAG] $\subseteq$ TARIF[PRO, MAG]

- **Exercice 3.4.** L'identifiant de la relation ACHAT3 est {CLI, PRO, MAG}. Celle-ci est donc normalisée.

- **Exercice 3.5.** Le graphe ADF comporte un circuit. Les identifiants de la relation ECRIT sont {AUTEUR, OUVRAGE} et {OUVRAGE, POSITION}. Celle-ci est normalisée.
- **Exercice 3.6.** Le graphe ADF comporte un circuit. Les identifiants sont {FILM, VILLE} et {SALLE, FILM}. Les deux DF suivantes sont donc anormales :
 $SALLE \longrightarrow VILLE$ et $DISTRIBUTEUR \longrightarrow DELEGUE$. Cette dernière étant externe, elle permet une première décomposition :

$CINE(FILM, VILLE, SALLE, DISTRIBUTEUR)$
 $DIS(DISTRIBUTEUR, DELEGUE)$
 $CINE[DISTRIBUTEUR] \subseteq DIS[DISTRIBUTEUR]$
 $SALLE \longrightarrow VILLE$
 $FILM, VILLE \longrightarrow DISTRIBUTEUR$

La DF $FILM, VILLE \longrightarrow DISTRIBUTEUR$, bien que non anormale, est externe et ne fait pas partie du noyau irréductible. Elle peut donc faire l'objet d'une décomposition :

$CINE(FILM, VILLE, SALLE)$
 $DISTR(FILM, VILLE, DISTRIBUTEUR)$
 $DIS\_DEL(DISTRIBUTEUR, DELEGUE)$
 $CINE[FILM, VILLE] \subseteq DISTR[FILM, VILLE]$
 $DISTR[DISTRIBUTEUR] \subseteq DIS\_DEL[DISTRIBUTEUR]$
 $SALLE \longrightarrow VILLE$

Le noyau résiduel {FILM, VILLE, SALLE} est irréductible et non normalisé. Selon le canevas 3.8.5, la dernière relation CINE peut être remplacée par un des trois schémas suivants :

1. $CINE(FILM, VILLE, SALLE); SALLE \longrightarrow VILLE$
 2. $CINE(FILM, SALLE); LOC(SALLE, VILLE)$
 $CINE[SALLE] = LOC[SALLE]$
 $CINE*LOC: FILM, VILLE \longrightarrow SALLE$
 3. $CINE(FILM, VILLE, SALLE); LOC(SALLE, VILLE)$
 $CINE[SALLE, VILLE] = LOC[SALLE, VILLE]$
- **Exercice 3.7.** Par réflexivité, on a $M \longrightarrow M$. Par additivité, $K \longrightarrow L$ et $M \longrightarrow M$ donnent $KM \longrightarrow LM$. Par transitivité, $KM \longrightarrow LM$ et $LM \longrightarrow N$ donnent $KM \longrightarrow N$. On peut aussi utiliser le concept de fermeture d'attributs. Calculons la fermeture de KM et vérifions que N en fait partie.
 On part de $KM \longrightarrow KM$. KM constitue le point de départ de la fermeture.
 Par hypothèse, $K \longrightarrow L$. Comme K est dans KM , on étend KM à KLM .
 Par hypothèse, $LM \longrightarrow N$. Comme LM est dans KLM , on étend KLM à $KLMN$.
 Comme il n'est plus possible d'étendre $KLMN$, $KM^+ = KLMN$. KM est donc bien un déterminant de N .

Des exercices et solutions additionnels ainsi que des compléments sont disponibles à l'annexe A3.

Chapitre 4

Éléments de technologie des bases de données

Ce chapitre décrit les grandes techniques d'implémentation des structures des bases de données relationnelles. On y présente les caractéristiques des disques magnétiques, le mode d'implantation des espaces de stockage et des lignes des tables. L'utilisation des tampons et de la lecture anticipée est décrite. On présente ensuite les techniques d'indexation des données ainsi que les règles d'évaluation des temps d'accès et des volumes occupés. Parmi ces techniques, on détaille plus particulièrement les organisations séquentielle indexée et calculée ainsi que les index secondaires. Les techniques d'agrégation sont décrites et évaluées. Cette matière, qui fait partie du **parcours de lecture complet**, suppose acquises des notions de base en SQL.

4.1 INTRODUCTION

Les structures de données sous forme de tables (chapitre 2) ou d'objets mathématique (chapitre 3) donnent une vue intuitive mais abstraite des données, vue dont le lien avec les ordinateurs tels qu'on les connaît dans la vie de tous les jours n'est pas évident. Nous savons que ces données sont stockées dans (ou plutôt **sur**) un disque d'un ordinateur, le nôtre ou un serveur, mais nous ne nous sommes pas intéressés jusqu'ici à *la manière* dont les données sont organisées sur ce disque. Or, si la connaissance de ces détails techniques n'est pas nécessaire pour utiliser une base de données, elle sera en revanche très utile au moment où nous entreprendrons de construire une base de données. Nous allons tenter de répondre à quelques questions

fondamentales : où et comment écrire une donnée (une ligne de table par exemple) sur le disque, comment retrouver une donnée sur un disque, comment écrire et lire une donnée de manière performante (et d'ailleurs que signifie *performant*) ?

Nous décrivons d'abord les composants d'un ordinateur (4.2) puis le fonctionnement et les caractéristiques des principaux supports de données, les disques magnétiques (4.3). Nous étudierons ensuite quelques techniques de rangement des lignes de tables dans un disque (4.5) puis nous décrivons les techniques d'implémentation des index, primaires (4.8 et 4.9) et secondaires (4.10), qui constituent les moyens les plus utiles pour améliorer les temps d'écriture et de lecture des données.

4.2 LES COMPOSANTS D'UN ORDINATEUR

Un ordinateur typique est formé de composants fonctionnels qui se transmettent des commandes et des données via un *bus d'échange* (figure 4.1). On distingue trois types de composants : le *processeur*, chargé d'exécuter les instructions des programmes, la *mémoire centrale*, contenant les programmes en cours d'exécution et leurs données locales, et les *contrôleurs* chargés chacun de piloter un type d'appareils externes. Ces composants constituent ce qu'on appelle l'*unité centrale*¹.

Les programmes que contient la mémoire centrale auront au préalable été chargés à partir d'un disque ou de l'internet. La figure montre trois de ces programmes : le *système d'exploitation* (SE), qui assure les services de base tels que la gestion des entrées/sorties, des processus, de l'espace de la mémoire centrale ou des fichiers, le *système de gestion de bases de données* (SGBD) qui gère les bases de données connectées et un *programme d'application* (PA), assurant par exemple la gestion des achats en ligne des clients. Le SGBD requiert les services du système d'application, par exemple pour lire des données sur le disque.

Lorsque le programme PA exécute une requête SQL, il la transmet au SGBD. celui-ci fait appel au système d'exploitation SE pour lire un bloc de données sur disque. Le SE utilise son service de gestion de fichiers, lequel invoque le contrôleur de disques en spécifiant une adresse sur un disque. Le contrôleur traduit cette demande en courants électriques qui actionnent les moteurs du disque pour lire le bloc de données². Ces données sont retournée au SE, qui les range dans l'espace local du SGBD (ce qu'on appellera le tampon dans la suite). Là, le SGBD analyse ce bloc, en extrait les données demandées par PA et les transmet à ce dernier.

1. Ce terme recouvre les composants électroniques formant ou directement connectés à la *carte-mère* de l'ordinateur. Les disques fixes (non amovibles) n'en font donc pas partie. Selon une autre acception, l'unité centrale est formée des composants que contient le boîtier d'un ordinateur de bureau ou d'un serveur. On y compte donc les disques fixes et le lecteur/graveur de DVD.

2. Il existe en réalité deux contrôleurs. Le premier se trouve dans l'unité centrale. Il gère le protocole standard d'interaction avec les disques (IDE, SCSI, SATA) et communique avec la mémoire centrale. Le second est incorporé au disque et assure des fonctions telles que la correction d'erreurs, la linéarisation des adresses, l'ordonnancement des accès, la conversion analogique/digitale, la détection des chocs, le contrôle des moteurs. Il possède aussi une mémoire tampon qui peut contenir une copie des données de plusieurs dizaines de pistes.

Une étude plus détaillée des relations entre PA, SGBD, SE et les disques sera proposée à la section 5.3.

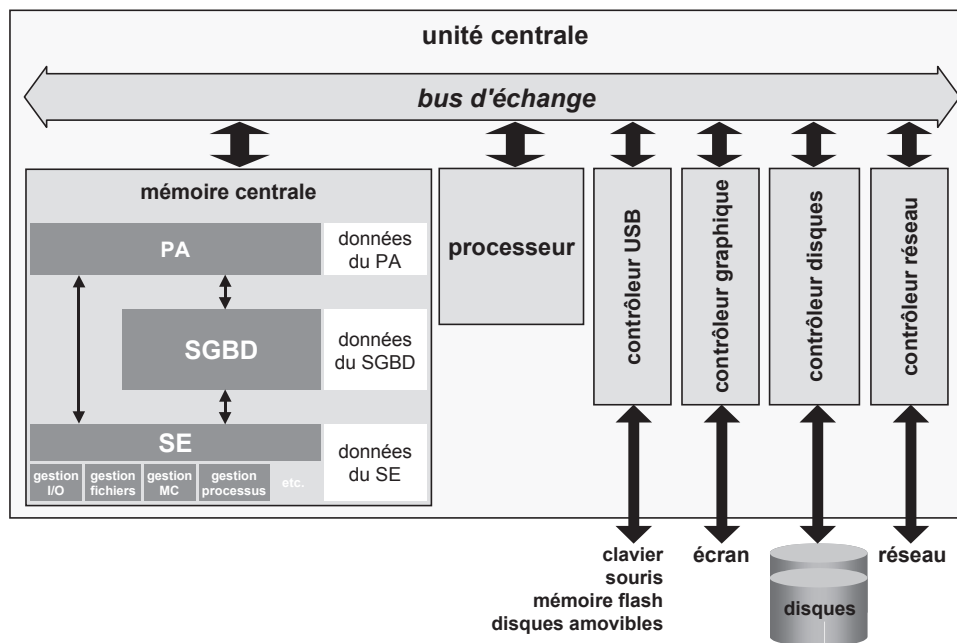


Figure 4.1 - Architecture générale d'un ordinateur

4.3 MÉMOIRE EXTERNE : LE DISQUE MAGNÉTIQUE

Nous nous intéressons principalement aux fichiers à accès direct, c'est-à-dire ceux dont l'organisation permet de lire et d'écrire un enregistrement dont on connaît l'adresse dans ce fichier sans qu'il soit nécessaire de lire au préalable les enregistrements qui le précèdent. Un tel fichier suppose bien sûr que la mémoire dans laquelle il se trouve soit adressable. Une mémoire est dite *adressable* lorsqu'elle est constituée de cellules, généralement de même taille, possédant chacune une adresse unique. Les mémoires à disque magnétique, que nous appellerons plus simplement disques (ou disques durs), constituent la technologie la plus répandue actuellement.

4.3.1 Géométrie d'un disque

Sur le plan mécanique, un disque apparaît comme une machine assez simple : un empilement de plateaux circulaires tournant à grande vitesse et une série de bras supportant des têtes de lecture/écriture magnétiques qui écrivent et lisent des données à la surface de ces plateaux. On y ajoute un moteur qui entraîne les plateaux et un autre qui déplace les bras ainsi qu'un peu d'électronique et de la filerie (figure 4.2). L'espace d'un disque est décomposé en **secteurs** de 512 octets (4 096 pour certains modèles récents) dont les coordonnées (leur adresse) sont bien définies et

sur lesquels les têtes magnétiques peuvent se positionner. L'ordinateur peut alors demander le transfert du contenu du secteur (ce qu'on appelle *lire un secteur*) et, à l'inverse, demander l'écriture d'un bloc de données de 512 octets dans le secteur dont on spécifie l'adresse. Néanmoins, son fonctionnement réel et le mode d'adressage des données, que nous allons étudier, sont un peu plus complexes³.

a) Structure physique d'un disque

Le support est formé d'un assemblage de 1 à 10 plateaux circulaires solidaires tournant à grande vitesse (de 3 600 à 15 000 révolutions par minute ou *rpm*).

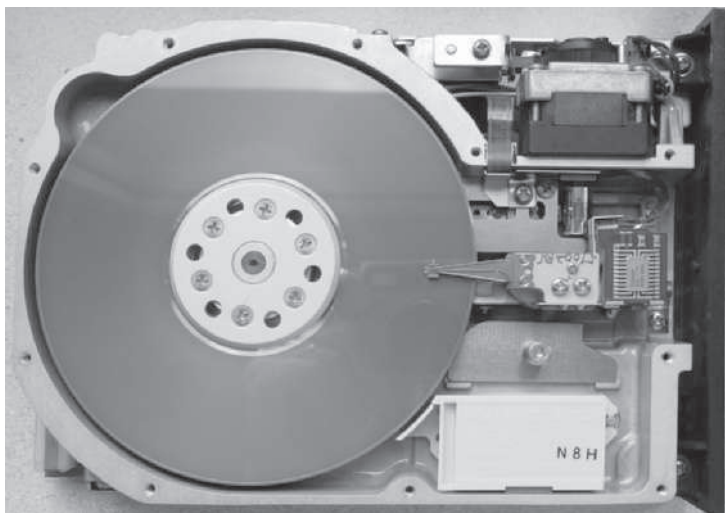


Figure 4.2 - Écorché d'un disque magnétique⁴ à déplacement linéaire du bras (1985)

Chacune des deux **faces** de chaque plateau est recouverte d'une substance magnétisable similaire à celle qu'on trouve dans une ancienne cassette audio ou vidéo. Une tête de lecture/écriture magnétique, fixée au bout d'un bras, flotte à faible distance de chaque face. La zone annulaire de la face qui défile sous la tête lorsque celle-ci occupe une position définie s'appelle une **piste**. Un secteur est une portion d'une piste, qui en contient de 200 à 2 000. Tous les bras sont solidaires et se déplacent en même temps. Ils forment ce qu'on appelle familièrement le *peigne*. Celui-ci est mû par un moteur qui permet de placer les têtes dans une des 50 000 à 300 000 positions prédéfinies sur les faces, définissant ainsi autant de pistes concentriques sur chaque face. Pendant la rotation du plateau, chaque tête peut écrire ou lire dans la piste, plus précisément dans un de ses secteurs, des signaux binaires sous forme magnétique.

3. Pour plus de détail sur le fonctionnement physique des disques magnétiques, consulter par exemple http://fr.wikipedia.org/wiki/Disque_dur et <https://www.smarthdd.com/en/help.htm> (juin 2018). Lire aussi [Deng, 2011] pour une description technique détaillée.

4. NEC D5126. Capacité de 20 Mb, 2 plateaux de 5,25", 612 cylindres, 17 secteurs par piste, temps moyen de déplacement du bras de 80 ms. A équipé les PC d'IBM des premières années. Des photos de disques anciens et récents sont disponibles sur le site de l'ouvrage.

Les pistes de même numéro constituent un **cylindre** virtuel dont le numéro est celui de ses pistes dans les faces.

b) Capacité d'un disque

On considère qu'un disque est décomposé en *cylindres*, formés de *pistes* (autant que de faces et de têtes), constituées de *secteurs* de $L_{\text{sec}} = 512$ octets. Un disque de 500 Go, que nous adopterons comme unité de référence (figure 4.6), est constitué de $N_{\text{cpd}} = 136\,000$ cylindres de $N_{\text{ppc}} = 8$ pistes (4 plateaux) de $N_{\text{spp}} = 900$ secteurs. Il a une capacité de $L_{\text{pis}} = 460\,800$ octets par piste, $L_{\text{cyl}} = 3\,686\,400$ octets par cylindre et $N_{\text{spd}} = 979\,200\,000$ secteurs, soit $L_{\text{dsq}} = 501\,350\,400\,000$ octets au total⁵. Les disques récents (2018) atteignent des capacités de 10 Go en format 3.5" et de 3 Go en 2.5".

c) Adressage d'un secteur

L'adresse *physique* ou *géométrique* d'un secteur, dite *Cylinder-Head-Sector* ou CHS, est le triplet $\langle c, p, s \rangle$, où c est le numéro du cylindre (par exemple de 0 à 135 999), p le numéro de la piste (ou de la tête) dans le cylindre (de 0 à 7) et s le numéro du secteur dans la piste (de 1 [et, curieusement, pas 0!] à 900) (figure 4.3). Dans les faits, on utilise plutôt un adressage logique linéarisé sous la forme d'un numéro de secteur ns (de 0 à 979 199 999), ce qu'on appelle *Logical Block Addressing* ou LBA. Le secteur $\langle c, p, s \rangle = \langle 63\,450, 2, 307 \rangle$ est à l'adresse LBA $ns = (63\,450 \times 8 + 2) \times 900 + (307 - 1) = 456\,842\,106$. La conversion entre l'adresse linéaire ns et l'adresse physique $\langle c, p, s \rangle$ est réalisée par le contrôleur du disque.⁶

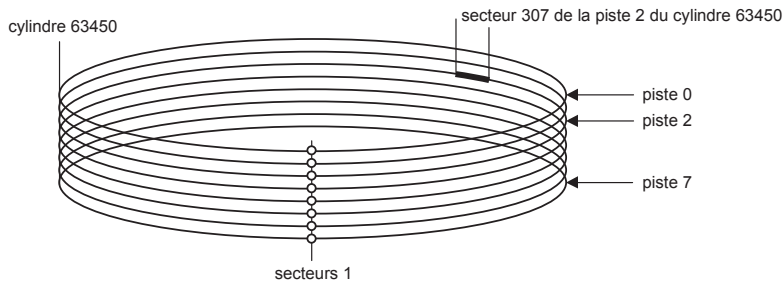


Figure 4.3 - Repérage du secteur $\langle c, p, s \rangle = \langle 63450, 2, 307 \rangle$

4.3.2 Lecture et écriture de données sur un disque

La **lecture d'un secteur** quelconque, ou plus généralement de n secteurs contigus, consiste à transférer en mémoire centrale, à une adresse spécifiée, le contenu d'un

5. Dans le langage informatique habituel, dans lequel kilo- = 1 024, méga- = 1 048 576 et giga- = 1 073 741 824, ce disque aurait une capacité réelle de 478 125 Mo ou 467 Go. Les producteurs de mémoires externes utilisent en effet les unités décimales classiques (kilo = 1 000, méga = 1 000 000, etc.) pour mesurer la capacité de leurs produits.

6. Les disques modernes contiennent plus de secteurs dans les pistes externes que dans les pistes internes. Ils sont alors dits *zonés*. Cette hétérogénéité est masquée par l'adressage LBA.

secteur dont on donne l'adresse physique $\langle c, p, s \rangle$ (ou logique ns) et de celui des $n-1$ secteurs suivants. Cette lecture s'effectue en quatre étapes :

1. Les bras se déplacent vers le cylindre c ; ce processus est coûteux en temps car il consiste en un déplacement mécanique de l'ensemble des bras ; on distingue trois valeurs : le temps minimum, $seek_1$, mesurant le temps de déplacement d'un cylindre au suivant, le temps moyen⁷, $seek_m$ et le temps maximum, $seek_M$, du déplacement du premier cylindre au dernier ; valeurs typiques, en ms : (1, 8, 15) ; la valeur qui nous intéresse, $seek_m$, va de 2,5 à 15 ms.
2. La tête p est commutée ; une opération électronique de durée négligeable.
3. La tête attend que le secteur s se présente sous son entrefer, ce qui correspond en moyenne à une demi-rotation, soit $8,33/2 = 4,17$ ms pour un disque à 7 200 rpm ; ce temps est appelé *délai rotationnel* (*latency time*).
4. Le contenu du secteur et des $n - 1$ suivants est lu et transmis vers la mémoire centrale ; pour un disque à 7 200 rpm, dont les pistes comportent 900 secteurs et $n = 8$, le temps du transfert ttr est de $(8,33 / 900) \times 8 = 0,074$ ms.

Pour notre disque de référence, le temps de lecture de 8 secteurs d'adresse initiale aléatoire est de $8 + 0 + 4,17 + 0,074 \cong 12,3$ ms. La lecture d'une piste entière d'adresse aléatoire consiste en un déplacement des bras suivi immédiatement du transfert des secteurs. Il n'y a pas de délai rotationnel (le premier secteur qui se présente sous la tête est à transférer), de sorte que le temps de lecture est de $8 + 8,33 = 16,33$ ms pour 900 secteurs.

En réalité, le secteur est une unité trop petite pour les besoins de transfert dans les SGBD. On adoptera donc une quantité unitaire de transfert plus importante, dénommée bloc ou **page**, constituée typiquement de 4, 8 ou 16 secteurs (figure 4.4). Les calculs précédents nous fournissent une grandeur importante : le temps de lecture d'une page de 4 Ko d'adresse aléatoire : **tia1 = 12,3 ms**.

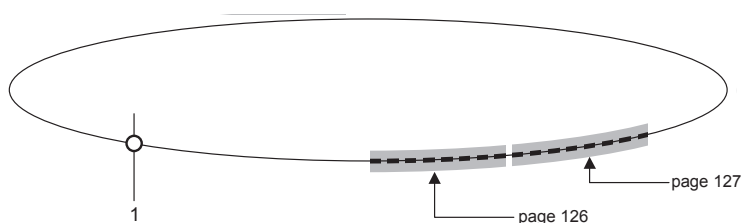


Figure 4.4 - Deux pages consécutives de 8 secteurs

Dans le même ordre d'idée, on sera souvent amené à lire une grande quantité de données, couvrant plusieurs pages consécutives dans une ou plusieurs pistes. Considérons par exemple une demande de lecture concernant 100 pages de 4 Ko, couvrant 800 secteurs. Selon la position de la première page, ces pages occupent une ou deux

7. Cette valeur, appelée *average seek time*, est fournie par le fabricant. Elle correspond au temps moyen nécessaire pour déplacer les têtes de *n'importe quel cylindre* vers *n'importe quel cylindre*, tous les cylindres du disque étant équiprobables comme points de départ et d'arrivée.

pistes, qu'on considérera pour simplifier dans le même cylindre. La figure 4.5 montre les deux cas de figure, la première page débutant au secteur 49 (à gauche) ou au secteur 199 (à droite). Le second exige un changement de piste, lequel n'entraîne aucune pénalité, puisque le temps de commutation d'une tête est négligeable. Le temps de lecture sera donc le même dans ces deux cas.

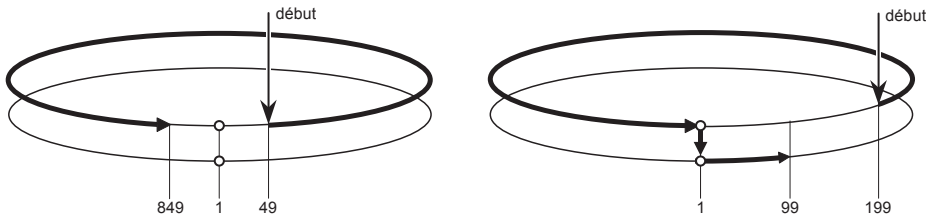


Figure 4.5 - Lecture de 100 pages de 4 Ko selon la position du premier secteur

Le **temps d'écriture** est en principe identique à celui de la lecture. Cependant, il est possible de demander une *écriture avec vérification*, qui consiste à faire suivre l'écriture d'une relecture des secteurs écrits afin de vérifier que les données ont été correctement enregistrées, soit une rotation complète. Le temps d'écriture avec vérification comporte donc un supplément de 8,33 ms.

La modification d'une donnée élémentaire (même un simple octet) sur le disque nécessite une lecture de sa page en mémoire centrale, la modification de son contenu puis la recopie de la page entière sur le disque, pour un total de 24,6 ms.

4.3.3 Optimisation : tampon d'échange et lecture anticipée

Le temps nécessaire au transfert d'un bloc de 4 Ko en mémoire centrale dépend de la source des données. Si le bloc est sur un disque, ce temps est en moyenne de 12,3 ms. S'il est déjà présent ailleurs en mémoire centrale, son transfert pourra se faire (à 9,6 Go/s) en moins de 0,00043 ms, soit 28 600 fois moins⁸. *Conclusion naïve* : il serait plus avantageux que la base de données réside en mémoire centrale ! Cette approche conduit au concept de *In-memory DB*, selon lequel la base de données (de petite taille) est chargée en mémoire centrale. Nous en reparlerons plus loin. Comme cette proposition est le plus souvent irréalisable, on adopte un compromis : on conserve en mémoire centrale les pages le plus souvent utilisées. L'espace en mémoire centrale réservé à ces pages est appelé (mémoire) **tampon**⁹ ou (mémoire) **cache**. Le tampon est décomposé en cadres dont chacun peut accueillir la copie d'une page de la base de données. Il est géré comme suit.

a) Lecture de données

Lors d'une demande de lecture (d'une ligne de table par exemple) de la part d'un programme d'application, le SGBD identifie la page concernée et recherche dans le

8. Le gain est plus important encore, le transfert concernant, comme nous allons le voir, un enregistrement et non une page entière.

9. En anglais *buffer* ou *cache*.

tampon s'il en existe une copie. Si tel est le cas, les données sont extraites du contenu du cadre et transmises au programme. Nous avons vu que le coût de cette lecture était négligeable. Si la page n'est pas dans le tampon, le SGBD en demande le transfert dans un cadre libre à partir du disque, puis procède comme précédemment. Que faire lorsque tous les cadres sont occupés ? Le SGBD repère un cadre dont le contenu n'a *plus été utilisé depuis le plus longtemps* et y transfère une copie de la nouvelle page¹⁰. Si ce contenu a été modifié depuis son chargement il est au préalable recopié sur le disque. Le coût est alors celui d'une écriture + une lecture.

b) Écriture de données

Lorsque le programme d'application demande une modification de données (création, modification ou suppression d'une ligne par exemple), le SGBD identifie la page concernée (ou *les pages* dans certains cas plus complexes que nous étudierons plus loin) et en recherche une copie dans le tampon. Si cette page est absente du tampon, le SGBD la charge au préalable comme pour une opération de lecture (coût : une lecture de page). Celle-ci est modifiée mais *n'est pas recopiée sur le disque*. En effet, il est fréquent qu'une mise à jour des données d'une page soit immédiatement suivie d'autres demandes relatives à cette même page. Bien sûr, il faudra à un certain moment recopier la page sur disque ! Quand ? Le plus tard possible, afin de répartir le coût de réécriture sur disque sur plusieurs mises à jour. Dans les faits, la recopie sur le disque d'une page modifiée est effectuée notamment dans les circonstances suivantes : lorsque le cadre de la page est sélectionné pour accueillir une nouvelle page lue, lors de la déconnexion du programme, lors de la fermeture de la base de données ou du serveur, à la fin de la transaction qui a provoqué la modification des données (voir 5.4). On peut également attendre que les pages modifiées consécutives atteignent la longueur d'une piste, auquel cas elles sont recopiées en un tour de piste. Ce comportement est appelé *écriture retardée*.

Le tampon joue un rôle très important dans l'optimisation des accès au disque. Sa gestion est en réalité un peu plus complexe que ce qu'on en a présenté. En particulier, certaines pages peuvent être maintenues dans le tampon sans risque d'être éjectées en cas de besoin. Tel sera le cas des pages comprenant certains index ou des métadonnées (tables du catalogue par exemple). Un gros serveur peut allouer plusieurs centaines, voire plusieurs milliers de giga-octets au tampon de ses bases de données.

c) Lecture anticipée

Il existe un autre *pattern* de lecture très fréquent : la lecture de pages consécutives sur disque. L'exemple le plus évident est celui de la lecture de toutes les lignes d'une table, dite *lecture séquentielle*. Constatant un tel comportement de la part d'un programme application, le SGBD peut décider d'anticiper les demandes d'accès et de charger dans le tampon une suite plus ou moins longue de pages dans l'espoir que le programme va en exploiter le contenu. C'est ce qu'on appelle la lecture anticipée¹¹. *Premier avantage* : les pages seront déjà dans le tampon lorsque le

10. Cette politique de gestion, communément utilisée, est appelée *Least Recently Used* ou LRU.

programme en demandera la lecture. *Deuxième avantage* : cette lecture anticipée est extrêmement rapide. En effet, après lecture d'une page, la tête de lecture est prête à lire sans aucun délai les secteurs suivants, qui constituent les pages suivantes de la piste puis ceux des autres pistes du cylindre courant (figure 4.5). En une seule rotation du disque (8,33 ms) le contenu de 900 secteurs consécutifs, qu'ils appartiennent à une ou deux pistes, est transmis en mémoire centrale. Considérons par exemple la lecture anticipée de 112 pages, qui représentent à peu près la capacité d'une piste. L'opération comprend la lecture de la première page, soit 12,3 ms, suivie du transfert des 111 pages suivantes, soit 111/112 rotation ou 8,26 ms. Le coût total est de 20,56 ms, soit $t_{ts1} = 0,184$ ms par page. À comparer avec l'accès aléatoire de $t_{la1} = 12,3$ ms par page, 67 fois plus coûteux. En revanche, la lecture anticipée exige un espace plus important dans le tampon.

Modèle de disque de référence

Géométrie : 136 000 cylindres de 8 pistes (ou têtes) de 900 secteurs de 512 octets

Capacité : 979 200 000 secteurs = 501 350 400 000 octets = 467 Go

Vitesse de rotation : 7 200 révol./min ou 120 révol./s ou 8,33 ms/rév.

Déplacement du bras : min 1 ms, moyenne 8 ms, max 15 ms

Lecture d'une page aléatoire (4 Ko) : $t_{la1} = 12,3$ ms

Transfert d'une page (4 Ko) : $t_{tr} = 0,074$ ms

Lecture d'une page (4 Ko) avec anticipation d'une piste : $t_{ts1} = 0,184$ ms

Gain de la lecture avec anticipation d'une piste : $\alpha_{as} = t_{la1} / t_{ts1} = 67$

Figure 4.6 - Caractéristiques techniques de notre disque de référence

4.4 LES TECHNOLOGIES ALTERNATIVES

Le disque magnétique est pris comme support de référence dans ce chapitre en raison de sa popularité. Il existe cependant des variantes ainsi que d'autres technologies de support de données dont nous dirons quelques mots.

4.4.1 Les disques RAID

Supposons que notre disque de référence ait, par jour, une probabilité effective de panne de 0,002. Sur un parc de 1 000 disques de ce type on peut s'attendre, chaque jour, à ce que deux disques tombent en panne en moyenne et doivent être remplacés. Les grandeurs que nous avons calculées pour ce disque nous disent en outre que la lecture des données d'un cylindre (3 686 400 octets) d'adresse aléatoire dure approximativement $8 + 8 \times 8,33 = 75$ ms, ce qui correspond à un débit de 49 Mo/s.

Supposons à présent qu'on cherche à améliorer la **fiabilité des données** en enregistrant les mêmes données sur deux disques identiques. On dit que ces disques fonctionnent en **miroir** (*mirroring*) : toute modification effectuée sur l'un l'est aussi, en parallèle, sur l'autre. La lecture s'effectue sur l'un des disques en bon état.

11. *Read ahead* ou *prefetching* en anglais.

En cas de panne d'un disque, celui-ci est désactivé et peut être remplacé à *chaud* sans interruption des opérations. Le nouveau disque est alors reconstruit à partir du contenu de l'autre. La probabilité que deux disques tombent en panne un jour donné est de $0,002^2 = 0,000\ 004$. Si la fiabilité est considérablement améliorée, le débit, en revanche, reste le même. Cette redondance multiplie aussi le coût du support à l'unité (calculé par Go le plus souvent).

Les incidents affectant les disques ne sont pas toujours aussi dramatiques. En raison des hautes densités actuelles, la probabilité que la lecture d'une donnée puisse être entachée d'erreurs n'est pas négligeable. Dans ce cas, doubler les disques n'aide pas, puisque, en cas de doute, on est confronté à deux données différentes alors qu'elles devraient être identiques. Le *mirroring* à trois disques permet de fonctionner en logique majoritaire : la bonne donnée est celle qui est fournie par deux disques au moins. Cependant, un troisième disque entier n'est pas nécessaire. Il existe des techniques de **codage à détection d'erreurs** permettant d'identifier la source de l'erreur et de la corriger (codage Reed-Solomon par exemple). Associé à une donnée de N bits, un code détecteur-correcteur de n bits ($n < N$) permet de détecter l'existence de m erreurs au plus, et d'en corriger $m/2$.¹² Plusieurs disques peuvent être ainsi protégés par un seul disque de code.

Cherchons à améliorer le **débit de lecture** en distribuant les données initiales sur deux disques. Par exemple, les pages paires sur le disque 1 et les pages impaires sur le disque 2 (*striping* ou *interleaving*). Les données qui occupaient auparavant un cylindre occupent à présent un demi-cylindre sur chaque disque. La lecture se faisant sur les deux disques en parallèle, le temps de lecture des 8 pistes initiales est de $8 + 4 \times 8,33 = 41,3$ ms, ce qui correspond à 89,3 Mo/s, soit un débit multiplié par 1,8. En revanche, la fiabilité de l'assemblage est moindre. La défaillance de l'un des disques entraîne la perte du contenu des deux disques. La probabilité d'un tel accident est de $1 - (1 - 0,002)^2 = 0,004$ par jour, soit le double de celle d'un disque unique. Le coût à l'unité reste inchangé.

Ces trois expériences permettent de comprendre le principe des disques **RAID** (*redundant array of inexpensive disks*)¹³. Ce terme désigne un assemblage de deux ou plusieurs disques gérés collectivement permettant d'augmenter, soit le débit de lecture de données enregistrées, soit leur fiabilité, soit encore les deux.

Il existe plus d'une dizaine de configurations d'assemblages désignées chacune par un numéro.¹⁴ On en cite quelques unes parmi celles qui sont ou ont été largement utilisées :

- **RAID 0** : *distribution* (striping) sans redondance ni code correcteur ;
- **RAID 1** : *redondance* (mirroring) sans distribution ni code correcteur ;
- **RAID 10** ou **01** : combinaison de RAID 0 et RAID 1 ;

12. http://en.wikipedia.org/wiki/Reed-Solomon_error_correction. (juin 2018).

13. L'interprétation actuelle, *redundant array of independent disks*, est peut-être plus politiquement correcte, mais n'en est pas moins sémantiquement incorrecte : les RAID n'ont de sens que parce que les disques sont très peu coûteux, lesquels disques ne sont absolument pas indépendants !

14. [http://fr.wikipedia.org/wiki/RAID_\(informatique\)](http://fr.wikipedia.org/wiki/RAID_(informatique)). (juin 2018).

- **RAID 5** : *distribution* sans redondance des données sur au moins 3 disques; *code correcteur distribué* sur ces disques; supporte la défaillance d'un disque quelconque ; déconseillée aujourd'hui¹⁵ ;
- **RAID 6** : *distribution* sans redondance des données; *code correcteur redondant et distribué* sur ces disques; peut supporter la défaillance de deux disques quelconque.

L'architecture RAID est transparente pour le système d'exploitation, qui perçoit l'assemblage comme un disque virtuel unique. La gestion de la distribution, de la redondance et du codage est assurée par un contrôleur (matériel) ou par un pilote (logiciel).

Il faut noter qu'en raison du temps de reconstruction d'un disque défaillant (plusieurs heures), certains constructeurs recommandent l'utilisation de disques de plus faible capacité (500 Go par exemple), à la fois plus fiables et plus rapides à reconstruire.¹⁶ De nouvelles architectures sont actuellement à l'étude, cherchant à compenser les faiblesses des RAID par l'inclusion de disques de réserve.¹⁷

L'architecture RAID est également utilisée pour structurer la mémoire centrale des serveurs de grande capacité, telle la technologie RAIM d'IBM.

4.4.2 Les disques électroniques (SSD)

Un disque électronique (*Solid-State Disk/Drive* ou **SSD** en anglais) est une mémoire externe constituée de circuits EEPROM, dite aussi *mémoire flash*¹⁸. Il s'agit d'une mémoire non volatile qui ne requiert pas d'énergie pour conserver les données enregistrées, contrairement à la mémoire RAM dont est faite la mémoire centrale. Le disque électronique s'utilise comme un disque magnétique bien que son fonctionnement interne soit radicalement différent.

Au **niveau électronique**, un circuit flash est une matrice de cellules élémentaires mémorisant chacune un niveau électrique (ou état) parmi 2, 4, 8 ou plus. Dans la

15. La reconstruction d'un disque défaillant requiert la lecture des autres disques. Deux dangers dans ce processus : (1) cette lecture peut prendre plusieurs heures, pendant lesquelles le RAID n'est plus protégé convenablement et est donc vulnérable, (2) durant cette lecture, la probabilité de rencontrer un secteur illisible (il y a 4.10^9 secteurs dans un disque de 2 To) n'est pas négligeable. Or, un tel incident fait échouer la reconstruction si le code correcteur était sur le disque perdu.

16. <http://en.wikipedia.org/wiki/RAID> (juin 2018)

17. J.-F. Pâris et al., Self-repairing Disk Arrays, <http://arxiv.org/abs/1501.00513> (juin 2018)

18. Dans une mémoire ROM (*Read-Only Memory*) on ne peut que relire ce qui y a été enregistré lors de sa fabrication. Une mémoire PROM (*Programmable ROM*) est une ROM dans laquelle l'utilisateur peut écrire des données, à la suite de quoi elle devient une ROM. Dans une EPROM (*Erasable PROM*) le contenu peut être effacé pour permettre d'écrire un nouveau contenu. L'effacement se fait généralement par une exposition à une intense lumière ultra-violette, d'où le nom de **flash**, qui a été transmis à ses successeurs. Une EEPROM (*Electrically EPROM*) est une EPROM effaçable électriquement sans devoir être extraite. Les mémoires flash actuelles sont des EEPROM. Elle prennent la forme, outre de SSD, de support du BIOS dans les ordinateurs depuis les années quatre-vingt, de cartes mémoire pour les appareils photo et autres appareils mobiles et de mémoire amovibles d'appoint, dites *clés USB*.

technologie SLC (*Single-Level Cell*), une cellule peut prendre deux états, mémorisant ainsi 1 bit. La technologie MLC (*Multi-Level Cell*) peut prendre quatre états pour mémoriser 2 bits. Les mémoires TLC (*Triple-Level Cell*, 8 états, 3 bits par cellules mais existent aussi en 4 bits par cellule) sont apparues plus récemment.

Examinons **l'architecture** d'un disque électronique.¹⁹ Celui-ci est constitué de **blocs**, eux-même formés d'une séquence de **pages**. Typiquement, une page contient 4 Ko et un bloc contient 32 ou 64 pages. Comme dans un disque magnétique, la page est l'unité de lecture et d'écriture. Le bloc est l'unité d'effacement. Lorsqu'une page a été écrite, il n'est plus possible de modifier son contenu avant que le bloc tout entier ait été effacé. Donc, pour modifier le contenu d'une page, il faut lire celle-ci à partir de son bloc, puis recopier le contenu modifié dans une page vierge, éventuellement dans un autre bloc. Comme les pages sont *nomades*, le contrôleur maintient une table de conversion entre les adresses logiques des pages et l'adresse physique de sa dernière version (numéro de bloc + numéro de page dans le bloc).

Pour améliorer de débit de lecture, la mémoire est structurée en **bancs** parallèles entre lesquels les données sont distribuées (*data striping*) comme dans l'organisation RAID 0. C'est ce qui explique que les performances des SSD haut de gamme sont largement supérieures à celles des cartes mémoires et autres clés USB.

Si les opérations de lecture sont inoffensives pour un circuit flash, il n'en est pas de même des opérations d'écriture, qui induisent des phases d'**effacement** de blocs via l'injection d'un courant intense. Ce courant dégrade chimiquement les circuits électroniques, de sorte qu'après un nombre déterminé de cycles d'effacement, le circuit devient inutilisable. Ce nombre dépend de la technologie employée : 10 000 pour SLC, 3 000 pour MLC, 500 pour TLC.²⁰ Pour éviter de concentrer les modifications sur quelques blocs de la mémoire, ce qui détruirait ceux-ci en quelques jours, le contrôleur répartit les écritures dans tout l'espace de manière à utiliser (et donc à *user*) les blocs le plus uniformément possible. Cette répartition obéit à des algorithmes complexes dits de *wear leveling*. Ces algorithmes peuvent entraîner des effets secondaires problématiques. Lorsqu'un fichier n'est pas modifié pendant longtemps, il empêche ses blocs de participer à la rotation des écritures. L'algorithme décide alors de le déplacer de temps en temps pour libérer ces blocs, ce qui entraîne des coûts d'écriture supplémentaires non liés à l'usage des données (concept de *write amplification*). Ce processus ainsi que la nécessité d'effacer des blocs contribue à la grande variabilité des temps de modification des données.

Les données nécessaires à la gestion de l'espace de la mémoire, y compris le catalogue, la table de conversion des adresses logique/physiques et les informations du *wear leveling*, sont enregistrées dans un bloc spécial très rapide et très résistant à la dégradation. En outre, certains blocs sont tenus en réserve afin de remplacer, lorsque nécessaire, les blocs devenus inutilisables.

Les mémoires SLC sont les plus rapides, les plus résistantes à l'usure et donc les plus fiables, mais aussi les moins compactes et (surprise !) les plus coûteuses. Au contraire, les TLS sont les moins rapides, les moins fiables, les plus compactes et les

19. Lire aussi Cornwell, M., Anatomy of a Solid-State Drive, <http://queue.acm.org/detail.cfm?id=2385276> (juin 2018)

20. Chiffres communiqués par Kingston (juillet 2017).

moins coûteuses. Les SLC se retrouvent dans des SSD haut de gamme, les MLC dans les SSD d'entreprise et les TLC dans les mémoires permanentes des appareils mobiles, les cartes multimedia et les SSD courants ou à haute capacité.

Concernant les performances des SSD, un modèle standard en 2018 offre un débit maximum, en lecture de grands fichiers, de 300 à 1 000 Mo/s et (de *un peu* à *nettement*) moins en écriture. Par comparaison, le débit maximum en streaming de notre disque de référence est de 52 Mo/s et le débit en lecture séquentielle est de 21 Mo/s. Le temps de lecture d'une page d'adresse aléatoire est de 0.5 ms pour un SSD et de 12,3 ms pour le disque magnétique.

Les SSD offrent beaucoup d'avantages sur le disque magnétique : vitesse de lecture et d'écriture, consommation réduite, robustesse (pas de pièces mobiles) et absence de bruit. Le principal point noir est le prix : un disque magnétique de 1 To coûte 60 € mais en coûte 400 en technologie SSD (prix 2017).²¹ En outre, l'usure va réduire la durée de vie des SSD de trop faible capacité par rapport aux besoins de stockage (par exemple un ordinateur portable doté d'un SSD de 128 Go).

4.4.3 Les disques hybrides

Un disque hybride est constitué d'un disque magnétique auquel est associée une petite mémoire flash (typiquement 8 Go en 2017, ce qui est jugé trop faible par certains critiques) jouant le rôle de mémoire tampon persistante. Après quelque temps d'utilisation, la mémoire flash conserve les données les plus utilisées, diminuant ainsi leur temps d'accès. Le degré d'amélioration des performances dépend du profil des accès au disque et de la qualité de l'algorithme de gestion des échanges.

L'amélioration est sensible avec un disque à 7 200 rpm, mais moins convaincante avec un disque lent à 5 400 rpm, malgré l'argumentaire enthousiaste des fabricants (Seagate, Samsung, Toshiba et WD en particulier). Malheureusement, la tendance actuelle dans le format 2,5" est d'aligner l'offre sur les disques lents ce qui en limite l'intérêt.

4.4.4 Les mémoires à bande

Autrefois support principal des données externes dans les configurations informatiques, la bande magnétique a été concurrencée puis supplantée au début des années soixante par le disque magnétique, qui permettait l'accès aléatoire aux données. La bande magnétique est un support séquentiel. Chaque fichier et chaque enregistrement dans celui-ci est écrit à la suite du dernier qui y a été enregistré. En outre, la bande est en mode écriture ou en mode lecture mais pas les deux. Les possibilités de modification sont réduites : ajouter des données à la suite du contenu actuel et réécrire sur la bande, après l'avoir rebobinée, perdant ainsi son contenu antérieur.

Il est cependant possible, grâce à des algorithmes ingénieux et performants, d'effectuer n'importe quel traitement de gestion à l'aide d'au moins 3 lecteurs de bande. Ces algorithmes sont aujourd'hui encore largement utilisés pour exploiter

21. Voir comparaison détaillée sur http://en.wikipedia.org/wiki/Solid-state_drive (juin 2018)

des fichiers séquentiels sur disque. Nous décrirons l'un d'eux, le tri externe, à la section 4.6.2.

Le principal support actuel (*Linear Tape Open* ou **LTO**) se présente sous la forme d'une cartouche de 10 cm de côté contenant de 600 à 1000 mètres de bande. Sa vitesse de défilement en lecture est de 5 mètres par seconde. La génération LTO-6 a une capacité de 2,5 To sans compression, un débit de 160 Mo/s et un coût unitaire de 40 €. Sony annonce le test d'une cartouche pouvant contenir jusqu'à 185 To. La bande magnétique est un support robuste, fiable, peu coûteux, rapide et de grande capacité. Elle est aujourd'hui réservée à l'archivage de données rarement utilisées et à la collecte à grande vitesse de gros volumes de données²². Si le coût du support est faible, le coût du lecteur est en revanche assez élevé, vu les performances exigées.

Dans les grandes configurations, les cartouches sont logées dans des armoires qui peuvent en accueillir plusieurs milliers. Le repérage de la cartouche, son transport vers le lecteur et l'accès à une donnée sont automatisés et exigent au total plus d'une dizaine de secondes.

4.4.5 Les machines de bases de données

Un serveur destiné à mettre à la disposition d'une communauté d'utilisateurs de grandes bases de données doit satisfaire des exigences particulièrement élevées. Un tel serveur dédié à cette tâche est souvent appelé *machine de bases de données*. Sans surprise, ces exigences portent notamment sur la quantité et les performances de mémoires externes, la taille de la mémoire centrale et la capacité d'exécuter de nombreux processus en parallèle. Afin de fixer les idées sur ce type de machine, nous donnerons et commenterons les caractéristiques d'un des modèles les plus récents et les plus puissants à l'heure actuelle.

Mais avant d'aller plus loin nous examinerons le concept de *hiérarchie de mémoires*. Lorsque nous associons un tampon en mémoire centrale à un disque, nous définissons une hiérarchie à deux niveaux, niveau **1** pour le tampon et niveau **2** pour le disque. Ces niveaux se distinguent par la vitesse d'accès, la taille et le coût de la mémoire. En généralisant à l'aide des matériaux étudiés dans cette section, nous pouvons construire la hiérarchie de la figure 4.7, qui inclut en outre de la mémoire flash²³ et une batterie de cartouches magnétiques.

La **vitesse d'accès** et le **coût unitaire** vont croissant de haut en bas tandis que la **taille** est croissante de bas en haut. Le principe est simple : lorsqu'une donnée du niveau **n** est fréquemment utilisée, le gestionnaire de la hiérarchie en fait une copie au niveau **n - 1**, de manière à diminuer son temps d'accès. Le niveau **n - 1** joue en quelque sorte le rôle de tampon pour le niveau **n**. Lorsqu'un fichier archivé doit être lu, il est copié de la cartouche vers un disque. Si, sur ce dernier, des parties du fichier font l'objet de nombreux accès, ces parties sont copiées dans la mémoire flash.

22. Capture des données du collisionneur du CERN, produisant 3 à 6 Go par seconde : <http://www.economist.com/news/technology-quarterly/21590758-information-storage-60-year-old-technology-offers-solution-modern-par-exemple> (juin 2018)

23. Généralement sous la forme de cartes internes plutôt que de SSD en boîtier.

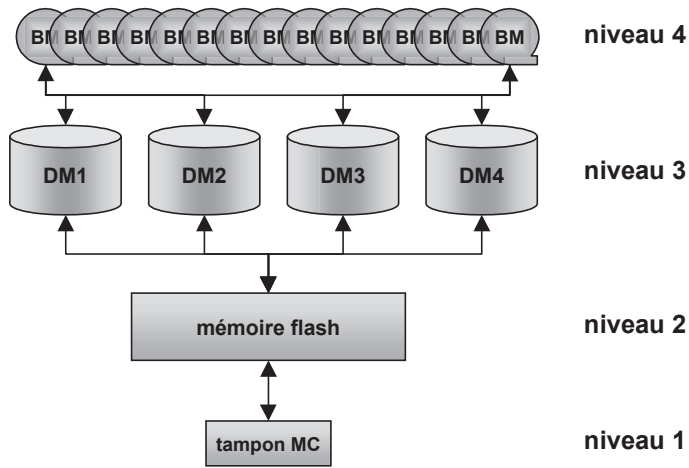


Figure 4.7 - Hiérarchie de mémoires d'un serveur de données

Selon la documentation technique disponible sur le site d'Oracle²⁴, le module standard de la machine Exadata X4-8 se présente sous la forme d'une armoire contenant principalement :

- un serveur de réseau,
- 2 serveurs de bases de données, chargés d'exécuter les requêtes SQL, comprenant 8 processeurs totalisant 240 cœurs,
- 14 serveurs de stockage, chargés des opérations de bas niveau, totalisant 168 cœurs,
- une mémoire centrale, une mémoire flash et des disques magnétiques,
- un réseau d'interconnexion entre serveurs et mémoires, ainsi qu'entre armoires, à 40 Go/s.

La machine comporte donc une hiérarchie de mémoire à trois niveaux :

- Au **niveau 1**, la RAM typique est de 4 To, extensible à 12 To.
- Au **niveau 2**, la mémoire flash, utilisée comme tampon persistant (et non comme SSD), peut atteindre 44,8 To.
- Au **niveau 3**, les 168 disques gérés par les serveurs de stockage offrent de 202 To (disques à 10 000 rpm) à 672 To (disques à 7 200 rpm).

Pour des raisons de sécurité, tous les composants sont dédoublés et remplaçables à chaud. La capacité réellement disponible peut donc être inférieure à ces chiffres. Elle est, pour les disques, de 90 à 300 To.²⁵

24. <http://www.oracle.com/technetwork/database/exadata/exadatax4-8datasheet-2243041.pdf> (juin 2018). Cette référence concerne les configurations disponibles en 2015.

25. Considérant ces chiffres ainsi que les performances annoncées en accès, Oracle adopte très certainement une technique proche d'un RAID 10.

Oracle annonce des performances impressionnantes, dues, comme il est de règle, à des conditions extrêmes de laboratoire²⁶. Il n'existe pas à l'heure actuelle de chiffres sur les performances en situation réelle d'exploitation.

4.5 ORGANISATION D'UN ESPACE DE STOCKAGE

Un espace de stockage est un conteneur dans lequel des lignes d'une ou de plusieurs tables sont, ou seront, stockées. Il occupe une partie d'une unité de mémoire externe, le plus souvent un disque magnétique, mais peut également s'étendre sur plusieurs unités. Sa taille est ajustable selon l'évolution des besoins. Pour simplifier l'exposé, et sauf mention contraire (comme les *clusters*, 4.13), nous admettrons qu'un espace contient toutes les lignes d'une table et elles seulement. On peut admettre qu'un espace de stockage correspond à ce qu'on appelle plus généralement un *fichier*. De même, une ligne constitue un *enregistrement* de ce fichier, c'est-à-dire un agrégat de valeurs logiquement corrélées constituant l'unité de lecture et d'écriture de la part des programmes. Les structures que nous allons décrire étant d'application plus large que le domaine des bases de données relationnelles, nous utiliserons désormais les termes de **fichier** et d'**enregistrement**.

4.5.1 Structure d'un fichier

Un fichier est constitué d'une suite²⁷ de N_p pages de taille L_p (figure 4.8) dans lesquelles les enregistrements seront rangés. On attribue à ces pages un numéro de 0 à N_p-1 , qui constitue l'**adresse locale** de la page dans le fichier. Connaissant l'adresse physique sur le disque de la première page du fichier, il est facile de calculer l'adresse sur le disque (n° de secteur) de la page dont on connaît l'adresse locale. Quelle longueur donne-t-on à une adresse locale ? Un champ de 4 octets contient un entier de 0 à 4 294 967 295. Il permet d'adresser une page de 4 Ko dans un fichier de 16,4 To, ce qui est suffisant pour une base de données locale.



Figure 4.8 - Un fichier est constitué d'une suite de N_p pages numérotées de 0 à $N_p - 1$

26. Par exemple, la vitesse maximum de chargement de données annoncée de 5,5 Go/s ne peut être obtenue que par une distribution des données sur au moins 32 disques à haute vitesse, dont le débit par disque est proche de 175 Mo/s.

27. Nous supposons dans un premier temps que ces pages sont consécutives sur le disque.

4.5.2 Allocation d'espace à un fichier

Lorsqu'un fichier est créé, un certain nombre de pages lui sont allouées (*allocation initiale*). Lorsque cet espace devient insuffisant, de nouvelles pages lui sont ajoutées (*extension*). La taille de cette extension est constante ou variable selon le contexte.²⁸

L'allocation initiale et les extensions sont choisies dans l'espace libre du disque. Pour des raisons qui seront discutées à la section 4.5.5, cet espace libre peut être fragmenté. Il se présente donc sous la forme d'un ensemble de suites de secteurs consécutifs. Par conséquent, un fichier peut, au cours du temps, devenir lui aussi fragmenté. Il existe plusieurs méthodes pour répertorier les fragments libres d'un disque : (1) ils sont regroupés pour former un fichier système caché, (2) ils forment une chaîne réalisée par des pointeurs tels que chaque fragment indique l'adresse du suivant, (3) une chaîne de bit décrit l'espace du disque, à raison d'un bit par unité d'espace (par exemple, blocs de 2 Ko), la valeur 0 indiquant que cette portion est libre et la valeur 1 qu'elle est occupée.

On trouvera à la section 4.15 l'exemple de gestion de l'espace d'une base de données propre à SQL Server.

4.5.3 Structure des enregistrements d'un fichier

Dans les SGBD classiques, un enregistrement se présente comme une chaîne d'octets constituée de la suite des valeurs des champs de cet enregistrement. Un enregistrement (une ligne) de la table CLIENT est formé d'une valeur de NCLI, suivie d'une valeur de NOM, suivie ... d'une valeur de COMPTE. À ces données *visibles* s'ajoutent le plus souvent des *données techniques* indiquant par exemple le type de l'enregistrement, son état (actif, supprimé), sa longueur totale, la longueur des valeurs de chaque champ, des indicateurs de présence des valeurs des champs facultatifs, l'adresse de certains composants spécifiques (page de niveau 1 de l'index d'un fichier séquentiel indexé) et pointeurs divers permettant l'accès rapide.

4.5.4 Rangement des enregistrements dans un fichier

Il existe plusieurs méthodes de rangement d'un ensemble d'enregistrements dans une séquence de pages. Nous en décrirons quatre. Appelons **page de base** la page dans laquelle nous *décidons* d'écrire un enregistrement.

- a) La technique la plus simple est le rangement contigu (figure 4.9, schéma a). Chaque enregistrement est écrit à la suite du précédent sans espace intercalaire. Si un enregistrement est trop long par rapport à l'espace libre de sa page de base, la portion qui déborde est stockée dans la (ou les) page(s) suivante(s). L'adresse <p,d> d'un enregistrement est formée du numéro local de sa page (p) et du déplacement de son premier octet dans la page (d). Cette organisation convient pour les

28. Chez Oracle, cette taille augmente au fur et à mesure que des extensions sont ajoutées, afin de diminuer le coût cet ajout : ajouter une grande extension est moins coûteux qu'en ajouter plusieurs petites équivalentes.

fichiers statiques, dans lesquels les enregistrements ne sont ni modifiés ni supprimés²⁹.

- b) Une deuxième technique (schéma *b*) est basée sur une décomposition de chaque page en n_s cellules de taille fixe (*slot*). Chaque enregistrement est écrit dans une cellule libre. L'adresse d'un enregistrement est $\langle p, c \rangle$, où c (de 0 à n_s-1) est le numéro de la cellule dans la page. Cette organisation permet la mise à jour aisée des enregistrements.
- c) La troisième technique décompose l'espace du fichier en cellules de taille fixe (*slot*), et ce, indépendamment des frontières de pages (schéma *c*). Une cellule peut couvrir plusieurs pages consécutives. Chaque cellule peut contenir un enregistrement, dont l'adresse est le numéro de cette cellule $\langle c \rangle$.
- d) La quatrième technique (schéma *d*) est plus complexe mais permet une gestion dynamique du contenu des pages. Les enregistrements sont rangés dans la page en vrac de manière séquentielle. À l'autre extrémité de la page est rangé un tableau de pointeurs, dont chaque élément reprend la position (numéro d'octet) d'un enregistrement dans la page. On convient que l'élément d'indice n du tableau référence l'enregistrement n° n de la page. L'adresse d'un enregistrement est $\langle p, n \rangle$.

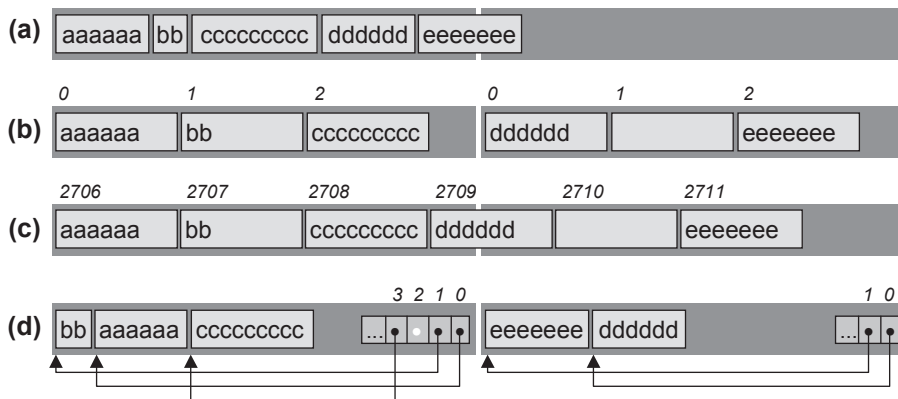


Figure 4.9 - Quatre techniques classiques de rangement d'enregistrements dans une suite de pages (ici deux pages consécutives)

On observe que, dans ces techniques, il est possible de lire un enregistrement dont on connaît l'adresse en un seul accès au disque, soit 12,3 ms, éventuellement complété de la lecture d'une ou plusieurs pages suivante(s) à 0,074 ms chacune. L'adresse d'un enregistrement occupe de 5 à 6 octets.

Que se passe-t-il lorsque, dans la technique *d*, un enregistrement est (ou est devenu) trop long pour l'espace qui lui est initialement alloué dans sa page de base ? Il existe deux techniques de résolution : l'indirection et la fragmentation. L'**indirection** consiste à placer l'enregistrement dans une autre page offrant suffisamment d'espace et à écrire dans la page de base un enregistrement très court (une *carte de*

29. Pourquoi ?

visite) qui indique simplement où l'enregistrement est stocké. La **fragmentation** consiste à écrire un premier fragment de l'enregistrement dans sa page de base, accompagné d'un pointeur qui donne l'adresse du fragment complémentaire suivant, stocké dans une autre page. Dans ces deux cas, la lecture d'un enregistrement exige au moins deux accès au disque, ce qui augmente le temps d'accès à un enregistrement. Il est à noter que la fragmentation permet le stockage d'un enregistrement dont la longueur est supérieure à celle des pages.

Il reste une dernière question à aborder : comment choisit-on la page de base d'un enregistrement ? Soit ce choix est indifférent, auquel cas la première page qui se présente convient. On parlera d'un **rangement en vrac**. Soit le choix est imposé par la technique d'organisation du fichier qui matérialise l'index primaire. Il s'agit alors d'un fichier **séquentiel indexé** (section 4.8) ou d'un fichier **calculé** (section 4.9). Soit, enfin, l'adresse d'un enregistrement est idéalement proche de celle d'un autre enregistrement auquel il est logiquement lié. On parlera alors d'**agrégation** physique ou *clustering* (section 4.13).

4.5.5 Le phénomène de fragmentation d'un fichier

Un fichier de petite taille est généralement constitué de pages consécutives sur le disque (figure 4.8). En revanche, un fichier de grande taille est souvent décomposé en sous-séquences contiguës appelées *fragments* (figure 4.10). On dit de ce fichier qu'il est *fragmenté*. Ce phénomène ne met pas en danger l'intégrité des données, mais peut augmenter significativement le temps d'accès séquentiel au fichier. Nous allons étudier les origines de la fragmentation des fichiers.

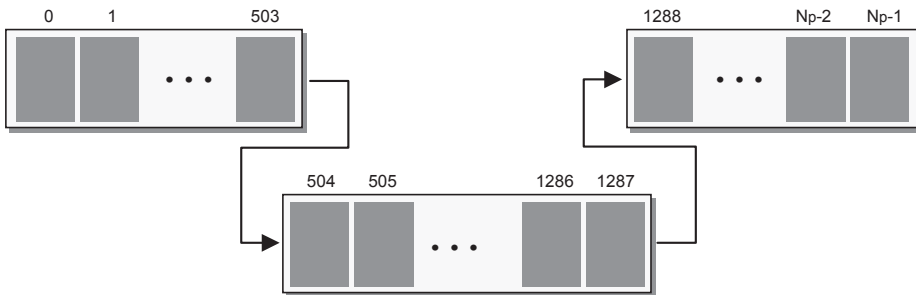


Figure 4.10 - Fichier réparti dans trois fragments éloignés les uns des autres sur le disque

En général, l'espace qu'un fichier occupe sur le disque est consécutif à celui du fichier précédent et est suivi de l'espace du fichier suivant. Lorsque ce fichier est supprimé, l'espace qu'il occupe est libéré et peut être réutilisé. Cet espace prend la forme d'un *trou* dans l'espace du disque, encadré par les pages des fichiers précédent et suivant. Après un certain temps, le disque comporte un grand nombre de tels *trous* de tailles diverses répartis de manière aléatoire. On dit que l'espace libre du disque présente une structure en *gruyère* ou qu'il est *fragmenté*.

Deux processus vont provoquer la fragmentation d'un fichier dans un disque lui-même fragmenté : la *création d'un fichier* et l'*extension d'un fichier existant*. Idéa-

lement, on alloue à un nouveau fichier une suite de pages contiguës. On recherche donc un *trou* dont la taille est au moins égale à celle de ce fichier. Si on en trouve un, ce *trou* va être partiellement occupé, laissant ainsi un *trou* plus petit. Si on n'en trouve pas, le fichier va être rangé dans plusieurs *trous*, chacun accueillant un fragment du fichier. Ce fichier est donc fragmenté dès sa création. Si un fichier ne dispose plus d'assez d'espace pour accueillir de nouveaux enregistrements, on lui alloue un lot de pages supplémentaires, idéalement à la suite de sa dernière page, mais plus généralement dans un *trou* disponible ailleurs. Le fichier se fragmente donc aussi au cours de son évolution³⁰.

Si les performances se dégradent de manière excessive, on procédera à une réorganisation de l'espace libre du disque et des fichiers via un utilitaire de *défragmentation*.

4.5.6 Organisation des fichiers sur le support

Un disque peut accueillir plusieurs dizaines de milliers de fichiers, classés dans des répertoires. L'accès à l'un de ces fichiers suppose qu'il soit possible de déterminer son emplacement sur le disque. Il existe sur chaque disque (plus généralement sur chaque support externe) un **catalogue** qui recense tous ses fichiers et répertoires. Ce catalogue est lui-même un fichier du système qui, pour chaque fichier du disque, reprend notamment son nom, son chemin d'accès, le mode de rangement des enregistrements et, pour chaque fragment, son adresse sur le disque et son nombre de pages. Grâce à ces informations, il est aisé de calculer l'adresse sur le disque d'un enregistrement dont on connaît le fichier et l'adresse dans celui-ci.

4.6 TRAITEMENT SÉQUENTIEL D'UN FICHIER

Il est fréquent qu'un fichier soit traité et géré de manière séquentielle, c'est-à-dire que ses enregistrements y sont rangés chacun à la suite du précédent et qu'ils sont relus dans le même ordre, du premier au dernier. Dans ce mode d'exploitation des données, on n'utilise pas les possibilités d'accès direct à un enregistrement.

Ce cas de figure recouvre deux situations : soit le fichier est séquentiel par construction (fichier enregistré sur une bande magnétique par exemple), de sorte que l'accès direct n'est pas prévu, soit le fichier supporte l'accès direct (accès direct à un enregistrement par son adresse ou via un index) mais les traitements considérés n'en font pas usage.

Le lecteur sera peut-être étonné de l'intérêt qu'il y a à étudier une structure de fichier aussi primitive, qui fleure bon les bandes magnétiques des années 1960. Il sera sans doute encore plus étonné d'apprendre que les fichiers séquentiels sont, et resteront encore longtemps, majoritaires et que des traitements de bases de données particulièrement complexes se basent sur leur utilisation. Mentionnons quelques exemples :

30. Pour nous résumer : en raison de la gestion dynamique des données, l'espace d'un disque, un fichier et un enregistrement peuvent, chacun à leur manière, être fragmentés.

- recherche des lignes satisfaisant une condition de sélection pour laquelle il n'existe pas d'index, forçant le SGBD à parcourir toute la table séquentiellement (`where COMPTE > 1000`);
- calcul de statistiques sur la totalité d'une table (`select count(*), sum(COMPTE) from CLIENT`);
- tri d'un million d'enregistrements (clause SQL `order by`);
- fusion de deux longues listes d'enregistrements (jointure de deux tables volumineuses);
- constitution de groupes d'enregistrements (clause SQL `group by`).

Nous établirons d'abord le **temps de lecture** des enregistrements d'un fichier puis nous nous intéresserons au processus de **tri** d'un fichier séquentiel.

4.6.1 Lecture séquentielle d'un fichier

Soit un fichier de **N_p** pages de taille **L_p**. La lecture séquentielle de ce fichier consiste à lire les pages successives du fichier et, dans chacune, à extraire les enregistrements dans leur ordre naturel. Avant de calculer le coût de cette lecture, nous allons établir un résultat préliminaire : la lecture d'une courte séquence de pages.

a) Lecture d'une courte séquence de pages

Une séquence de **p** pages comporte une première page suivie des **p - 1** pages suivantes. On suppose que ces **p** pages sont consécutives sur le disque, que, pour simplifier, elles appartiennent au même cylindre (le bras reste fixe) et que la lecture n'est pas interrompue par une demande d'un autre processus. Le temps de lecture de la première page est **t_{la1}** et celui des pages suivantes se réduit à leur temps de transfert, soit, pour notre disque de référence, **t_{tr} = 0,074 ms**. Le temps de lecture de la séquence est donc :

$$t_{isp} = t_{la1} + (p - 1) \times t_{tr}$$

b) Lecture séquentielle d'un fichier

Dans cette analyse, nous négligerons le coût d'accès au catalogue et le temps de calcul nécessaire aux opérations de lecture. Le coût de la lecture du fichier est donc celui de la lecture de ses pages sur le disque. Il dépendra de deux facteurs :

- Le fichier est-il d'un seul tenant ou est-il fragmenté ?
- Le disque est-il réservé au processus lecteur, le bras ne se déplaçant qu'à la demande de celui-ci, ou est-il fortement partagé servant dès lors un grand nombre de demandes simultanées ?

Dans un premier temps, nous supposons que le fichier **n'est pas fragmenté**, reportant à la section suivante l'analyse du phénomène de fragmentation. Nous retiendrons trois scénarios, du plus favorable au plus coûteux.

1. *Disque dédié au processus lecteur.* Le fichier étant d'un seul tenant, la lecture peut s'effectuer sans interruption, en *streaming*³¹. Le coût est celui de la lecture d'une séquence de N_p pages. Si le fichier occupe un seul cylindre, il vient :

$$t_{sf1} = t_{ia1} + (N_p - 1) \times t_{tr}$$

Si le fichier occupe plusieurs cylindres, on comptera le temps des déplacements du bras du cylindre courant vers le cylindre suivant, soit, dans notre modèle de disque, 1 ms. Il reste à déterminer le nombre $ncyl$ de cylindres supplémentaires à atteindre de la sorte (en ignorant le premier). On admet qu'en moyenne le fichier débute au milieu d'un cylindre. On peut donc affiner le calcul :

$$ncyl = \lfloor N_p \times L_p / L_{cyl} + 0,5 \rfloor$$

$$t_{sf1} = t_{ia1} + (N_p - 1) \times t_{tr} + ncyl \times seek1$$

On observe cependant que les coûts de l'accès initial (12,3 ms) et des déplacements du bras ($ncyl \times seek1$) sont négligeables et peuvent être ignorés. Le calcul de la lecture en streaming peut dès lors se simplifier :

$$t_{sf1} \cong N_p \times t_{tr}$$

2. *Disque partagé et lecture anticipée.* Il s'agit d'une situation courante : la lecture se traduit par une suite de lectures anticipées. L'évaluation est très simple :

$$t_{sf2} = N_p \times t_{sf1}$$

3. *Disque fortement partagé et lecture non anticipée ou pages disposées aléatoirement.* Il s'agit d'une configuration extrême qu'on ne rencontrera pas en pratique. Chaque page réclame une lecture aléatoire³². On a donc :

$$t_{sf3} = N_p \times t_{ia1}$$

c) Écriture dans un fichier séquentiel

On sait que le coût de l'écriture d'une page est égal à celui de sa lecture, si on écarte l'option d'une vérification après écriture. On pourrait donc imaginer que le coût d'écriture d'un fichier séquentiel est égal à celui de la lecture d'un fichier de même taille. En fait ce coût dépend de l'existence préalable du fichier de destination. Si ce fichier existe déjà, le processus d'écriture consiste à écrire une séquence de pages dans un espace pré-réserve. Si au contraire, ce fichier doit être créé pour la circonstance, avec allocation initiale et éventuellement allocation d'extensions, alors le coût de cette allocation devra être comptabilisé. Dans ce chapitre, nous adopterons la première hypothèse, qui conduit à adopter un coût d'écriture égal à celui de lecture.

31. Il existe une hypothèse supplémentaire : le traitement des enregistrements est plus rapide que leur lecture, de sorte qu'il ne freine pas cette lecture en continu. Dans cette hypothèse, la lecture anticipée est inutile.

32. Notons cependant qu'il existe des techniques d'ordonnancement des demandes d'accès au disque qui optimisent globalement les déplacements du bras. Point d'entrée utile : http://en.wikipedia.org/wiki/I/O_scheduling [septembre 2010]

On rappelle à ce propos que l'écriture dans une page d'adresse aléatoire (il ne s'agit plus ici de fichiers séquentiels) exige que cette page ait été au préalable lue sur le disque et chargée dans le tampon.

d) Influence de la fragmentation

Il apparaît clairement que la fragmentation d'un (grand) fichier n'affectera pas les processus des scénarios 2 et 3, l'unité de lecture (respectivement *piste* et *page*) y étant, sauf cas pathologique, plus petite que les fragments. On ne considère donc que la lecture en *streaming*.

Selon ce scénario, la différence de temps de lecture entre un fichier monolithique et un fichier décomposé en q fragments est celui des $q - 1$ sauts aux fragments suivants (figure 4.10). En toute rigueur, on remplace q lectures séquentielles (coût = t_{tr}) par q lectures aléatoires (coût = t_{la1}). Il vient donc, en désignant le temps de lecture du fichier fragmenté par t_{lsf4} :

$$t_{lsf4} = N_p \times t_{tr} + q \times (t_{la1} - t_{tr})$$

La dégradation D se mesure directement par le rapport entre t_{lsf4} et t_{lsf1} , soit, pour notre disque de référence :

$$D = t_{lsf4} / t_{lsf1} = (N_p \times t_{tr} + q \times (t_{la1} - t_{tr})) / (N_p \times t_{tr}) = 1 + 166 \times q / N_p$$

On observe qu'une dégradation de 10% ($D = 1,1$) apparaît dès que q / N_p atteint 0,000 602 4, soit, pour un fichier de 250 000 pages, $q = 150$ fragments. Le même fichier composé de 15 fragments présenterait une dégradation de 1%, ce qui serait négligeable. Tous les scénarios étudiés sont repris à la figure 4.11.

e) Exemple

Calculons le temps de lecture d'un fichier de $N_p = 250\,000$ pages de 4 Ko implanté sur notre disque de référence.

Dans le cas le plus favorable (scénario 1), la lecture continue coûtera :

$$n_{cyl} = \lfloor 250\,000 \times 4\,096 / 3\,686\,400 + 0,5 \rfloor = 278$$

Calcul du temps de lecture d'un fichier séquentiel

Courte sous-séquence de p pages

$$t_{isp} = t_{la1} + (p - 1) \times t_{tr}$$

Disque dédié, fichier non fragmenté (streaming)

$$t_{lsf1} \cong N_p \times t_{tr}$$

Disque partagé, lecture anticipée

$$t_{lsf2} = N_p \times t_{ls1}$$

Disque partagé, pas de lecture anticipée ou pages distribuées aléatoirement

$$t_{lsf3} = N_p \times t_{la1}$$

Disque dédié, fichier fragmenté (streaming avec q fragments)

$$t_{lsf4} = N_p \times t_{tr} + q \times (t_{la1} - t_{tr})$$

Figure 4.11 - Temps de lecture d'un fichier séquentiel selon cinq scénarios

$$t_{\text{tsf1}} = 12,3 + (250\,000 - 1) \times 0,074 + 278 \times 1 = 18\,790 \text{ ms} = \mathbf{18,8 \text{ s}}$$

$$t_{\text{tsf1}} \cong 250\,000 \times 0,074 = 18\,790 \text{ ms} = \mathbf{18,5 \text{ s}} \text{ (calcul simplifié)}^{33}$$

La différence entre ces deux derniers résultats est négligeable. Le cas intermédiaire (scénario 2) nous donne un coût plus élevé mais plus proche des performances d'un serveur en charge :

$$t_{\text{tsf2}} = 250\,000 \times 0,184 = \mathbf{46 \text{ s}}$$

Quant au cas défavorable (scénario 3), il conduit à un coût très élevé qui démontre, s'il en était besoin, l'intérêt de la lecture anticipée :

$$t_{\text{tsf3}} = 250\,000 \times 12,3 = \mathbf{3.075 \text{ s}}$$

4.6.2 Tri d'un fichier séquentiel

Le tri de courtes séquences d'enregistrements s'effectuera le plus simplement par leur rangement dans un tableau en mémoire centrale suivi d'un tri interne selon une des nombreuses méthodes de tri disponibles. Cette technique n'est plus envisageable pour trier un fichier de grande taille qui occuperait un espace excessif en mémoire centrale. On effectuera alors ce qu'on appelle un **tri externe**, lequel travaille sur de simples fichiers séquentiels³⁴. Pour simplifier, nous supposons que le fichier doit être trié sur un *champ identifiant* et que ce dernier est de type *entier*, de sorte que la procédure peut se réduire au tri d'une suite de nombres entiers.

a) La procédure de base

Observons d'un peu plus près une suite arbitraire (figure 4.12). Elle n'est pas triée mais on observe qu'elle est naturellement constituée de sous-séquences plus ou moins longues d'éléments déjà en ordre : (82, 98), (87), (78, 83), etc. Une telle sous-séquence est appelée **monotonie**. Toute liste est donc constituée d'une suite de monotonies. On peut montrer [Knuth, 1998] qu'une liste de **N** éléments *parfaitement en désordre* comporte en moyenne **N** / 2 monotonies. Une liste triée ne comporte qu'une seule monotonie et une liste triée dans le sens inverse en comporte **N**.

82 98 87 78 83 23 79 56 85 92 72 91 80 34 67

Figure 4.12 - Liste arbitraire comportant 8 sous-séquences triées (ou *monotonies*)

Distribuons les 8 monotonies de cette liste en deux sous-listes, la première reprenant les 4 monotonies impaires et la seconde les 4 monotonies paires (figure 4.13, haut). On appelle cette distribution **éclatement** de la liste source. Ensuite, nous procédons à la **fusion** (ou *interclassement*) des deux sous-listes, de telle sorte que les monotonies de même rang de celles-ci sont fusionnées pour former une monotonie de la liste résultante (figure 4.13).

33. Soit une vitesse de lecture de 55 Mo/s

34. La technique décrite a été proposée par von Neumann en 1945 pour évaluer le jeu d'instructions de l'EDVAC [Knuth, 1998].

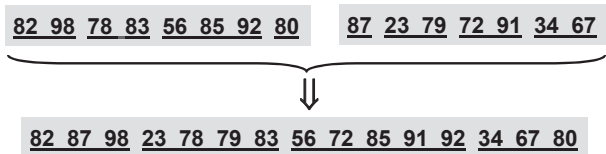


Figure 4.13 - Fusion des monotonies de deux listes non ordonnées.
Des 8 monotonies des listes sources, il en reste 4 dans la liste résultante.

La liste résultante ne contient plus que 4 monotonies, le nombre initial ayant été divisé par 2. En traitant cette liste de la même manière (éclatement puis fusion), on obtient une liste de 2 monotonies, puis une liste d'une seule monotonie qui constitue la liste triée.

Dans l'enchaînement des étapes, on observe qu'on constitue une liste complète pour la reparcourir immédiatement après. Il serait plus efficace de ne pas matérialiser la liste qui résulte d'une fusion mais d'effectuer directement la répartition en deux demi-listes des éléments successifs qui sont fournis par le processus de fusion. On définit ainsi le processus de *fusion-éclatement*, qui parcourt deux demi-listes et qui constitue deux demi-listes comportant deux fois moins de monotonies. On procède donc à l'éclatement de la liste source, puis on applique itérativement le processus de *fusion-éclatement* jusqu'à ce qu'une des listes résultantes soit vide. La figure 4.14, où chaque barre verticale identifie une monotonie, détaille les étapes successives de ce processus. Elle montre que la liste source est triée en 3 phases de fusion-éclatement précédées d'une étape d'éclatement.

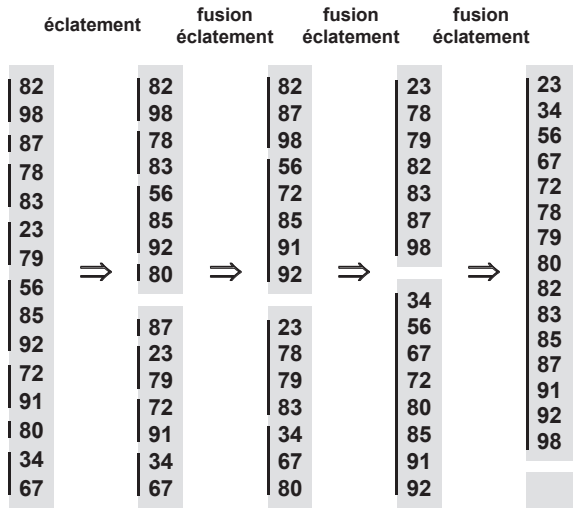


Figure 4.14 - Tri d'une liste ou d'un fichier

Cette procédure, qui se réduit à lire et écrire séquentiellement des listes, est directement applicable aux fichiers séquentiels. Les quatre opérations ont le même coût : une lecture et une écriture du fichier complet (ou, ce qui est équivalent, de deux demi-fichiers). En considérant que l'écriture se fait sans vérification, lecture et écri-

ture ont même coût, le temps du tri du fichier source est de $2 \times (1 + 3) = 8$ fois celui de la lecture de ce fichier.

b) Calcul du temps de tri d'un fichier

Considérant que le tri d'un fichier de **Nr** enregistrements nécessite une phase d'éclatement et **P** phases de fusion-éclatement et que le temps de lecture ou d'écriture du fichier est **tisf2** (disque partagé, lecture anticipée) le temps de tri du fichier est :

$$ttri0 = 2 \times (1 + P) \times tisf2$$

P se calcule en considérant que :

- chacune des **P** phases divise le nombre de monotonies par deux
- le nombre de monotonies dans le fichier source est **M**; pour un fichier quelconque en désordre, on l'estime égal à $Nr / 2$ [Knuth, 1998].
- au terme de la dernière phase, le nombre de monotonies est de 1.

La phase **1** produit $M/2$ monotonies. La phase **k** en produit $M / 2^k$ et la phase **P** en produit $M / 2^P$, qui ne doit pas dépasser 1. On a donc (annexe A4, section A4.2.1/d) :

$$P = \lceil \log_2 M \rceil$$

Nous obtenons ainsi une première estimation du temps de tri :

$$ttri0 = 2 \times (1 + P) \times tisf2 = 2 \times (1 + \lceil \log_2 M \rceil) \times tisf2$$

Supposons à présent que le fichier soit en désordre parfait par rapport au critère de tri ($M = Nr / 2$), on a dès lors $\lceil \log_2 M \rceil = \lceil \log_2 Nr \rceil - 1$. D'où :

$$ttri0 = 2 \times \lceil \log_2 Nr \rceil \times tisf2$$

Puisque **tisf2** est proportionnel à **Nr**, la complexité de l'algorithme est $O(Nr \log Nr)$.

c) Optimisation du tri externe

La procédure décrite ci-dessus est très sommaire et peut être considérablement améliorée sans grands efforts. Nous décrirons quelques optimisations classiques, dont les détails sont précisés dans l'annexe A4, section A4.2.1.

• Tri interne lors de la phase initiale d'éclatement

Lors de la phase initiale d'éclatement, les enregistrements du fichier d'entrée transitent par la mémoire centrale pour être distribués entre les deux demi-fichiers de sortie. On décide de lire d'affilée **b** enregistrements (par exemple 10 000), de les ranger en mémoire centrale, **de les y trier**, puis de les écrire dans un des demi-fichiers. On crée ainsi des monotonies de taille 10.000 au lieu de 2 dans la procédure standard, ce qui diminuera le nombre de monotonies et donc le nombre de phases. Mieux, en appliquant une variante du *tri du tournoi* (annexe A4.2.1/e), on produit des monotonies dont la taille moyenne est de $2b$. On calcule le nombre **M'** de phases :

$$M' = \lceil Nr / 2b \rceil$$

$$P' = \lceil \log_2 M' \rceil = \lceil \log_2 \lceil Nr / 2b \rceil \rceil = \lceil \log_2 Nr / 2 \rceil - \lfloor \log_2 b \rfloor = P - \lfloor \log_2 b \rfloor$$

$$ttri1 = 2 \times (1 + P') \times ttsf2 = 2 \times (1 + P - \lfloor \log_2 b \rfloor) \times ttsf2$$

Par exemple, $b = 8\,200$ permet d'économiser 13 phases.

• Augmentation du nombre de voies

La procédure de base distribue les monotonies dans deux (demi-)fichiers. On parle alors d'un tri à 2 **voies**. En les distribuant dans un nombre plus important, soit v , de fichiers, le nombre de phases peut être réduit :

$$P'' = \lceil \log_v M' \rceil = \lceil \log_2 M' / \log_2 v \rceil \cong \lceil P' / \log_2 v \rceil$$

$$ttri2 = 2 \times (1 + P'') \times ttsf2 = 2 \times (1 + \lceil P' / \log_2 v \rceil) \times ttsf2$$

Si on choisit autant de voies que de monotonies ($v = M'$), on obtient $P'' = 1$. Le tri est alors linéaire en Nr .

$$ttri2' = 4 \times ttsf2$$

• Parallélisme lecture/écriture

Si, pour chaque phase, les fichiers d'entrée sont stockés sur un disque et les fichiers de sortie sur un autre, alors les opérations de lecture et d'écriture peuvent s'exécuter en parallèle. Le temps d'une phase est dès lors de $ttsf2$ au lieu de $2 \times ttsf2$.

$$ttri3 = (1 + P'') \times ttsf2 = (1 + \lceil P' / \log_2 v \rceil) \times ttsf2$$

Ou, si $v = M'$: $ttri3' = 2 \times ttsf2$

• Parallélisme des voies en lecture et en écriture

Il est possible d'étendre le principe du parallélisme aux différentes voies d'entrée et/ou de sortie. En distribuant les voies sur autant de disques indépendants, celles-ci sont lues en parallèle, de sorte que le coût de lecture et/ou d'écriture des voies se réduit à celui d'une seule voie. Le temps de tri est alors, si toutes les voies sont parallélisées :

$$ttri4 = (1 + P'') \times ttsf2 / v = (1 + \lceil P' / \log_2 v \rceil) \times ttsf2 / v$$

Par exemple, la répartition de 5 voies sur 5 disques utilise 10 disques mais divise le temps de chaque phase par 5. Cette technique réclamant un grand nombre de disques sera surtout appliquée dans les machines de base de données.

d) Exemple

On considère un fichier séquentiel de 1 000 000 enregistrements de 400 octets. Les enregistrements possèdent un champ identifiant mais sont en désordre parfait par rapport à celui-ci. L'écriture se fait sans vérification. On calcule le temps du tri du fichier selon ces optimisations. Si on adopte le mode de stockage (a) de la figure 4.9. le fichier comporte 97 657 pages. Le temps de lecture ou d'écriture du fichier est de :

$$ttsf2 \cong Np \times tss1 = 97\,657 \times 0,184 = 17\,970 \text{ ms ou } 18 \text{ s}$$

- **Procédure de base**

$$M = \lceil Nr / 2 \rceil = 500\,000$$

$$P = \lceil \log_2 M \rceil = 19$$

$$ttri0 = 2 \times (1 + P) \times t_{sf2} = 720 \text{ secondes ou } 12 \text{ minutes}$$

- **Tri interne initial**

On réserve un espace de tri de 50 Mo, soit $b = 100\,000$

$$M' = \lceil Nr / 2b \rceil = 5$$

$$P' = \lceil \log_2 M' \rceil = 3$$

$$ttri1 = 2 \times (1 + P') \times t_{sf2} = 144 \text{ s ou } 2 \text{ min } 24 \text{ s}$$

- **Augmentation du nombre de voies**

On conserve un espace de tri initial de 40 Mo et on répartit les enregistrements sur 5 voies en entrée et 1 en sortie (théoriquement 5 en sortie mais une seule sera utilisée). 6 fichiers sont donc ouverts durant le processus.

$$M' = \lceil Nr / 2b \rceil = 5$$

$$P'' = \lceil \log_v M' \rceil = \lceil \log_2 M' / \log_2 v \rceil = 1$$

$$ttri2 = 2 \times (1 + P'') \times t_{sf2} = 72 \text{ s ou } 1 \text{ min } 12 \text{ s}$$

- **Parallélisme lecture/écriture**

Dans les mêmes conditions que ci-dessus, les voies d'entrées sont sur un disque et les voies de sortie sur un autre.

$$ttri3 = (1 + P'') \times t_{sf2} = 36 \text{ s}$$

- **Parallélisme des voies**

Etant donné qu'il ne reste qu'une voie de sortie, répartir les 5 fichiers en entrée sur 5 disques ne présente pas d'intérêt.

e) **Les calculs en résumé**

La figure 4.15 reprend les différents résultats par optimisation. Ces modèles de calcul du temps de tri selon les différentes optimisations ont été traduits dans la feuille de calcul SORT.xls disponible sur le site de l'ouvrage.

4.7 LES INDEX

Un index est une structure technique, en général invisible pour le programmeur, qui permet un accès sélectif et rapide aux enregistrements d'un fichier³⁵ qui possèdent des valeurs déterminées de certains champs.

35. Bien que la plupart des index soient associés à des **tables** et permettent d'accéder à leurs **lignes**, nous continuerons à parler plus généralement de **fichiers** et d'**enregistrements**. Nous verrons en effet que les index peuvent être associés à d'autres constructions, telles que *clusters*.

Calcul du tri d'un fichier*Procédure de base*

$$P = \lceil \log_2 M \rceil$$

$$ttri0 = 2 \times (1 + P) \times t_{lsf2}$$

Tri interne initial

$$M' = \lceil Nr / 2b \rceil$$

$$P' = \lceil \log_2 M' \rceil$$

$$ttri1 = 2 \times (1 + P') \times t_{lsf2}$$

Augmentation du nombre de voies

$$P'' = \lceil \log_v M' \rceil = \lceil \log_2 M' / \log_2 v \rceil$$

$$ttri2 = 2 \times (1 + P'') \times t_{lsf2}$$

$$\text{si } v = M', ttri2 = 4 \times t_{lsf2}$$

Parallélisme lecture/écriture

$$ttri3 = (1 + P'') \times t_{lsf2}$$

$$\text{si } v = M', ttri3 = 2 \times t_{lsf2}$$

Figure 4.15 - Temps de tri d'un fichier (échanges avec la mémoire secondaire)

Une requête telle que :

```
select NOM,LOCALITE from CLIENT where NCLI = 'C400'
```

pourra être exécutée par une lecture séquentielle des lignes de la table CLIENT jusqu'à trouver celle dont la valeur de NCLI est 'C400'. Si cette table comprend 200 000 pages, cette procédure en lira en moyenne la moitié, ce qui consommera, dans des conditions favorables, un temps de l'ordre de $100\,000 \times 0,184$ ms, soit 18,4 secondes. Un index construit sur les valeurs de NCLI permettra, dans des conditions normales, de lire cette ligne en moins de 25 ms, soit en 736 fois moins de temps. On distingue les index primaires et les index secondaires.

a) Index primaires

Un index primaire est créé en même temps que le fichier car il impose un mode de stockage spécifique des enregistrements³⁶. Il est le plus souvent (mais pas exclusivement) associé à l'identifiant primaire d'une table. Il ne peut être créé (`create index`) ni supprimé (`drop index`) dynamiquement sans entraîner des coûts importants de restructuration de la table. En revanche, il fournit d'excellentes performances en temps d'accès et en espace occupé. Comme nous le verrons plus loin, les technologies des index primaires se retrouveront aussi dans l'implémentation des index secondaires et des *clusters*. Nous étudierons les deux techniques principales : l'organisation séquentielle indexée et l'organisation calculée.

36. En Oracle, un index primaire de type *séquentiel indexé* est déclaré lors de la création de la table (`create table T(...) organization index`) et non par l'instruction classique `create index`.

b) Index secondaires

Un index secondaire est une structure indépendante du fichier des enregistrements qui peut être ajoutée et supprimée à tout moment. En revanche, les volumes et les temps d'accès sont en général moins favorables que ceux des index primaires. De nombreuses bases de données ne contiennent que des index secondaires, malgré de moins bonnes performances, en raison de leur souplesse de création et suppression.

4.8 ORGANISATION SÉQUENTIELLE INDEXÉE

Un *fichier séquentiel indexé* construit sur une suite de champs K , appelée **clé**, est formé d'un fichier séquentiel, dit **fichier de base**, contenant les enregistrements **ordonnés** selon les valeurs de K et d'un **index** basé sur K . La figure 4.16 montre clairement ces deux composants. On y trouve à droite un fichier de base contenant une suite d'enregistrements (ou lignes) CLIENT, ordonnés par valeurs croissantes de la clé, constituée du champ (ou colonne) NCLI. À gauche, l'index se présente sous la forme d'un arbre de pages qu'on parcourt de gauche à droite.

4.8.1 Structure d'un fichier séquentiel indexé

a) Le fichier de base

Le fichier est constitué d'une suite de pages contenant chacune un nombre entier d'enregistrements, qu'on supposera de taille fixe pour simplifier. Les enregistrements sont ordonnés selon K dans chaque page et les pages entre elles sont ordonnées selon K . On observe que certaines pages ne sont pas pleines.

b) L'index

Il est également constitué d'un ensemble de pages mais doté d'une structure plus complexe³⁷. Une page d'index contient des enregistrements particuliers appelés *entrées d'index*, constitués de deux champs (K , ptr). Le champ K contient une valeur de la clé et ptr est un pointeur contenant l'adresse d'une page, symbolisée par une pastille noire et une flèche qui pointe vers cette page. La valeur de K d'une entrée est la plus grande de celles qui apparaissent dans la page référencée. Les pages et leurs entrées sont structurées en un arbre (à 3 niveaux dans notre exemple).

Le niveau 1, à l'extrême gauche, est constitué d'une seule page. Le niveau m est constitué des pages référencées par les pointeurs des entrées du niveau $m - 1$. Le dernier niveau (n) possède des pointeurs vers des pages du fichier de base. Il comporte autant d'entrées que le fichier de base possède de pages. Les pages de l'index sont rangées dans un fichier séparé, ou dans une région dédiée du fichier de base, ou encore sont distribuées aléatoirement parmi pages du fichier de base.

37. Cette structure d'index est une variante de *B-arbres*, ou *arbres équilibrés* (en anglais *B-tree* ou *B+ tree*). Chaque nœud non terminal (sauf le dernier de la fratrie) d'un *B-arbre* d'ordre k possède de $\lceil k/2 \rceil$ à k enfants. Tous les chemins de la racine aux feuilles ont la même longueur.

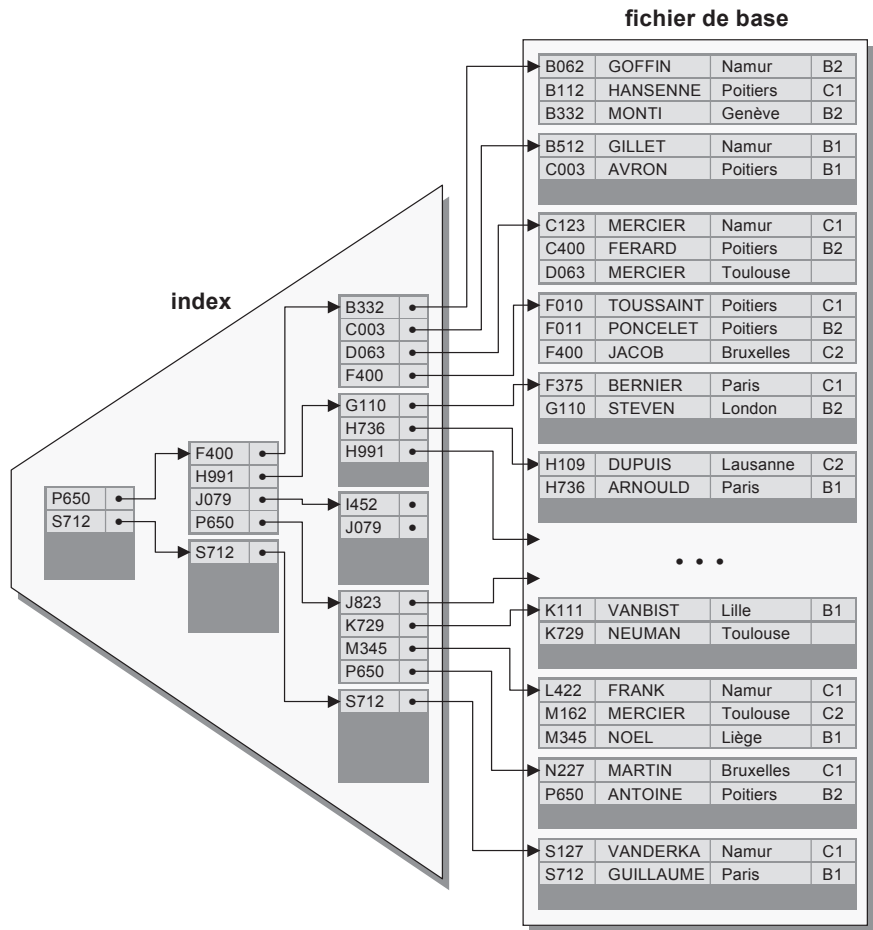


Figure 4.16 - Un fichier séquentiel indexé

4.8.2 Lecture et modification d'un fichier séquentiel indexé

a) Lecture de l'enregistrement

La lecture de l'enregistrement de valeur de clé $K = k$ via l'index s'effectue de la manière suivante (figure 4.17)³⁸. On connaît $K_{\max}$, la plus grande valeur de K présente dans le fichier et l'adresse p_1 de la page d'index de niveau 1. On recherche dans cette page la première entrée I telle que $I.K \geq k$. On lit ensuite la page pointée par le champ $I.ptr$ de cette entrée et on procède de même.

38. Dans cet algorithme, $A := E$ représente l'assignation à la variable de nom A de la valeur de l'expression E et $[p]$ désigne la page référencée par la valeur de p .

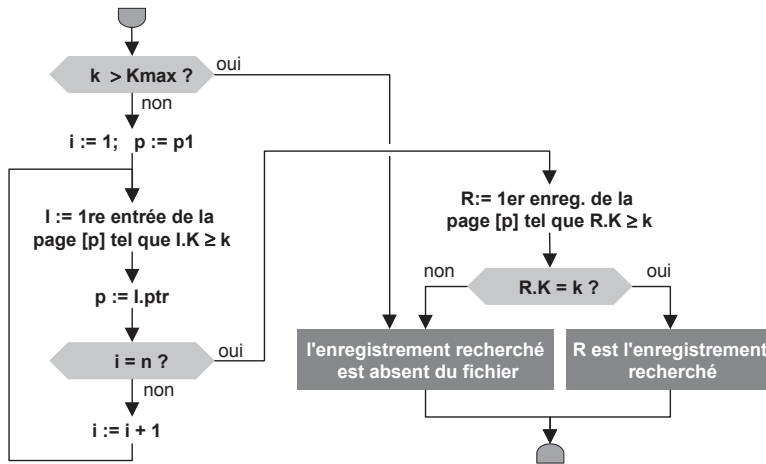


Figure 4.17 - Recherche de l'enregistrement R dont $R.K = k$, s'il existe

Lorsque la page l qui vient d'être examinée est de niveau n , alors $l.ptr$ référence la page du fichier de base dans laquelle l'enregistrement recherché, s'il existe, doit se trouver. Si on ne l'y trouve pas, c'est qu'il est absent du fichier mais c'est dans cette page qu'il aurait dû être rangé. L'accès à un enregistrement via l'index nécessite en théorie la lecture d'une page par niveau d'index + une page du fichier de base, soit $n + 1$ pages.

Un fichier séquentiel indexé offre **trois modes d'accès** (figure 4.18) :

- (a) l'accès via l'index par clé $K = k$ que nous venons d'examiner,
- (b) l'accès par intervalle de clé $K \in [k_1, k_2]$ qui consiste en un accès par clé $K = k_1$ suivi d'une lecture séquentielle des enregistrements tant que $K \leq k_2$
- (c) l'accès séquentiel trié, selon lequel les enregistrements sont lus dans leur ordre naturel.

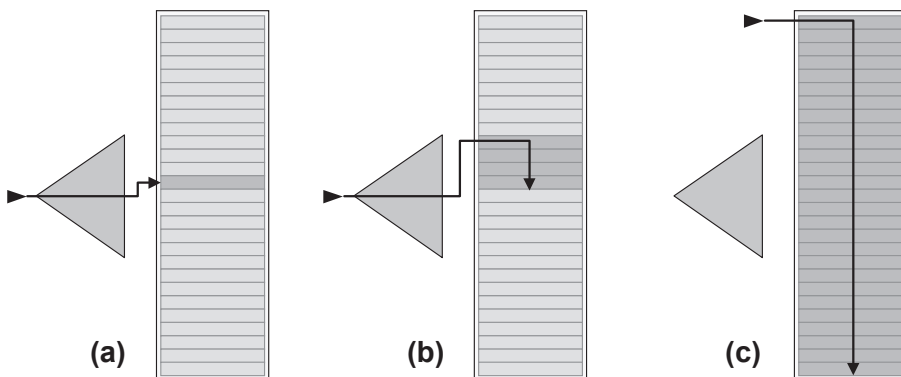


Figure 4.18 - Les trois modes d'accès à un fichier séquentiel indexé

b) Insertion d'un enregistrement

L'insertion d'un enregistrement de clé $K = k$ est réalisée comme suit. Au préalable, on recherche l'enregistrement dans le fichier comme décrit précédemment de manière à (1) s'assurer qu'il est absent et (2) connaître la page [p] dans laquelle il doit être inséré. De deux choses l'une : ou bien la page [p] contient assez d'espace pour accueillir l'enregistrement ou bien l'espace libre est insuffisant.

- Dans le premier cas, l'enregistrement est inséré dans la page de manière à conserver l'ordre selon K dans la page puis celle-ci est recopiée sur le disque. Le dernier enregistrement de cette page étant préservé, l'index reste inchangé.
- Dans le second cas, on recherche une page vide dans le fichier (ou on étend le fichier avec de nouvelles pages vides) et on répartit le contenu de la page [p] entre elle-même et la nouvelle page. Ce processus est appelé **éclatement** de la page [p]. La page remplie à (un peu plus de) 100 % donne naissance à deux pages remplies à au moins 50 %. Puisqu'une nouvelle page est créée, il faut insérer l'entrée correspondante dans une page du niveau n de l'index. Cette dernière opération est strictement identique à celle de l'insertion de l'enregistrement. Elle peut donc à son tour entraîner l'insertion d'une entrée dans une page de niveau $n - 1$ et, si nous n'avons pas de chance, ainsi de suite jusqu'au niveau 1. Si la page de niveau 1 éclate, alors ce niveau contient deux pages, ce qui entraîne la création d'un nouveau niveau 1. L'index comporte alors un niveau supplémentaire.

Le scénario selon lequel $K > K_{\max}$ correspond à l'insertion d'enregistrements de valeurs de K croissantes. Ce cas très fréquent doit être géré de manière différente, puisque l'algorithme de recherche de la page d'insertion échoue. Ici intervient un paramètre nouveau, τ_0 , le *taux de chargement initial*. Lorsqu'un enregistrement tel que $K > K_{\max}$ est inséré, on cherche à le ranger dans la dernière page du fichier de base. Si le taux d'occupation de celle-ci est inférieur à τ_0 , l'enregistrement est rangé dans cette page. Sinon, il est rangé dans une nouvelle page vide placée à la fin du fichier de base. Nous reparlerons de ce paramètre dans la section suivante.

Il existe plusieurs variantes de l'algorithme d'insertion décrit ci-avant. Selon l'une d'elles, lorsqu'une page est pleine, son contenu est réparti avec celui de sa voisine. Si celle-ci est également pleine, une page vide leur est ajoutée et le contenu des deux pages pleines est réparti entre les trois. Celles-ci sont donc remplies à 66% au lieu de 50%.

c) Suppression d'un enregistrement

La suppression d'un enregistrement libère de l'espace dans la page d'origine. Lorsque celle-ci est remplie à moins de 50 %, on rééquilibre son contenu avec celui de la page précédente ou suivante. Si cette nouvelle répartition ne permet pas de remplir ces deux pages à au moins 50 % alors les deux pages sont fusionnées : on déplace le contenu d'une page vers l'autre et on la supprime. Cette contraction du fichier de base entraîne la suppression d'une entrée dans une page du niveau n de l'index, lequel peut aussi subir une telle contraction ainsi que, en cascade, les niveaux en amont. Lorsque le niveau 2 ne contient plus qu'une seule page, on peut supprimer le niveau 1. L'index comporte alors un niveau de moins.

d) *Chargement d'un fichier séquentiel indexé*

Il est fréquent qu'un fichier séquentiel indexé soit, préalablement à son exploitation, garni d'une série d'enregistrements fournis par un fichier source, généralement séquentiel. Leur transfert entre ces deux fichiers est appelé *chargement initial* du fichier séquentiel indexé. Ce chargement ne s'effectue pas par l'insertion des enregistrements dans un ordre aléatoire, ce qui serait trop coûteux, mais à partir du fichier source **trié** sur la clé K. Lors de ce chargement, les pages ne sont pas remplies à 100 %, mais au taux τ_0 , de manière à laisser de la place libre dans chacune d'elles en vue de retarder le phénomène d'éclatement lors des insertions aléatoires ultérieures. Nous discuterons plus loin de la valeur optimale de ce paramètre, mais disons déjà que 0,7 à 0,8 constituent des valeurs favorables. En revanche si les insertions se feront **toujours** par valeurs croissantes de K, un taux de 1 est optimal.

Le chargement est une opération très rapide consistant à copier un fichier séquentiel dans un autre fichier séquentiel. Le coût de constitution de l'index est aussi un processus séquentiel mais dont le coût est négligeable.

e) *Réorganisation d'un fichier séquentiel indexé*

Comme nous le verrons dans la section suivante, les ajouts et suppressions successives dans un fichier séquentiel indexé vont conduire à une dégradation progressive de ses performances, rendant nécessaire la **réorganisation** de ce fichier. Cette opération réarrange les enregistrements dans les pages de manière à éliminer les sources d'inefficacité. Cette réorganisation consiste à créer une nouvelle version du fichier par chargement des enregistrements de la version courante (qui est, ne l'oublions pas, un *fichier séquentiel trié*) selon les principes décrits au point (d) ci-dessous.

4.8.3 Caractéristiques et performances d'un fichier séquentiel indexé

Les algorithmes d'insertion et de suppression d'enregistrements préservent l'ordre des enregistrements et des entrées de l'index. Ils garantissent également aux pages du fichier de base et de l'index (sauf la dernière) un **taux d'occupation au moins égal à 50 %**. Lors de la création du fichier, il est habituel de charger chaque page à un taux d'occupation initial, typiquement de l'ordre de 60 à 80 %. En cours d'exploitation, ce taux d'occupation va évoluer selon la taille des pages et le nombre d'opérations de mise à jour (insertions/suppressions). Comme nous allons le voir, il conditionnera le calcul des volumes et des temps de lecture et de modification des données. On ne reprendra ici que la synthèse des calculs dont le détail est reporté à l'annexe A4.

a) *Les volumes*

L'estimation du volume d'un fichier séquentiel indexé ne pose pas de problèmes majeurs. On a en effet :

$$N_p = N_{pb} + N_{pi}$$

où **Npb** et **Npi** sont les nombres de pages respectivement du fichier de base et de l'index. Dans les calculs ci-dessous, on ignore les éventuelles pages vides.

• Volume du fichier de base **Npb**

Le fichier de base contient **Nr** enregistrements de taille **Lr** rangés dans une suite de **Npb** pages de taille **Lp**. Pour simplifier, on admet qu'un enregistrement est entièrement compris dans une page. On connaît **tb**, le taux d'occupation moyen des pages de base à un moment donné. Soient **Mrpp** et **Nrpp** les nombres maximum et moyen d'enregistrements par page :

$$\mathbf{Mrpp} = \lfloor \mathbf{Lp} / \mathbf{Lr} \rfloor$$

$$\mathbf{Nrpp} = \mathbf{tb} \times \mathbf{Mrpp}$$

Ceci nous permet de calculer la taille du fichier de base :

$$\mathbf{Npb} = \lceil \mathbf{Nr} / \mathbf{Nrpp} \rceil$$

• Volume de l'index **Npi**

On connaît **ti**, le taux d'occupation moyen des pages d'index à un moment donné. La longueur d'une entrée d'index est **Li**, celle du champ clé est **Lk** et celle d'un pointeur de page **Lptr**. Soient **Mipp** et **Nipp** les nombres maximum et moyen d'entrées d'index par page. On a :

$$\mathbf{Li} = \mathbf{Lk} + \mathbf{Lptr}$$

$$\mathbf{Mipp} = \lfloor \mathbf{Lp} / \mathbf{Li} \rfloor$$

$$\mathbf{Nipp} = \mathbf{ti} \times \mathbf{Mipp}$$

On observe que la taille de l'index est généralement très faible et que celle du dernier niveau (**Npi(n)**) y est largement prédominante. Ce niveau contenant une entrée par page de base, on peut donc admettre :

$$\mathbf{Npi} \cong \mathbf{Npi(n)} = \lceil \mathbf{Npb} / \mathbf{Nipp} \rceil$$

• Nombre de niveaux d'index **n**

Chaque niveau de l'index comporte **Nipp** fois plus de pages que le précédent. On a donc, en évaluant la taille du niveau de rang **n** :

$$\mathbf{Npi(n)} = \lceil \mathbf{Npb} / \mathbf{Nipp} \rceil \cong \mathbf{Nipp}^{n-1}$$

D'où :

$$\mathbf{Npb} \cong \mathbf{Nipp}^n$$

dont on tire, **b** étant une base quelconque (2, e, 10) :

$$\mathbf{n} = \lceil \log_{\mathbf{Nipp}} \mathbf{Npb} \rceil = \lceil \log_{\mathbf{b}} \mathbf{Npb} / \log_{\mathbf{b}} \mathbf{Nipp} \rceil$$

• Exemple

Soit un fichier de **Nr** = 2 000 000 enregistrements de **Lr** = 400 octets, dont **Lk** = 40 pour la clé K. Les pages sont de **Lp** = 4 Ko remplies à **tb** = **ti** = 0,8. Le fichier de base est formé de **Npb** = 250 000 pages et l'index de **Npi** = 3 408 pages. L'index comporte

trois niveaux, respectivement de 1, 46 et 3 361 pages. L'estimation $N_{pi} \equiv N_{pi(m)}$ induit une erreur négligeable de 47 pages sur 253 408, soit moins de 0,02%.

b) Les temps de lecture

Nous étudierons successivement la lecture séquentielle, la lecture par index et la lecture par intervalle.

• Lecture séquentielle des enregistrements

Lorsque le fichier est jeune, les pages successives du fichier de base sont physiquement contiguës. On a montré (4.6) que le temps de lecture séquentielle t_{lsf} du fichier, dans l'hypothèse réaliste d'une lecture anticipée des 112 pages d'une piste, était de :

$$t_{lsf} = t_{lsf2} = N_p \times t_{ls1} = 46 \text{ s}$$

Chaque éclatement de page provoque 2 ruptures de continuité car la nouvelle page peut être très éloignée de la page qui éclate. Cependant, grâce à la lecture anticipée, la pénalité ne sera que d'une rupture : lors de la lecture des pages 99, 100, 101, 201 556, 103, 104, ..., les pages 99, 100, 101, 102, 103, 104, ... sont en général présentes dans le tampon lors de l'accès à la page 201 556. En admettant qu'un nombre h de pages ont éclaté, il faut remplacer dans le calcul du coût de lecture h lectures séquentielles par h lectures aléatoires. Nous mesurons ce phénomène par le **taux de rupture** $\pi_r = h / N_p$, indiquant la proportion de pages *hors séquence*, qui exigeront une lecture aléatoire. $\pi_r = 0$ correspond à un fichier jeune d'un seul tenant et $\pi_r = 1$ implique que toutes les pages sont disposées aléatoirement sur le disque. Il vient donc, en désignant par t_{lr} le temps d'accès à une page hors séquence :

$$t_{lsf} = (1 - \pi_r) \times N_{pb} \times t_{ls1} + \pi_r \times N_{pb} \times t_{lr}$$

Par exemple, si 20 % des pages ont éclaté ($\pi_r = 0,2$), le temps de lecture du fichier de base est $t_{lsf} = (1 - 0,2) \times 250\,000 \times 0,184 + 0,2 \times 250\,000 \times 12,3 = 652 \text{ s}$, soit 14 fois plus que le temps de lecture d'un fichier jeune, en admettant $t_{lr} = t_{la1}$ ³⁹ !

• Lecture par index d'un enregistrement

En théorie, le temps de lecture d'un enregistrement dont $K = k$ est celui de la lecture d'une page dans chacun des n niveaux d'index suivie de celle d'une page de base, soit dans notre exemple $t_{ik1} = (3 + 1) \times t_{la1} = 49,2 \text{ ms}$. L'index permet donc 20 lectures/s. C'est ici qu'intervient le tampon. Supposons que nous maintenions le premier niveau d'index (1 page) dans le tampon. Pour un coût négligeable nous obtenons un gain de 12,3 ms. On a donc $t_{ik1} = 36,9 \text{ ms}$, correspondant à une vitesse de 27 lectures/s. En maintenant dans le tampon les pages des niveaux 1 et 2, soit $1 + 46$ pages d'index, l'accès à l'enregistrement est limité à 2 pages, soit 24,6 ms, pour une vitesse de lecture de 40,6 lectures par seconde. Poussons le raisonnement à la limite : on charge la totalité de l'index dans le tampon. La consommation

39. Cette valeur sera sans doute excessive pour un fichier n'occupant qu'une faible proportion du disque.

d'espace est évidemment importante (3 408 pages, soit 13,3 Mo), mais le temps d'accès par l'index n'est plus que de 12,3 ms. (81 lectures/s).

• Lecture par index des enregistrements d'un intervalle

On rappelle que l'accès par intervalle de clé $K \in [k_1, k_2]$, consiste en un accès par clé $K = k_1$ (en réalité au premier enregistrement dont $K \geq k_1$) suivi d'une lecture séquentielle des enregistrements dont $K \leq k_2$. On note π_{ik} la probabilité de réalisation de la condition d'intervalle. $n_{rik} = \pi_{ik} \times N_r$ enregistrements tombent dans cet intervalle. Ceux-ci occupent n_{pik} pages consécutives⁴⁰ :

$$n_{pik} = \lfloor (n_{rik} - 1) / N_{rpp} + 0,5 \rfloor + 1$$

Le temps de lecture de ces enregistrements est celui de la lecture d'une séquence de pages consécutives. Si nous ignorons le phénomène de rupture, il vient, pour un accès séquentiel avec lecture anticipée :

$$t_{lik} = n_{pik} \times t_{ls1}$$

En tenant compte des ruptures, on affine le calcul comme suit :

$$t_{lik} = t_{la1} + (1 - \pi_r) \times n_{pik} \times t_{ls1} + \pi_r \times n_{pik} \times t_{lr}$$

À ces valeurs de t_{lik} , il convient d'ajouter le temps de parcours initial de l'index.

Il existe une variante intéressante de l'accès par intervalle, l'accès par *index préfixe*⁴¹. Un index sur les champs (A1, A2) permet d'accéder aux enregistrements qui satisfont la condition *where* A1 = k. Nous en reparlerons à la section 20.4.3.

c) Les temps de modification des données

Nous estimerons le coût de chacune des trois primitives de modification des données. Certaines opérations impliqueront, dans certaines circonstances, la modification de l'index. Ces cas étant rares, nous les ignorerons. Pour le détail des calculs, on se reportera à l'annexe A4.

• Insertion d'un enregistrement

L'insertion consiste à accéder à la page de base de l'enregistrement (t_{1k}), selon la procédure d'accès par index, puis à recopier la page sur disque (t_{la1}). Si l'éclatement de cette page est nécessaire, alors il faut en outre rechercher et lire une page libre (t_{la1}), puis recopier les deux pages sur disque après répartition ($2 \times t_{la1}$). Il faut enfin mettre à jour l'index (t_{la1}). Si on note τ_{split} la proportion des insertions qui provoquent un éclatement, il vient :

$$t_{ins} = t_{1k} + (1 + 3 \times \tau_{split}) \times t_{la1}$$

40. Calcul approché pour $n_{rik} > 0$. Chaque enregistrement peut constituer le début d'un intervalle. On admet qu'en moyenne ce début est au milieu d'une page.

41. *Index skip scan*, selon la terminologie d'Oracle.

• **Suppression d'un enregistrement**

La suppression consiste à accéder à la page de l'enregistrement (qu'on suppose déjà présente dans le tampon, sinon on compte t_{1k}), à y effacer l'enregistrement puis à la recopier sur disque (t_{1a1}). On ignore le cas particulier de la suppression du dernier enregistrement de la page (probabilité $1 / N_{rpp}$), qui demanderait la modification de l'index. Si un rééquilibrage est nécessaire, on lit la page suivante (t_{1a1}), on recopie les deux pages ($2 \times t_{1a1}$) puis on ajuste l'index (t_{1a1}). Si une fusion est nécessaire, on lit la page suivante (t_{1a1}), on recopie la page restante (t_{1a1}), on libère la page vide (t_{1a1}) et son entrée est supprimée de l'index (t_{1a1}). Notons τ_{bal} la proportion des opérations de suppression qui entraînent un rééquilibrage de pages et τ_{merge} la proportion des opérations de suppression qui entraînent une fusion de pages. On peut dès lors calculer le temps moyen d'une opération de suppression :

$$t_{del} = (1 + 3 \times \tau_{bal} + 3 \times \tau_{merge}) \times t_{1a1}$$

• **Modification d'un enregistrement**

La modification d'un champ appartenant à la clé de l'index primaire induit la modification de la valeur de l'identifiant de l'enregistrement. Comme la position de ce dernier dépend directement de cette valeur, il est nécessaire de déplacer cet enregistrement. Du point de vue de son coût, cette opération revient à supprimer puis à recréer cet enregistrement⁴².

d) Synthèse des calculs

La figure 4.19 synthétise les différents modèles de calcul. Certains d'entre eux ont été traduits dans la feuille de calcul ISAM.xls disponible sur le site de l'ouvrage.

Calcul d'un fichier séquentiel indexé

Taille du fichier

$$N_p = N_{pb} + N_{pi}$$
$$N_{pb} = \lceil N_r / N_{rpp} \rceil$$
$$N_{rpp} = \tau_b \times M_{rpp}$$
$$M_{rpp} = \lfloor L_p / L_r \rfloor$$

$$N_{pi} \cong \lceil N_{pb} / N_{ipp} \rceil$$
$$N_{ipp} = \tau_i \times M_{ipp}$$
$$M_{ipp} = \lfloor L_i / L_p \rfloor$$
$$L_i = L_k + L_{ptr}$$

Temps de lecture séquentielle du fichier

$$t_{lsf} = (1 - \tau_r) \times N_{pb} \times t_{ls1} + \tau_r \times N_{pb} \times t_{lr}$$

Temps de lecture d'un enregistrement via l'index

$$t_{1k1} = (n - m + 1) \times t_{1a1} \quad (m \text{ niveaux d'index dans le tampon})$$

42. Si cet enregistrement est une ligne d'une table que référencent d'autres lignes, le SGBD va exécuter des opérations additionnelles de préservation de l'intégrité référentielle, telle que la modification des clés étrangères (on update cascade). L'évaluation du coût de la suppression d'un enregistrement en situation est dès lors plus complexe mais se réduit à une suite d'opérations élémentaires.

Temps de lecture des enregistrements d'un intervalle (proportion $\mathbf{p_{ik}}$) via l'index

$$\mathbf{n_{rik}} = \mathbf{p_{ik}} \times \mathbf{N_r}$$

$$\mathbf{n_{pik}} = \lfloor (\mathbf{n_{rik}} - 1) / \mathbf{N_{rpp}} + 0,5 \rfloor + 1$$

$$\mathbf{t_{lik}} = (\mathbf{n} - \mathbf{m} + 1) \times \mathbf{t_{la1}} + (1 - \mathbf{\pi_r}) \times \mathbf{n_{pik}} \times \mathbf{t_{ls1}} + \mathbf{\pi_r} \times \mathbf{n_{pik}} \times \mathbf{t_{lr}}$$

Temps d'insertion d'un enregistrement

$$\mathbf{t_{ins}} = \mathbf{t_{lk1}} + (1 + 3 \times \mathbf{\tau_{split}}) \times \mathbf{t_{la1}}$$

Temps de suppression d'un enregistrement

$$\mathbf{t_{del}} = (1 + 3 \times (\mathbf{\tau_{bal}} + \mathbf{\tau_{merge}})) \times \mathbf{t_{la1}}$$

Temps de modification de la clé d'un enregistrement

$$\mathbf{t_{updi}} = \mathbf{t_{del}} + \mathbf{t_{ins}}$$

Figure 4.19 - Calculs relatifs aux fichiers séquentiels indexés

e) Évaluation de τ_b , τ_i , π_r , τ_{split} , τ_{bal} et τ_{merge}

Le calcul des performances nécessite la connaissance de différentes mesures du comportement du fichier, en particulier π_r , τ_b , τ_i , τ_{split} , τ_{bal} et τ_{merge} . L'étude mathématique de ce comportement étant très complexe, nous utiliserons un *simulateur* qui nous fournira ces mesures lors de l'application d'une suite d'opérations d'insertion et de suppression à un tableau de pages en mémoire centrale modélisant en réduction un fichier réel. L'annexe A4 contient une description détaillée du simulateur et une discussion de ses résultats. Nous en reprendrons ici quelques conclusions.

La gestion du fichier de base et celle de l'index étant identique, on ne distinguera pas l'étude de τ_b de celle de τ_i , que nous désignerons collectivement par τ . Une simulation fournit (notamment) le taux d'occupation τ et les nombres d'événements de restructuration n_{split} , n_{bal} et n_{merge} pour chaque pas de simulation, chacun comportant un certain nombre d'insertions et de suppressions. Pour chaque simulation, on précise $\mathbf{N_{p0}}$ le nombre initial de pages, $\mathbf{M_{rpp}}$ la taille des pages, τ_0 le taux d'occupation initial, $\%ins$ la proportion d'insertions parmi les opérations, $\mathbf{N_{op}}$ le nombre d'opérations (*nombre de pas* $\times$ *nombre d'opérations par pas*).

La figure 4.20 montre graphiquement⁴³ le résultat d'une série de simulations pour les valeurs de $\mathbf{N_{p0}} = 5\,000$, $\mathbf{M_{rpp}} = 40$, $\tau_0 = 0,8$ ($\mathbf{N_{rpp}} = 32$ au départ) et $\%ins$ de 50 à 100 par pas de 10. La valeur $\%ins = 50$ correspond à un fichier dont le nombre d'enregistrements est constant alors que $\%ins = 100$ représente un fonctionnement en croissance pure (pas de suppressions). Chaque pas représente 2 000 opérations, dont, pour $\%ins = 80$, 1 600 insertions et 400 suppressions en moyenne.

Les résultats des simulations de l'annexe A4 conduisent aux observations suivantes.

- Pour toutes les valeurs $\mathbf{k} > 0$, les simulations $[\mathbf{k} \times \mathbf{N_{p0}}, \mathbf{M_{rpp}}, \mathbf{N_{rpp0}}, \%ins, \mathbf{k} \times \mathbf{N_{op}}]$ donnent les mêmes résultats pour τ . Pour étudier le comportement d'un fichier de 1 000 000 pages pour 100 000 000 opérations, il suffit d'exécuter une simulation pour $\mathbf{N_{p0}}=5\,000$ et $\mathbf{N_{op}}=500\,000$].

43. Une version des graphiques de ce chapitre plus aisée à consulter (en haute définition et en couleur) est disponible à l'annexe A4.

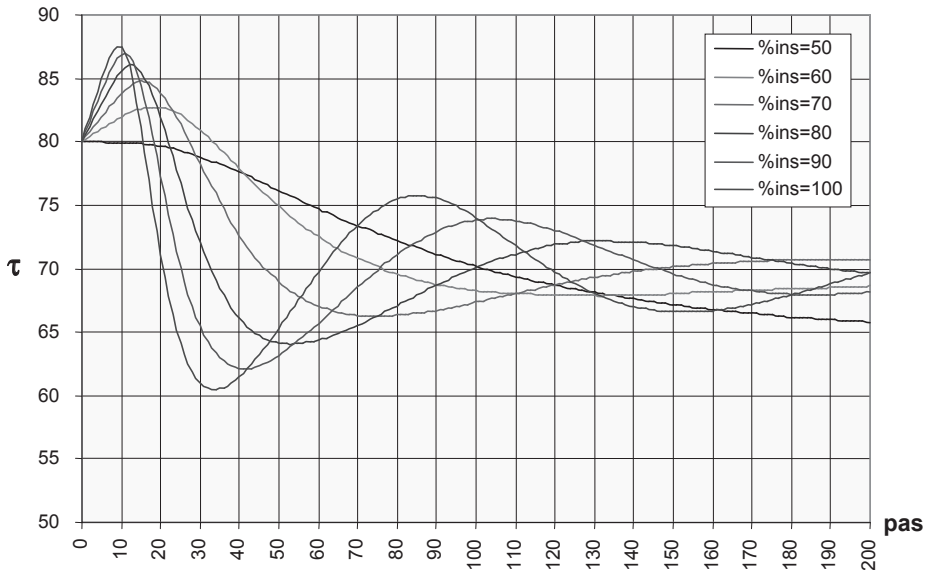


Figure 4.20 - Évolution (en %) du taux d'occupation τ d'un fichier
[$Np0 = 5\,000$; $Mrpp = 40$; $\tau_0 = 0,8$; $\%ins = 50$ à 100 ; $Nop = 400\,000$]

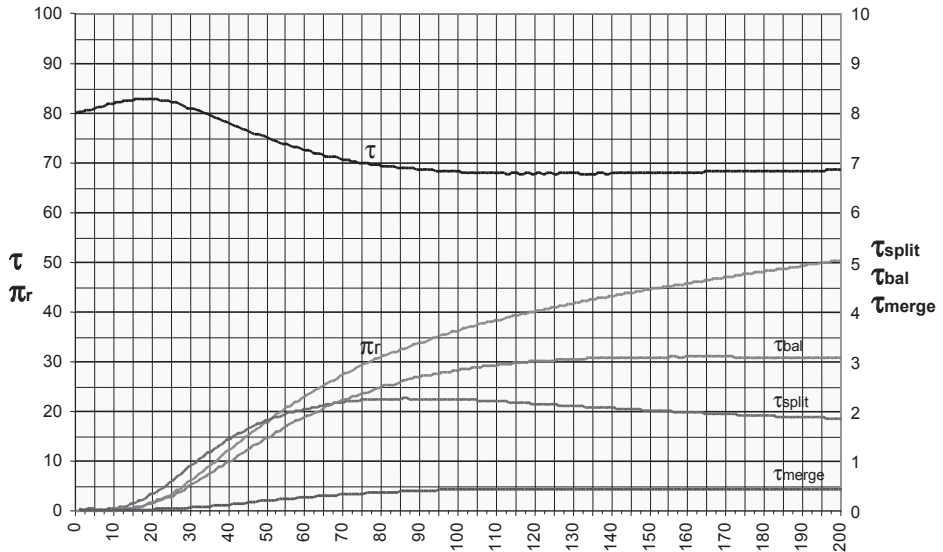


Figure 4.21 - Évolution (en %) des taux τ , π_r , τ_{split} , τ_{bal} et τ_{merge}
[$Mp0 = 6\,250$; $Mrpp = 40$; $\tau_0 = 0,8$; $\%ins = 60$; $Nop = 400\,000$]

- Au début du fonctionnement du fichier (partie gauche), certaines courbes montrent une grande variation, qui peut aller de 53% à 100% (pour $Mrpp = 100$ et $\%ins = 100$ par exemple), soit pratiquement du simple au double. Cette variation est une fonction croissante de $Mrpp$ et de $\%ins$.

- Lorsqu'on prolonge la simulation, on constate que les valeurs de τ se stabilisent et tendent vers une valeur asymptotique, dépendant de **Mrpp** et de **%ins** mais pas de τ_0 . (63% pour **Mrpp** = 100 et **%ins** = 50; 71% pour **Mrpp** = 10 et **%ins** = 100)

Les courbes de la figure 4.21 montrent l'évolution de τ et des événements de restructuration mesurés par π_r , τ_{split} , τ_{bal} et τ_{merge} pour un des cas de la figure 4.20 (**%ins** = 60). On observe que (conclusions à adapter pour chaque jeu de paramètres) :

- le taux de rupture π_r , nul pendant une première période, se met à croître vers des valeurs très importantes (au-delà de 50%), néfastes pour la lecture séquentielle;
- le taux d'éclatement τ_{split} se stabilise aux alentours de 2%; en suppression, les fusions τ_{merge} sont très rares (0,5%) alors que les rééquilibrages τ_{bal} se stabilisent autour de 3%. L'impact des restructurations sur les coûts de modification des données est donc assez faible.
- ces taux sont d'autant meilleurs que la taille des pages est importante : τ augmente et les autres taux diminuent.

f) Conclusions et recommandations

L'observation de l'évolution de ces mesures nous suggère les recommandations suivantes.

- La valeur optimale du taux de chargement initial τ_0 tourne autour de 0,7. Un taux de 1 n'est admissible que si le fichier ne fait l'objet d'aucune insertion ou si les enregistrements sont insérés par valeurs de clé croissantes.
- Les performances du fichier sont d'autant meilleures que la taille des pages **Mrpp** est importante, tant pour le fichier de base que pour l'index.
- D'une manière générale, certaines mesures essentielles (τ et π_r notamment) deviennent très défavorables à partir du pas 50, voire plus tôt. On en conclut qu'une **réorganisation** du fichier devrait être envisagée bien avant que le nombre d'opérations de modification (insertions et suppressions) se rapproche de 40 à 50% du nombre d'enregistrements au départ.

4.8.4 Étude de cas

Soit un fichier séquentiel indexé chargé initialement à 80% de 2 000 000 enregistrements de 200 octets. Les pages sont de 8 Ko et le taux d'insertion est de 60% (**Nr0** = 2 000 000; **Lr** = 200; **Lp** = 8 Ko; τ_0 = 0,8; **%ins** = 60%). On effectue 200 000 modifications par mois. On cherche à calculer le volume du fichier de base et les temps de lecture séquentielle, d'insertion et de suppression après 4 mois de fonctionnement.

Il vient, au démarrage : **Mrpp** = 40; **Nrpp0** = 32; **Np0** = **Nr0** / **Nrpp0** = 62 500. On utilise les informations de la figure 4.21, obtenues par simulation d'une durée de **Nop** = 2 x **Nr0**. Chaque pas représente **Nr0** / 100 opérations, soit dans notre exemple 20 000. Au terme des 4 mois, 800 000 opérations auront été effectuées, ce qui correspond au pas 40. À ce pas, les courbes nous donnent les informations suivantes : τ = 78% (donc **Nrpp** = 31,2); π_r = 12,1%; τ_{split} = 1,42%; τ_{bal} = 0,97%; τ_{merge} = 0,11%. Ces 800 000 opérations créeront 480 000 enregistrements et en supprimeront

320 000, soit une augmentation de 160 000. Les 2 160 000 enregistrements occuperont $N_p = N_r / N_{rpp} = 2\,160\,000 / 31,2 = 69\,231$ pages (= 160 cylindres).

Pour calculer le temps de lecture séquentielle il faut évaluer t_{lr} , le temps de lecture d'une page hors séquence. Le fichier occupant 0,01% du disque, un coût de 12,3 ms serait excessif. Adoptons un temps de 6,25 ms⁴⁴.

- Lecture séquentielle sans ruptures : $t_{lsf} = N_p \times t_{ls1} = 69\,231 \times 0,184 = 15,75$ s
- Lecture séquentielle ($\pi_r = 12,1$) : $t_{lsf} = (1 - \pi_r) \times N_p \times t_{ls1} + \pi_r \times N_p \times t_{lr} = 63,6$ s
- Insertion ($t_{11k} = 2 \times t_{1a1}$) : $t_{ins} = t_{11k} + (1 + 3 \times \tau_{split}) \times t_{1a1} = 37,5$ ms
- Suppression : $t_{del} = (1 + 3 \times (\tau_{bal} + \tau_{merge})) \times t_{1a1} = 12,7$ ms (éventuellement ajouter $t_{11k} = 24,6$, pour un total de 37,3 ms.

4.9 ORGANISATION CALCULÉE

L'index décrit dans la section précédente agit comme une fonction $p = f(k)$ qui transforme une valeur de clé k en une adresse de page p en utilisant une **table de correspondance mémorisée**. On pourrait imaginer que cette fonction f soit exprimée sous forme d'un **calcul**, d'où l'appellation d'*organisation calculée* (figure 4.22).

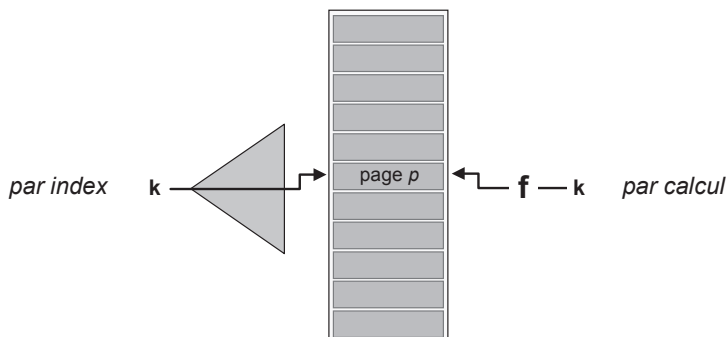


Figure 4.22 - Deux modes de conversion valeur de clé k / adresse de page p

4.9.1 Principes de l'accès calculé

Un premier exemple permet de concrétiser cette idée. Considérons la table LIVRE de la figure 2.10, dont l'identifiant NUMERO est constitué d'un nombre de 4 chiffres au maximum. Cette table peut donc contenir jusqu'à 10 000 lignes de 200 octets. On désire stocker ces lignes sous forme d'enregistrements dans un fichier F de la manière suivante.

Le fichier contient $N_p = 500$ pages de longueur $L_p = 4$ Ko, pouvant chacune accueillir jusqu'à $M_{rpp} = 20$ enregistrements (selon une des techniques b ou d de la figure 4.9). Nous décidons de stocker chaque ligne dans la page dont le numéro p est calculé à partir de la valeur de son identifiant **NUMERO** selon la formule :

44. Déplacement du bras de 2 ms, délai rotationnel de 4,17 ms, transfert de 0,074 ms.

$$p = \lfloor \text{NUMERO} / \text{Mrpp} \rfloor$$

Ainsi, les enregistrements dont NUMERO est compris entre 0 et 19 seront rangés dans la page 0, ceux dont NUMERO est compris entre 20 et 39 dans la page 1, et ainsi de suite. La même procédure est utilisée pour retrouver un enregistrement : on sait que l'enregistrement relatif au livre de numéro 2 627 a été rangé dans la page $\lfloor 2\,627 / 20 \rfloor$, soit 131. Il suffit de lire cette page, puis de parcourir son contenu pour retrouver l'enregistrement recherché ou pour constater qu'il est absent.

On observe que (1) cette technique permet de stocker un enregistrement quelconque en **une** lecture de page et **une** réécriture (24,6 ms), (2) elle permet de retrouver un enregistrement quelconque en **une** lecture de page (12,3 ms), (3) elle n'utilise pas de structure de données additionnelle telle qu'un index et (4) elle est d'une extrême simplicité. Comparée à l'organisation séquentielle indexée, elle semble présenter de sérieux avantages.

Malheureusement, les choses ne seront pas toujours aussi favorables. L'espace du fichier est prévu pour autant d'enregistrements qu'il existe de valeurs possibles pour le champ servant de clé (ici NUMERO). Si une petite partie seulement de ces valeurs est effectivement utilisée, il s'ensuit un énorme gaspillage de place. Dans la réalité, les numéros seront souvent attribués de manière discontinue et irrégulière (1 029, 1 041, 1 042, 1 043, 1 187, etc.). Dans ce cas, les valeurs se concentrent sous forme d'agrégats denses (1 041, 1 042, 1 043) séparés par des espaces vides (rien de 1 044 à 1 186). Même problème si des enregistrements sont supprimés. Il en résulte que certaines pages ne contiendront que peu d'enregistrements, voire aucun. De plus, les valeurs de l'identifiant ne sont pas toujours des entiers. Les valeurs de NCLI de la table CLIENT (figure 2.8) en sont une bonne illustration : B062, B112, B332, C123, etc. Comment dans un tel cas attribuer un numéro de page à une chaîne de caractères quelconques ? La solution réside dans la structure de la fonction *f* qui calcule l'adresse de la page.

4.9.2 Fonction de calcul d'adresse

La fonction *f*, qui produit un numéro de page *p* à partir d'une valeur *k* de la clé *K*, a trois objectifs, réalisés par trois fonctions partielles $f(k) = f_3(f_2(f_1k))$ (figure 4.23). La première, *f*₁, est la fonction de **numérisation** qui a pour but de transformer la valeur *k* en un nombre. La deuxième, *f*₂, est la fonction de **hachage**, chargée d'homogénéiser l'ensemble des nombres obtenus. La troisième, *f*₃, est la fonction de **cadrage**, qui ajuste la valeur obtenue à la taille du fichier.

- *f*₁ transforme *k*, qui en toute généralité est une chaîne de caractères, en un nombre entier **p1**. L'une des techniques les plus rapides consiste à considérer la valeur *k* comme une chaîne de bits, puis à découper celle-ci en fragments de longueur fixe (par exemple 32 bits) et enfin à appliquer l'opérateur XOR⁴⁵ sur les fragments.
- *f*₂ casse (ou *hache*, d'où le nom de *fonction de hachage*) les agrégats et les éventuels *patterns* que présente l'ensemble des valeurs de *K*, et qui sont partiellement

45. La définition de cet opérateur est rappelée dans l'annexe A24.

transmis aux valeurs de $p1$. Elle produit un entier $p2$. Ce type de fonction a été largement étudié et de nombreuses variantes ont été proposées. Citons en deux.

- Le *pliage* consiste à couper la valeur de $p1$ en deux parties égales $p11$ et $p12$, à inverser les chiffres (binaires ou décimaux) de $p12$ donnant $p21$ puis à additionner $p11$ à $p21$. Par exemple, $p2 = f2(1024365879) = 10243 + 97856 = 1080099$.
- La technique du *milieu du carré* élève $p1$ au carré puis supprime les premiers et les derniers chiffres. Pour la même valeur 1024365879 de $p1$ adoptée précédemment, on obtiendrait le carré 1049325454059442641, dont on conserverait les 7 chiffres centraux : $p2 = 5454059$.

Le nombre $p1$ peut faire l'objet de plusieurs opérations successives de ce type.

- $f3$ projette $p2$ dans l'espace des numéros de pages disponibles. La fonction *modulo* (reste de la division entière) est le plus souvent utilisée : $p = p2 \bmod Np$.

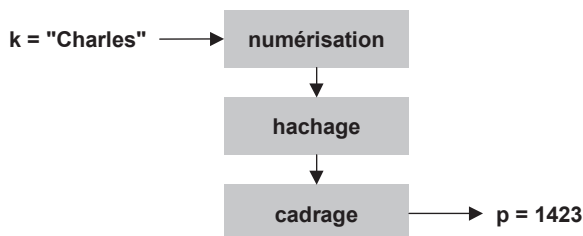


Figure 4.23 - Architecture d'une fonction de calcul d'adresse $p = f(k)$

Exemple

Où doit-on ranger l'enregistrement dont l'identifiant est "Charles" dans un fichier de 5 000 pages ?

- En considérant les codes ASCII des caractères, la chaîne de caractères "Charles" correspond à la chaîne de 56 bits :

01000011 01101000 01100001 01110010 01101100 01100101 01110011.

En appliquant un XOR entre les tranches de 32 bits de cette chaîne (ici, les 32 premiers et les 24 suivants, étendus à 32 bits par des 0), on obtient le nombre :

00101111 00001101 00010010 01110010

soit 789385842 en décimal. On a donc :

$f1(\text{"Charles"}) = 789385842$.

- On adopte pour le hachage la technique du pliage, appliquée une seule fois. Elle nous donne :

$f2(789385842) = 78938 + 02485 = 81423$.

- La fonction de cadrage ajuste cette valeur à un espace de 5.000 pages :

$f3(81423) = 81423 \bmod 5000 = 1423$

On a donc :

$f(\text{"Charles"}) = 1423$

L'enregistrement sera rangé, et en principe retrouvé, dans la page $p = 1\,423$.

4.9.3 Gestion des débordements

Si ce type de fonction permet d'associer un numéro de page à n'importe quelle chaîne de caractères, elle entraîne en revanche un nouveau problème : quelle que soit la fonction f , pour un ensemble arbitraire de valeurs de K , rien ne garantit qu'un même numéro de page ne sera pas attribué à plus de $Mrpp$ enregistrements. Le phénomène selon lequel une page reçoit plus de $Mrpp$ enregistrements est dénommé **débordement**.

Le problème du débordement peut être résolu de plusieurs manières. La fonction f calcule l'**adresse de base** de l'enregistrement (notée p). Les enregistrements qui ne peuvent y être stockés, faute de place, sont stockés dans une autre page, d'adresse effective p' , soit du même fichier, soit d'un espace indépendant dédié.

Nous décrivons et analyserons une des techniques les plus répandues⁴⁶. Le fichier est composé de deux parties : l'*espace de base* qui est de taille fixe (Npb pages, dites *de base*) et l'*espace de débordement* qui est extensible (figure 4.24).

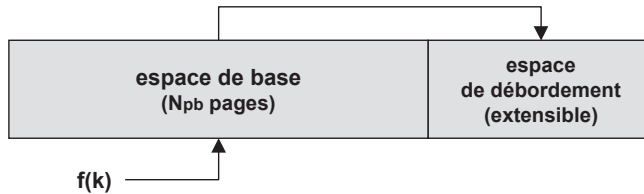


Figure 4.24 - Structure typique d'un fichier calculé

La fonction f donne l'adresse p d'une page de base dans laquelle l'enregistrement doit être stocké. Si la page $[p]$ offre de l'espace libre, on y stocke cet enregistrement. Sinon, on réserve une page dans l'espace de débordement, on y stocke l'enregistrement et on ajoute à la page de base un pointeur référençant cette page, dite *de débordement*. Plus tard, dès que cette dernière est pleine à son tour et doit accueillir un nouvel enregistrement, on procède de la même manière : une nouvelle page de débordement, référencée par la précédente, est réservée pour accueillir cet enregistrement. Les enregistrements sont donc rangés dans des chaînes de pages dont la première est une page de base et les suivantes sont des pages de débordement⁴⁷.

Exemple

La figure 4.25 montre un fragment de l'espace de base comprenant les pages 82 à 85 et un fragment de l'espace de débordement (pages 907 à 910). On y observe que f a attribué l'adresse de base 82 aux enregistrements A, B, C, D, P et Q. Les quatre premiers ont été stockés dans leur page de base tandis que, pour les deux autres, une page de débordement (907) a été allouée et chaînée à leur page de base. La page 83 n'est pas pleine et peut encore accueillir un enregistrement

46. Adoptée notamment par Oracle pour l'accès aux *clusters*.

47. Nous ignorons dans cette section l'effet des suppressions d'enregistrements.

avant de déborder. La page 84 en revanche, a été attribuée à 9 enregistrements, dont les 5 derniers ont été stockés dans les deux pages de débordements 908 et 910. La page 85 est pleine mais ne débord pas (encore). Les pages 82 à 85 ont *accueilli* 22 enregistrements. Parmi ceux-ci, 15 ont trouvé place dans leur page de base, 6 sont dans la première page de débordement et 1 dans la deuxième. Leur accès nécessitera respectivement 1, 2 et 3 lectures de pages. Soit en moyenne $(1 \times 15 + 2 \times 6 + 3 \times 1) / 22 = 1,36$ lectures. Le temps d'accès moyen à un enregistrement est donc, ici, de **16,8 ms** (minimum 12,3 et maximum 36,9).

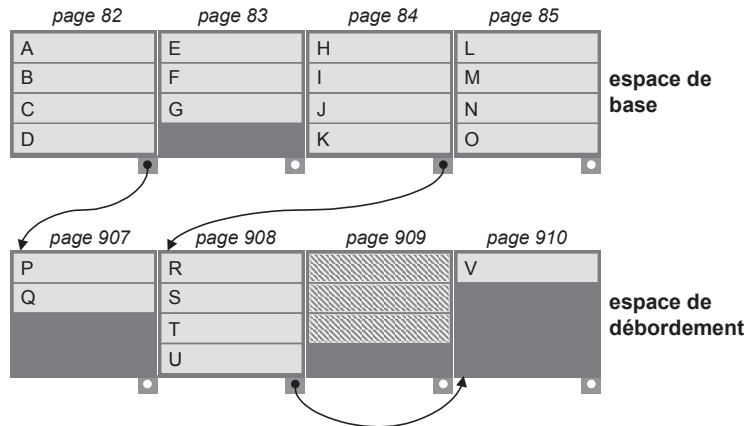


Figure 4.25 - Gestion des débordements en zone indépendante

4.9.4 Caractéristiques et performances d'un fichier à accès calculé

Cette section reprend la synthèse de l'analyse de l'annexe A4. Les paramètres de définition d'un fichier calculé sont le nombre de pages de base **Npb** et la taille de ces pages **Mrpp**.

Son fonctionnement est décrit par une troisième grandeur : le nombre moyen d'enregistrements **Nrpp** que la fonction **f** a envoyés dans une page. Ce nombre peut dépasser **Mrpp**. On l'exprimera sous la forme du *taux de chargement* **Tch**, qui mesure la proportion de ce nombre par rapport à la taille des pages. Ce taux est compris typiquement entre 0,5 et 2. On a donc **Nrpp** = **Tch** x **Mrpp**.

Exemple

Pour un fichier de **Npb** = 500 000 pages de **Mrpp** = 20 enregistrements, un taux **Tch** = 0,9 indique qu'on a essayé de stocker en moyenne **Nrpp** = 0,9 x 20 = 18 enregistrements par page, soit 18 x 500 000 = 9 000 000 enregistrements au total. Bien que, dans cet exemple, **Nrpp** < **Mrpp**, un nombre significatif de pages vont déborder (dans 134 000 pages de débordement, soit un accroissement de 27%, comme nous allons le voir), mais ces pages de débordement contiendront peu d'enregistrements (450 000 enregistrements, soit 5% du total). 9 000 000 - 450 000 = 8 550 000 enregistrements se trouvent dans leur page de base et leur accès nécessitera 1 lecture de page. Les autres en exigeront 2. En moyenne, l'accès par clé à un enregistrement réclamera $0,95 \times 1 + 0,05 \times 2 = 1,05$ lectures de page ou

12,9 ms. Cette excellente performance est obtenue au prix d'un taux d'occupation effectif du fichier de 71%, ce qui est assez médiocre.

Un fichier à organisation calculée en cours de fonctionnement est donc décrit par trois grandeurs : **Npb**, **Mrpp** et **τ_{ch}**. Cependant, les règles qui nous permettront d'étudier le volume et les temps d'accès d'un tel fichier en toute généralité ne dépendront que de deux paramètres, **Mrpp** et **τ_{ch}**.

a) Volume d'un fichier à accès calculé

Le nombre total d'enregistrements est

$$Nr = \tau_{ch} \times Mrpp \times Npb$$

La taille d'un fichier, y compris l'espace de débordement, se calcule comme suit :

$$Np = \tau_{ap} \times Npb$$

où **τ_{ap}** exprime le taux d'accroissement du nombre initial de pages (**Npb**) dû au phénomène de débordement (il est de 1,27 dans l'exemple ci-dessus).

Une troisième grandeur nous sera très utile, le taux d'occupation global **τ** (72% ou 0,72 dans notre exemple ci-dessus). Ce taux mesure l'efficacité d'utilisation de l'espace du disque par le fichier. Il se calcule comme le rapport entre le volume strictement nécessaire (**Nr / Mrpp**) et le volume réellement occupé (**Np**) :

$$\tau = Nr / (Mrpp \times Np)$$

La seule grandeur inconnue est le taux d'accroissement du fichier **τ_{ap}**. Les courbes de la figure 4.26 donnent ce taux pour les valeurs les plus courantes de taille des pages **Mrpp** et du taux de chargement **τ_{ch}**.

Exemple

Soit un fichier de base de 500 000 pages de 40 enregistrements et un taux de chargement de 1,5. La courbe relative à **Mrpp** = 40 indique, pour **τ_{ch}** = 1,5, la valeur **τ_{ap}** = 2. Ce fichier comporte **Np** = 1 000 000 pages dans lesquelles sont stockés **Nr** = 30 000 000 enregistrements. Le taux d'occupation global est **τ** = 0,75.

b) Temps de lecture séquentielle d'un fichier à accès calculé

L'ordre des pages et des enregistrements dans celles-ci étant non significatif, on effectue une lecture séquentielle des pages de base suivie de celle des pages de débordement. Ces pages étant généralement consécutives sur le disque, on adoptera le temps de lecture séquentielle avec lecture anticipée d'une piste, soit **t_{ts1}** :

$$t_{tsf} = Np \times t_{ts1}$$

c) Temps de lecture par clé d'un fichier à accès calculé

L'analyse de l'annexe A4 conduit au graphique de la figure 4.27, qui indique, pour des tailles de pages **Mrpp** de 5 à 50, le nombre moyen de pages à lire (**n_{lp}**) pour retrouver un enregistrement en fonction du taux de chargement **τ_{ch}** du fichier.

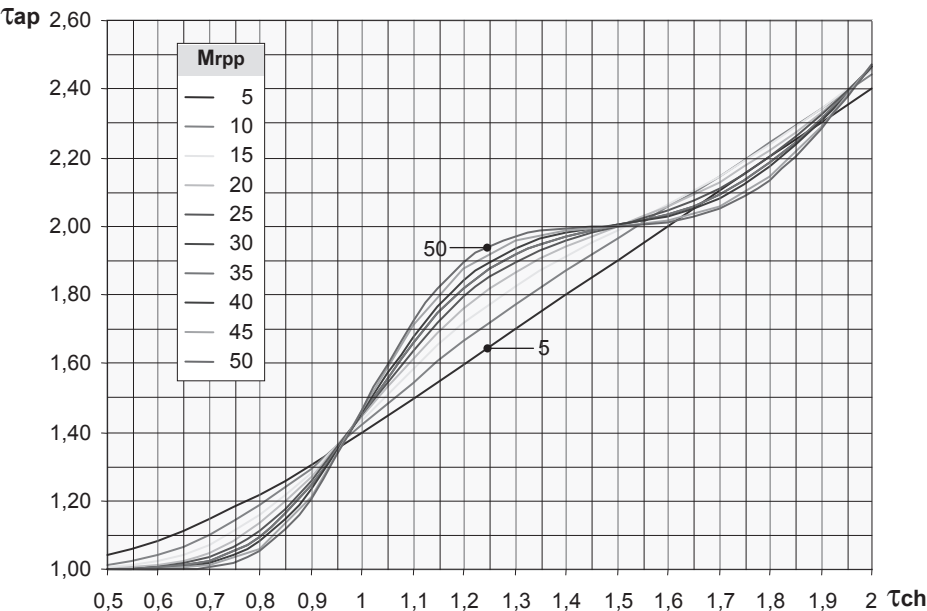


Figure 4.26 - Rapport du nombre total de pages au nombre de pages de base (τ_{ap}), en fonction du taux de chargement τ_{ch} pour les tailles de pages $Mrpp$ de 5 à 50

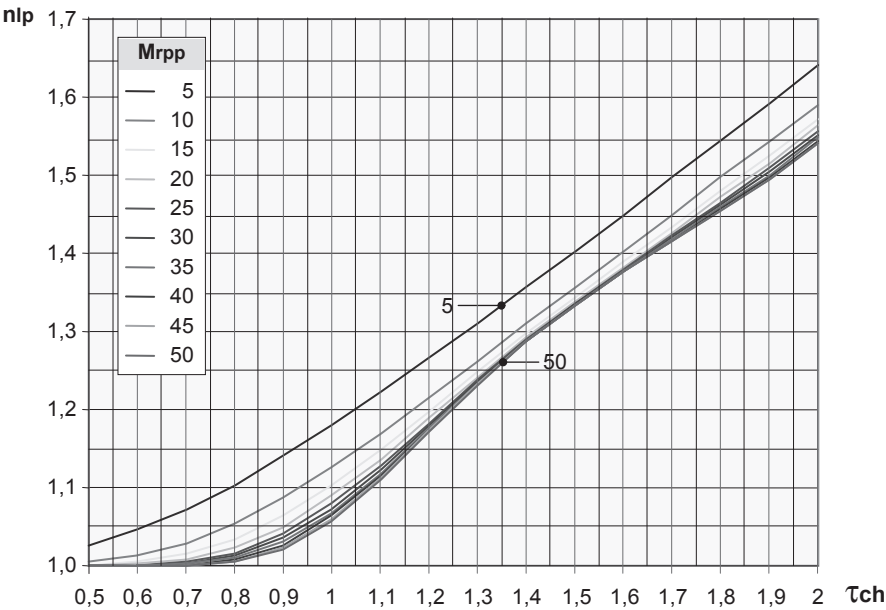


Figure 4.27 - Nombre moyen d'accès physiques (nlp) en fonction du taux de chargement de base τ_{ch} , pour les tailles de pages $Mrpp$ de 5 à 50

Exemple

Reprenons notre fichier de 500 000 pages de 40 enregistrements au taux de chargement de 1,5. La courbe relative à $Mrpp = 40$ indique, pour $\tau_{ch} = 1,5$, $n_{lp} = 1,33$. L'accès à une page coûtant 12,3 ms, le temps moyen d'accès par clé à un enregistrement est de 16,4 ms. Le temps minimum est de 12,3 ms et le maximum⁴⁸ est de 4 lectures de pages, soit 49,2 ms.

4.9.5 Chargement et réorganisation d'un fichier à organisation calculée

Le remplissage initial d'un fichier à organisation calculée s'effectue en principe par l'insertion des enregistrements d'un fichier source lu séquentiellement. Ce processus est particulièrement coûteux, chaque enregistrement réclamant la lecture d'une page d'adresse aléatoire et sa réécriture, soit au minimum 24,6 ms, soit encore une vitesse misérable de 40 enregistrements/s. Le chargement d'un fichier de 1 000 000 enregistrements de 400 octets durerait dans ces conditions près de 7 heures, pour un volume de moins de 400 Mo. Il est heureusement possible de réduire ce temps de chargement de manière très simple, en procédant comme suit.

1. On complète le fichier source **F0** d'un nouveau champ **A**, auquel on assigne, pour chaque enregistrement, sa future adresse de base, calculée par la fonction d'adresse **f**. Le coût de cette opération est celui de la lecture du fichier **F0** et de l'écriture du fichier **F1** complété du champ **A**. L'exercice de la section 4.6.2/d nous apprend que le coût est proche de $2 \times 18 = 36$ s, si on néglige la taille de **A**.
2. On trie le fichier **F1** sur **A**, produisant ainsi le fichier **F2**. Le coût est celui du tri d'un fichier, qu'on a appris à calculer à la section 4.6.2, soit ici 72 s. sans parallélisme.
3. On procède à l'insertion des enregistrements de **F2** comme suggéré plus haut. Cette fois cependant, le coût est considérablement réduit : les enregistrements se présentent dans l'ordre des numéros de page dans le fichier. Si on néglige les débordements, qui seront très rares à ce stade, le coût du chargement se réduit à celui de la lecture de **F2** + celui de l'écriture séquentielle du fichier final, dont la taille, pour un taux de chargement initial de 70% est évaluée à 572 Mo. Le coût de cette phase serait, dans notre exemple, de $18 + 26 = 44$ s.

Au total, le temps de chargement serait de 152 s, soit 0,6% des 7 heures de la première procédure.

Lorsque le coût d'accès par clé au fichier calculé sera jugé excessif, on pensera à le réorganiser tout comme nous l'avons fait pour le fichier séquentiel indexé (section 4.8.2). La **réorganisation** est simplement un chargement dans un espace de base plus grand.

48. Ce maximum est observé dans la feuille de calcul détaillée qui a permis de construire les graphiques de cette section. Elle est décrite à l'annexe A4 et est disponible en ligne.

4.9.6 Comparaison des deux techniques d'index primaires

Le tableau ci-après compare les caractéristiques des deux organisations de fichier. La *disponibilité* indique si la technique est fréquemment utilisée, la *stabilité des adresses* indique la propension à conserver à un enregistrement son adresse d'insertion, la *constance du temps d'accès* est évaluée pour les enregistrements à un instant déterminé, la *dégradation* indique la manière dont le volume occupé et les temps d'accès augmentent au fur et à mesure des insertions et modifications, la *reconstruction* mesure le temps nécessaire à la (re)construction de l'index.

	séquentiel indexé	calculé
Disponibilité	très répandu	plus rare
Occupation de l'espace disque	moyen	moyen à excellent
Occupation mémoire centrale	premiers niveaux d'index : quelques dizaines à quelques milliers de pages	une seule page
Stabilité des adresses	faible	bonne
Temps moyen d'accès par clé (nbre de pages lues)	moyen : $n+1$, mais améliorable via le tampon; typiquement 2	excellent : 1 à 1,2
Constance du temps d'accès par clé parmi les enregistrements	temps constant	temps variable ; typiquement de 1 à 4
Temps d'accès par clé en fonction de Npb	variable, proportionnel à $\log Npb$	constant
Accès ordonné	oui, selon valeurs de la clé	non
Accès par intervalle	oui, selon valeurs de la clé	non
Dégradation espace	progressive	nulle
Dégradation accès par clé	progressive, faible	progressive
Dégradation accès séqu.	progressive, importante	nulle
Reconstruction	rapide	lente

4.10 LES INDEX SECONDAIRES

Alors que le fichier de base est organisé, soit en vrac (les enregistrements y sont rangés de manière aléatoire), soit selon un index primaire (et un seul), on peut lui associer un nombre quelconque d'index secondaires. Contrairement aux index primaires, un index secondaire ne perturbe pas l'organisation du fichier de base⁴⁹. Dans une base de données relationnelle, il peut donc être ajouté (*create index*) ou

49. Du moins pour les techniques basées sur des listes inversées (ou listes de pointeurs) décrites dans cet ouvrage.

retiré (*drop index*) à tout moment sans précautions particulières, souplesse qui se paie cependant par des performances inférieures. C'est la technique généralement adoptée par la plupart des SGBD relationnels car elle permet une modification aisée et peu coûteuse des structures physiques en fonction de l'évolution des besoins. Nous étudierons deux techniques de référencement des enregistrements dans le fichier de base : par *liste de pointeurs* et par *bitmap*.

4.10.1 Structure d'un index secondaire par listes de pointeurs

Un index secondaire sur un champ (ou suite de champs) K est constitué de deux composants : un *dictionnaire de valeurs* et un *index* permettant un accès rapide à ce dictionnaire. Le dictionnaire contient en un seul exemplaire toutes les valeurs de K présentes dans les enregistrements du fichier. Chaque *entrée* de ce dictionnaire reprend une valeur k de K accompagnée de la liste des adresses des enregistrements tels que $K = k$. Si K est un identifiant des enregistrements, alors le dictionnaire contient autant d'entrées que le fichier compte d'enregistrements⁵⁰. L'accès à une valeur du dictionnaire se fait par un index, qui est obligatoirement un index primaire⁵¹ (pourquoi ?). En général, on désire que les valeurs du dictionnaire soient ordonnées, de sorte que ce dictionnaire est le plus souvent de type séquentiel indexé.

La figure 4.28 montre un fichier en vrac contenant des enregistrements (ou lignes de table) CLIENT.

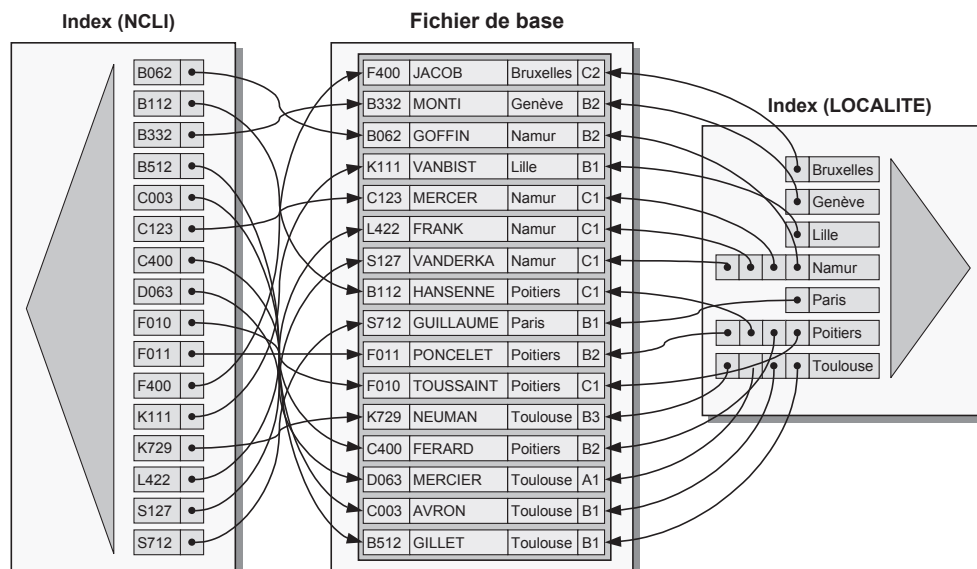


Figure 4.28 - Deux index secondaires associés à un fichier en vrac

50. On qualifie d'ailleurs ce type d'index, qui comporte une entrée par enregistrement, de **dense**, par opposition aux index des fichiers séquentiels indexés, qui comptent une entrée par page.

51. Donc, un index secondaire contient lui-même un (petit) index primaire ! Il faut bien voir dans un index secondaire un (petit) fichier séquentiel indexé ou calculé au plein sens du terme.

À gauche, un index secondaire est construit sur le champ NCLI, qui est un identifiant. Le dictionnaire de valeurs est constitué d'entrées contenant une valeur de NCLI et un pointeur vers l'enregistrement qui possède cette valeur. Le triangle à l'extrême gauche représente symboliquement l'index primaire de ce dictionnaire. À droite, un index secondaire est défini sur le champ LOCALITE, dont le dictionnaire reprend chaque valeur distincte de ce champ, accompagnée de la liste des pointeurs vers les enregistrements qui possèdent cette valeur. Par simplicité, la structure des pages n'est pas visualisée.

Il existe plusieurs techniques de gestion de la liste de pointeurs, dont la longueur, d'une valeur à l'autre, et d'un instant à l'autre, peut être extrêmement variable (une localité peut compter plusieurs milliers de clients tandis qu'une autre peut n'en compter qu'un seul). Mentionnons deux techniques. La première, utilisée par DB2 d'IBM, consiste à segmenter cette liste en sous-listes de R pointeurs au maximum (par exemple R = 64). Chaque sous-liste fait l'objet d'une entrée reprenant la valeur de K et un champ sous forme de tableau de taille R contenant cette sous-liste. L'entrée théorique que nous venons de décrire est donc remplacée par une suite d'entrées réelles de même valeur de K, fournissant chacune **une fraction** de la liste de pointeurs (figure 4.29).

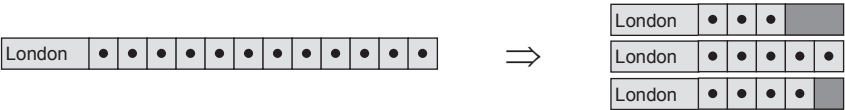


Figure 4.29 - Représentation d'une entrée théorique de taille variable sous forme d'une suite de taille variable d'entrées réelles de taille fixe

La seconde technique, adoptée par la plupart des autres SGBD, pousse la technique précédente à ses limites en adoptant R = 1. Il y a donc une entrée par enregistrement, même pour un index non identifiant (figure 4.30). Cette technique est plus simple à gérer mais conduit à des index plus volumineux.

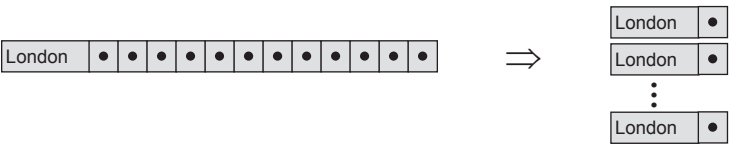


Figure 4.30 - Représentation d'une entrée théorique de taille variable sous forme d'une suite de taille variable d'entrées réelles comportant un seul pointeur

4.10.2 Performances d'un index secondaire par listes de pointeurs

Ainsi que nous l'avons observé, un index secondaire est un *dictionnaire de valeurs* organisé comme un fichier séquentiel indexé, ou, plus rarement, comme un fichier calculé. Nous pouvons donc lui appliquer le modèle de calcul des volumes et des temps d'accès développés aux sections 4.8 et 4.9. Les calculs seront différents selon que l'index secondaire est identifiant ou non. Notons **Nv** le nombre de valeurs présentes dans le dictionnaire et **Lv** leur longueur, supposée constante.

a) Index secondaire identifiant

Lorsque l'index est défini sur un identifiant K , le dictionnaire contient N_v entrées, constituées d'une valeur de K et d'un pointeur d'enregistrement de longueur L_{ptr} , typiquement 6 octets, pour un total de L_v octets. L'hypothèse d'unicité impose ici que $N_v = N_r$. On calcule, pour le dictionnaire, M_{vpp} la capacité des pages et N_{vpp} le nombre moyen d'entrées par page (τ_v est le taux d'occupation du dictionnaire) :

$$L_v = L_k + L_{ptr}$$

$$M_{vpp} = \lfloor L_p / L_v \rfloor$$

$$N_{vpp} = \tau_v \times M_{vpp}$$

On en déduit la taille N_{pv} du dictionnaire :

$$N_{pv} = \lceil N_v / N_{vpp} \rceil$$

Si le dictionnaire est d'organisation séquentielle indexée, son index primaire comporte nv niveaux. L'accès par clé implique le parcours de nv pages de l'index du dictionnaire + 1 page du dictionnaire + 1 (ou plusieurs) page(s) du fichier de base, soit $nv + 2$ pages. Lorsque la taille du dictionnaire est importante, on ne peut envisager de charger dans le tampon que les pages des $nv - 1$ premiers niveaux ou au mieux celles des nv niveaux de l'index du dictionnaire. Le nombre d'accès physiques lors de l'accès selon $K = k$ via un index secondaire est de 3 ou au mieux 2, soit 1 de plus qu'en séquentiel indexé pour un index identifiant. L'exercice 4.2 compare les performances en volume et temps d'accès des différentes techniques d'accès par clé identifiante.

b) Index secondaire non identifiant

Si l'index n'est pas identifiant, deux éléments nouveaux sont à prendre en compte. D'une part, en raison de la présence d'une liste de pointeurs, les entrées du dictionnaire sont de longueur variable et en général sont plus volumineuses. D'autre part, chaque entrée du dictionnaire conduit à un ensemble d'enregistrements dans le fichier de base. Sauf cas extrême, $N_v < N_r$. À chaque valeur de K sont associés en moyenne N_{rpv} enregistrements :

$$N_{rpv} = \lceil N_r / N_v \rceil$$

À combien d'entrées réelles correspond une entrée théorique ? Nous devons d'abord évaluer le contenu moyen d'une sous-liste. Une analyse rapide nous permet de conclure qu'elles devraient se comporter comme les pages des fichiers séquentiels indexés (pourquoi ?). On peut donc tabler sur un taux d'occupation de 80 % (au démarrage) à 70 % (après de nombreuses modifications), soit, pour les besoins de la présentation, 75 %. Pour une taille de tableau R , chaque entrée peut accueillir $R' = 0,75 \times R$ pointeurs. On en conclut qu'à une entrée théorique correspondent en moyenne ner entrées réelles :

$$ner = \lceil N_{rpv} / R' \rceil$$

Le dictionnaire contiendra donc N_{er} entrées réelles :

$$N_{er} = N_v \times ner$$

Ceci permet de calculer la taille réelle du dictionnaire et celle de son index⁵².

Seconde nouveauté, chaque entrée théorique (et les entrées réelles correspondantes) pointe vers plusieurs (**Nrpv** ou **R'**) enregistrements. Sauf mention contraire, ces derniers sont distribués aléatoirement dans l'espace du fichier. Le coût **tixk** de lecture des enregistrements référencés par une entrée théorique de dictionnaire est donc :

$$\mathbf{tixk} = \mathbf{Nrpv} \times \mathbf{tia1}$$

Cependant, si l'index est du type *clustering* (section 4.13), alors les enregistrements sont en outre en ordre sur **K** dans le fichier de base, ce qui réduira le temps d'accès à ces enregistrements. Le calcul du temps de lecture des **Nrpv** enregistrements sera examiné en 4.13.

Remarque

On vient d'admettre que la lecture d'enregistrements distribués aléatoirement dans l'espace du fichier coûtait pour chacun d'eux **tia1**. Cette estimation est pessimiste et peut être réduite moyennant l'optimisation suivante. Les pointeurs de l'entrée ou des entrées sont préalablement chargés en mémoire puis triés par valeurs croissantes. L'accès se fait alors à partir des adresses considérées dans cet ordre. Si le disque est exclusivement dédié à ce processus de lecture, le temps de lecture de ces pages ne doit pas dépasser le temps de lecture séquentielle en *streaming* du fichier.

c) Lecture par index des enregistrements d'un intervalle

L'accès par intervalle de valeurs de **K** comporte deux opérations : la lecture des entrées du dictionnaire qui satisfont la condition $K \in [k1, k2]$ et la lecture des enregistrements référencés par ces entrées. Le coût de la première opération a été étudié en détail à la section 4.8.3. Le coût de la seconde est celui de **Nrpv** lectures aléatoires (pour autant que **Nrpv** << **Nr**), soit **Nrpv** x **tia1**. Ici encore, un index du type *clustering* apportera une forte amélioration des performances.

d) Nature des pointeurs

Les pointeurs d'enregistrement pourraient être remplacés par des pointeurs de page. En effet, une recherche dans la page sur le critère $K = k$ permettrait de repérer les enregistrements qu'on recherche. On peut escompter deux avantages : pointeurs plus courts (4 octets au lieu de 6) et moins nombreux (un pointeur peut désigner plusieurs enregistrements). Cependant, ces avantages sont minimes. On gagnerait par exemple 20 Mo pour une table de 10 000 000 d'enregistrements. En outre, la présence de plusieurs enregistrements satisfaisant $K = k$ par page supposerait que **Nrpv** > **Np**, ce qui serait tout à fait inhabituel !

52. Petite particularité : plusieurs pages du fichier de base peuvent contenir des entrées de même valeur de **K**. Le calcul de l'index est dans ce cas identique à ce que nous avons étudié plus haut.

4.10.3 Structure d'un index *bitmap*

Dans un index *bitmap*, la liste de pointeurs est remplacée par une chaîne de bits, le bit de rang *i* désignant le *i*^e enregistrement dans le fichier. Pour une entrée de valeur de clé *k*, tout bit de la chaîne correspondant à un enregistrement dont *K* = *k* est à 1, sinon il est à 0. Ces chaînes étant très longues (**Nr** bits exactement), une technique de compression sans perte, du type LZW par exemple, est appliquée au moment du stockage. Une discussion détaillée et le modèle de calcul des index *bitmap* sont présentés à la section A4.6.1 de l'annexe A4.

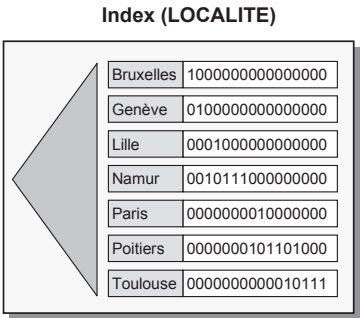


Figure 4.31 - Index *bitmap* sur la colonne LOCALITE (table en vrac de la figure 4.28)

4.10.4 Les index partiels

Un index partiel n'indexe qu'une partie des lignes de la table à laquelle il est associé. Les autres lignes ne sont pas accessibles via cet index. L'index créé ci-dessous ne référence les lignes de COMMANDE qu'à partir du 1^{er} janvier 2014.

```
create index NDX_COM_DATE on COMMANDE (DATECOM)
where DATECOM >= '2014-01-01';
```

Cette forme d'index permet en particulier de contourner les lacunes des SGBD (SQL Server par exemple) qui ne gèrent pas correctement les identifiants facultatifs (section 2.6), grâce au filtre *where C is not null*.

4.10.5 Les index sur expression

Un index peut indexer une table sur une fonction ou plus généralement sur une expression :

```
create index NDX_CLI_NOM on CLIENT (lower(NOM)) ;
```

4.10.6 Les index secondaires augmentés

Certains SGBD (DB2, SQL Server) proposent le concept d'*index augmenté*⁵³, qui consiste en un *index secondaire identifiant* dans lequel les entrées du dictionnaire de valeurs possèdent un ou plusieurs champs additionnels copiés des enregistrements

du fichier de base. Par exemple, l’index sur NCLI de la figure 4.28, dont la structure d’origine des entrées, (NCLI, ptr), serait augmenté des champs LOCALITE et CAT, correspondant à la structure (NCLI, LOCALITE, CAT, ptr) de la figure 4.32.

La taille du dictionnaire est plus importante et entraîne une gestion plus coûteuse, mais un tel index permet de répondre à certaines requêtes sans imposer l’accès au fichier de base. Les nouveaux champs peuvent servir à vérifier des conditions de sélection additionnelles mais aussi à fournir les valeurs de la réponse aux requêtes. L’index augmenté qui vient d’être décrit permet de répondre aux requête suivantes sans accès à la table CLIENT :

```
select NCLI, LOCALITE, CAT from CLIENT
where NCLI between k1 and k2 and CAT like 'B%';

select distinct LOCALITE from CLIENT
where CAT like 'B%';54
```

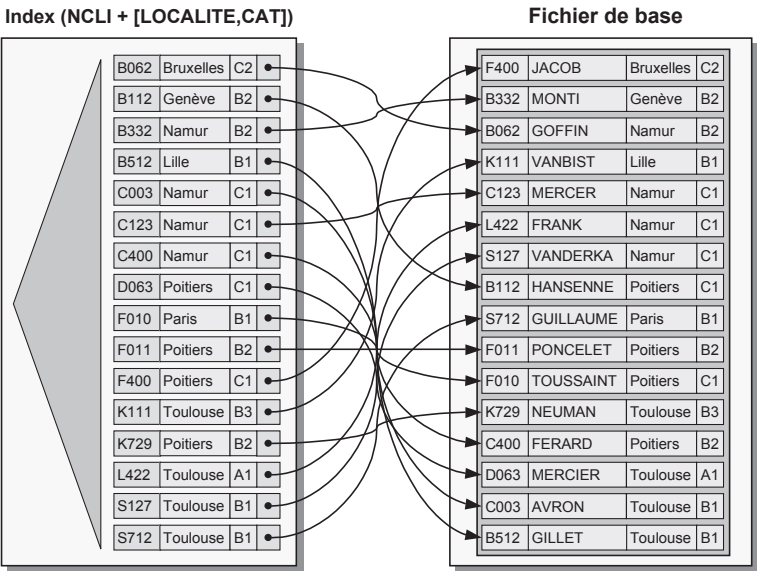


Figure 4.32 - Index identifiant sur NCLI augmenté des champs LOCALITE et CAT

53. On parle aussi d’**index couvrant**, ou *covering index*. Ce concept est cependant un peu différent : un index (augmenté ou non) est couvrant pour une requête Q si toutes les colonnes nécessaires à l’évaluation de Q sont reprises dans l’index.

54. Cette requête nécessitera évidemment une lecture séquentielle du dictionnaire de valeurs, mais celui est généralement beaucoup plus petit que la table de base.

4.10.7 Disponibilité des techniques d'indexation primaires et secondaires

Les techniques offertes par les SGBD sont en évolution constante. En 2010, ceux-ci offraient notamment les types d'index suivants (SI = séquentiel indexé).

- **Oracle.** Table organisée en vrac ou en SI. Index secondaire en SI ou bitmap. *Clusters* en SI ou en calculé (voir 4.13).
- **DB2.** Table organisée en vrac. Index secondaire en SI, éventuellement de type *clustering* (voir 4.13).
- **SQL Server.** Table organisée en vrac ou en SI (dénommée *clustered index* !). Index secondaire en SI.
- **PostgreSQL.** Table organisée en vrac. Index secondaire en SI ou en calculé. L'usage du calculé est découragé.
- **MySQL.** Table organisée en vrac. Index secondaire en SI ou en calculé.

4.11 ACCÈS PAR INDEX OU RECHERCHE SÉQUENTIELLE ?

Un index sera utilisé pour exécuter une requête pour autant que le SGBD juge son usage plus efficace qu'une simple recherche séquentielle dans le fichier. Considérons la requête suivante :

```
select NCLI,NOM,ADRESSE from CLIENT where CAT = 'B1';
```

qui s'adresse à une table de 1 000 000 enregistrements de 400 octets occupant 140 000 pages de 4 Ko. Supposons que CAT utilise 10 valeurs distinctes et fasse l'objet d'un index. En négligeant le temps de consultation de l'index, l'accès par index à 10% des enregistrements, répartis aléatoirement dans le fichier, coûtera en moyenne

$$t_{ixk} = N_{rpv} \times t_{ia1} = 100\,000 \times 0,0123 = 1\,230 \text{ s}$$

Une simple lecture séquentielle coûtera, quant à elle,

$$t_{isf2} = N_p \times t_{is1} = 140\,000 \times 0,00184 = 26 \text{ s}$$

Il est évident que le SGBD ignorera l'index au profit d'une recherche séquentielle **47 fois plus rapide** ! On trouvera à la section 20.9 et à la section A4.6 de l'annexe de ce chapitre une étude plus détaillée de cette question.

4.12 QUAND CRÉER UN INDEX ?

Il est habituel, lorsqu'une table est créée, d'y **charger** une grande quantité de données issues de sources externes telles qu'un fichier au format CSV (voir annexe A4) ou une version antérieure de la table lors d'une réorganisation. Ces processus ont été abordés dans les sections 4.8.2 et 4.9.5 pour les organisations séquentielle

indexée et calculée pour ce qui concerne leur index primaire. On y a vu l'importance du tri initial des données sources dans les performances du chargement.

Souvent, la table nouvellement créée ou en réorganisation est dotée d'**index secondaires**. Une question fréquemment posée est la suivante : *est-il préférable de créer ces index **avant** ou **après** le chargement des données ?*

Soit un index secondaire défini sur la clé **K**. Considérons deux cas.

- Les données sources sont *ordonnées selon K*, ce qui n'est possible que si la table est organisée *en vrac*, sans index primaire. Le chargement des données sources est un processus séquentiel extrêmement rapide. Le chargement de l'index s'effectue en parallèle, lui aussi par un processus séquentiel. Sa création a posteriori demande une relecture supplémentaire de la base de données.

Conclusion : il est plus efficace de créer l'index **avant** le chargement de la table.

- L'ordre des données sources est *aléatoire selon K*, ce qui sera le cas lors du chargement d'une table dotée d'un index primaire, qui demande que les données sources soient triées selon l'identifiant primaire (ou sa fonction de hachage).
 - Si l'index secondaire est *construit avant le chargement*, chaque insertion dans la table entraîne une insertion aléatoire dans l'index, qui exigera en général un accès physique à la mémoire externe dès lors que la totalité de l'index ne peut résider dans le tampon.
 - La construction de l'index secondaire *après le chargement* demande (1) une relecture séquentielle de la table, (2) la constitution d'une liste [clé, adresse], (3) le tri de cette liste puis (4) son stockage sous la forme de l'index. Ces processus sont constitués de lecture/écritures séquentielles et sont donc beaucoup plus efficaces que les accès aléatoires. On rappelle les chiffres de la figure 4.6 : lecture d'une page aléatoire = 12,3 ms et lecture séquentielle d'une page = 0,183 ms, soit un rapport de 67.

Conclusion : sauf dans certains cas limites, il est plus efficace de créer l'index **après** le chargement. Si cet index existait auparavant, on envisagera de le supprimer avant le chargement puis de le créer après. Si la table contenait déjà des données, la décision de conserver ou de reconstruire l'index dépendra du rapport du volume nouveau au volume préexistant.

4.13 LES TECHNIQUES D'AGRÉGATION (CLUSTERING)

L'idée de placer les enregistrements dans l'espace du fichier de manière à favoriser certaines requêtes est à la base de l'organisation séquentielle indexée. Plus généralement, on cherche à placer à proximité les uns des autres les enregistrements qui sont souvent demandés simultanément ou successivement. Ainsi, l'accès à l'un permet d'obtenir les autres sans accès physiques supplémentaires. Ce mécanisme est dénommé *agrégation* ou *clustering*. Il apparaît sous deux formes : les *clustering index* et les *clusters*.

4.13.1 Clustering index

Il s'agit d'un index secondaire sur une clé K associé à un fichier dont les enregistrements sont (plus ou moins) maintenus en ordre selon les valeurs de K. Lors d'une insertion, un enregistrement est placé, autant que possible, dans la page de son prédécesseur ou de son successeur.

Plus précisément, le chargement initial du fichier de base laisse dans chaque page une quantité déterminée d'espace libre (60 à 80% par exemple). Lors de l'insertion d'un enregistrement, celui-ci est rangé dans la page de son prédécesseur (ou successeur) selon l'ordre des valeurs de K. S'il n'existe plus d'espace libre dans cette page de base, il y a rupture de séquence, et l'enregistrement est rangé dans la page la plus proche de sa page de base. L'accès séquentiel ordonné selon K est réalisé via l'index. Ses performances sont au début aussi bonnes que celles du séquentiel indexé, mais cet avantage se perd progressivement à cause des ruptures de séquence. Après un certain temps, les performances se rapprochent de celles d'un index secondaire d'un fichier en vrac. En DB2, le statut de *clustering* est automatiquement désactivé et ignoré par l'optimiseur SQL au-delà d'un taux déterminé de ruptures. Il est alors conseillé de recharger la table et de reconstruire l'index.

La technique des *clustering index* combine les avantages des index secondaires et la rapidité de l'accès séquentiel trié ou par intervalle du séquentiel indexé (figure 4.18/b et /c). On la retrouve dans DB2, Oracle et SQL Server.

4.13.2 Clusters

Un agrégat, ou *cluster*, est une collection d'enregistrements, éventuellement de types différents, qui possèdent un champ de même valeur et qui sont regroupés physiquement dans la même page ou dans des pages dédiées. L'application la plus répandue est le regroupement des lignes de tables dont on demande fréquemment la jointure. Par exemple, l'accès à une ligne de COMMANDE est généralement accompagnée de l'accès aux lignes de DETAIL qui lui sont associées selon la condition DETAIL.NCOM = COMMANDE.NCOM. On pourra définir un *cluster* regroupant les tables COMMANDE et DETAIL sur les colonnes NCLI. Lors de l'accès à une ligne de COMMANDE, les lignes de DETAIL qui lui sont associées et qui ont été rangées dans la même page se trouveront dans le tampon lorsqu'on demandera à y accéder. La lecture d'une ligne de COMMANDE et de ses **d** lignes de DETAIL réclamera 1 accès au disque et non 1 + **d** comme dans d'autres schémas de rangement. Un *cluster* est considéré comme un **super-enregistrement** qui est localisé via un index classique, du type séquentiel indexé, calculé ou secondaire. Dans certaines organisations, chaque *cluster* se voit assigner une page (figure 4.33) ou plus généralement une chaîne de pages dédiées. Cette technique est offerte notamment par Oracle⁵⁵ et par F1 de Google.

55. La syntaxe est la suivante :

```
create cluster COMDET (NCOM char(12));
create table COMMANDE (...) cluster COMDET (NCOM);
create table DETAIL (...) cluster COMDET (NCOM);
```

La définition d'un cluster implique que des enregistrements de types différents puissent être stockés dans le même fichier. Il est à noter qu'il existe, pour les enregistrements composants des clusters, *une vie en dehors de ces clusters* ! On peut en particulier définir d'autres index (secondaires) sur chacun des types d'enregistrements.

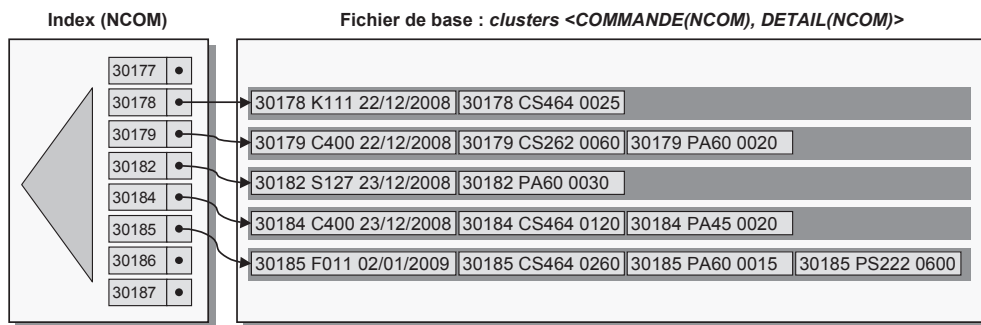


Figure 4.33 - Fichier de base contenant des clusters COMMANDE, DETAIL sur la colonne NCOM et index secondaire. Dans cet exemple, chaque cluster occupe une page.

4.13.3 Performances des techniques d'agrégation

a) *Clustering index*

Le calcul des performances d'un *clustering index* sur la clé K tient compte du fait que la structure comprend un index secondaire dont le dictionnaire est ordonné selon K et un fichier de base dont les enregistrements sont aussi ordonnés selon K. Le calcul des temps d'accès se distingue de celui de l'index secondaire sur fichier en vrac dans deux cas : l'accès par un *index non identifiant* et l'accès par *intervalle*.

Dans le cas de l'accès par un index non identifiant, le calcul du temps de lecture des **Nrpv** enregistrements consécutifs dans le fichier de base est identique à celui de la *lecture par index des enregistrements d'un intervalle* étudié dans l'organisation séquentielle indexée (4.8.3), où la grandeur **nrik** est remplacée par **Nrpv** :

$$npik = \lfloor Nrpv / Nrpp + 0,5 \rfloor + 1 = \lfloor Np / Nv + 0,5 \rfloor + 1$$

$$tlik = tla1 + (npik - 1) \times ttr \quad \text{sans ruptures, optimiste}$$

$$tlik = npik \times tla1 \quad \text{sans ruptures, réaliste}$$

$$tlik = tla1 + (1 - \pi_r) \times npik \times tla1 + \pi_r \times npik \times tla1 \quad \text{avec ruptures}$$

Pour l'accès par intervalle, nous nous inspirons du calcul de la même opération via un index secondaire sur fichier en vrac (4.10.2). L'accès comporte encore deux opérations : la lecture des entrées du dictionnaire qui satisfont la condition $K \in [k1, k2]$ et la lecture des enregistrements référencés par ces entrées. Le coût de la seconde est celui de **Nrpv** lectures séquentielles et non plus aléatoires, soit **Nrpv x tla1**.

b) Clusters

Le calcul des temps d'accès à des clusters en mode séquentiel, ou via l'index ou encore par intervalle est similaire à celui des temps d'accès à des enregistrements classiques. La principale différence concerne la taille de ces *super-enregistrements*, qui peut dépasser celle d'une page.

On montre sans difficulté que le cluster favorise plusieurs formes d'accès proposées par SQL, notamment :

```
select * from COMMANDE where NCOM = x
select * from DETAIL where NCOM = x
select * from COMMANDE M, DETAIL D where M.NCOM = D.NCOM
```

Il accélère également l'évaluation de certaines sous-requêtes ainsi que la gestion de l'intégrité référentielle dans les opérations `delete`.

4.14 STABILITÉ DES ADRESSES

La stabilité d'adresse est la capacité qu'ont des enregistrements de conserver leur adresse au travers des avanies qu'ils subissent durant leur vie active, éventuellement nomade. Ces enregistrements sont déplacés (éclatement, rééquilibrage, fusion de pages en séquentiel indexé, migration vers une autre page en cas de croissance), relocalisés (réorganisation d'un fichier calculé, modification de l'identifiant primaire), fragmenté, défragmentés, etc. Ces actions ont souvent pour effet de modifier la position des enregistrements dans l'espace de stockage et donc leur adresse. Cette modification est problématique pour les techniques d'accès basées sur des pointeurs, telles que les index secondaires, qui doivent être modifiés en conséquence. Il est donc particulièrement important d'utiliser des techniques d'implémentation qui garantissent cette stabilité. Nous analyserons brièvement l'impact des techniques de stockage sur la stabilité des adresses des enregistrements.

- **Stabilité du rangement en vrac.** L'adresse physique d'un enregistrement est indépendante de l'état de l'enregistrement. Elle est donc intrinsèquement stable et peut être utilisée pour référencer l'enregistrement.
- **Stabilité de l'organisation séquentielle indexée.** Les règles d'ajustement local lors des insertions et suppressions entraînent fréquemment des déplacements d'enregistrements, ce qui rend les adresses très instables. Il n'est donc pas possible de les utiliser dans les index secondaires sauf à retrouver et modifier tous les pointeurs référençant l'enregistrement déplacé, ce qui peut s'avérer très coûteux. Trois solutions:

1. Au lieu de l'adresse de l'enregistrement, on utilise la valeur de son identifiant primaire, laquelle est supposée constante durant la vie de l'enregistrement. Le prix de la stabilité n'est pas négligeable puisque l'accès à l'index secondaire est suivi d'un accès à l'enregistrement par index primaire. Cette technique est utilisée par de nombreux SGBD (voir *SQL Server* ci-après) en raison de sa simplicité. Elle sera d'autant plus efficace que la longueur de l'identifiant primaire est faible, ce qui sera le cas des identifiants techniques.

2. On utilise comme valeur des pointeurs la valeur de l'identifiant primaire **et** l'adresse de l'enregistrement. L'accès se fait d'abord à l'aide de l'adresse, puis, en cas d'échec (l'enregistrement ayant été déplacé), par la technique n° 1 ci-dessus.
- Dans ce dernier cas, on peut soit mettre à jour l'adresse, soit laisser les choses en l'état et reconstruire l'index secondaire dès que les échecs sont trop fréquents⁵⁶.
3. On utilise une table d'indirection (voir ci-dessous).
- **Stabilité d'un *clustering index*.** Les adresses sont intrinsèquement instables dès lors qu'on est amené à réorganiser le fichier.
 - **Stabilité de l'organisation calculée.** L'adresse d'un enregistrement dépend de la valeur de son identifiant (primaire le plus souvent). Cette adresse est stable tant que cette valeur n'est pas modifiée et que le fichier n'est pas réorganisé, auquel cas l'enregistrement change d'adresse. Pour éviter les conséquences d'une réorganisation, on peut comme ci-dessus utiliser la valeur de l'identifiant primaire. Ici, la pénalité est négligeable puisque, dans un fichier bien géré, le temps moyen d'accès est très proche de celui de l'accès à une page.
 - **Stabilisation par table d'indirection.** Cette technique consiste à attribuer à chaque enregistrement un numéro arbitraire *k* qu'il conservera durant toute sa vie. Elle utilise un tableau *T* comportant autant de cellules qu'il existe d'enregistrements dans l'espace de stockage ou dans le fichier (table). La cellule *T*[*k*] contient l'adresse réelle de l'enregistrement de numéro *k*. Toute référence à cet enregistrement se fera à l'aide de son numéro *k* (figure 4.34).

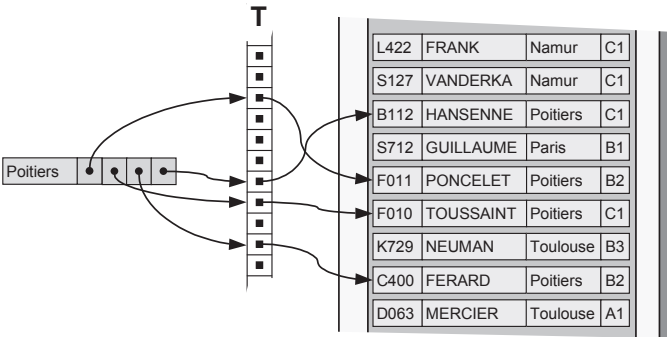


Figure 4.34 - La table d'indirection *T* stabilise les adresses instables

Lorsque l'enregistrement numéro *k* change d'adresse, seule la valeur de *T*[*k*] doit être modifiée. Cette technique nécessite un accès supplémentaire à la table d'indirection. Cependant, le coût additionnel peut être minime si cette table est

56. Technique utilisée par Oracle. La décision de ne pas mettre à jour l'adresse physique ne doit pas surprendre. En effet, une simple consultation deviendrait dans ce cas une opération complexe de modification, nécessitant des conditions d'exécution particulières décrites au chapitre 5.

préchargée en mémoire centrale. En considérant des adresses de 6 octets, le tableau T relatif à une table de 10 000 000 de lignes occupera moins de 60 Mo, ce qui est négligeable sur disque et reste raisonnable en mémoire centrale.

4.15 UN EXEMPLE : SQL SERVER

SQL Server de Microsoft (dérivé du SGBD Sybase) est un des SGBD les plus populaires. Il offre des structures de données relativement simples qu'il est facile de décrire à l'aide des concepts présentés dans ce chapitre⁵⁷.

• Les espaces de stockage

Une base de données comprend un ou plusieurs espaces de stockage dénommés *groupes de fichiers* (**FileGroup**). L'un d'eux est primaire et contient entre autres les tables du catalogue. Un groupe de fichiers est formé d'un ou plusieurs fichiers (**File**). Il est possible d'ajouter un fichier à un groupe, d'en retirer et d'étendre un fichier. Les fichiers d'un groupe peuvent être stockés sur des disques différents. Il existe deux types de fichiers : fichiers de données et fichiers du journal des transactions (voir chapitre suivant).

Un fichier est structuré en **pages** de 8 Ko. L'espace est alloué par extensions entières (**Extent**). Une extension est un bloc de 8 pages contiguës, soit 64 Ko. Un fichier contient jusqu'à 64 000 extensions, soit 512 000 pages ou à peu près 4 Go.

• Les tables et les lignes

Une table est assignée à un groupe de fichiers. Ses lignes peuvent être distribuées dans plusieurs fichiers du groupe. Une extension contient des lignes d'une seule table⁵⁸. Une table peut être partitionnée en plusieurs sous-tables (dites **partitions**) qui contiennent chacune un sous-ensemble de lignes.

Les lignes sont rangées dans les pages de manière dynamique (figure 4.9, technique *d*). Une page comprend un **header** de 96 octets, un tableau de pointeurs (**row offset table**) de 36 octets et 8 060 octets d'espace libre. Une ligne est entièrement contenue dans une page. Cependant, les valeurs de colonnes variables (varchar, text, XML, etc.) des lignes qui ne tiennent pas dans une page sont déplacées dans des pages de débordement. Lorsque, suite à sa modification, une ligne devient trop longue pour être recopiée dans sa page d'origine, elle provoque l'écèlement de sa page dans une *table clustered* et une indirection dans une *table en vrac*.

Une ligne est identifiée par son **row-id**, formé de l'identifiant de son fichier (2 octets), du n° de sa page (4 octets) et de son n° dans sa page.

57. Sources : Bruchez, R., *Optimiser SQL Server*, Dunod, 2008; site SQL Server de Microsoft (*Tables and Index Data Structures Architecture*); site <http://sqlserverpedia.com/>;

58. Une exception : les extensions *mixtes*, qui peuvent contenir plusieurs très petites tables.

• L'organisation des tables et les index

Il existe deux types de tables qui diffèrent selon leur structure de stockage : les *tables en vrac* (**heap tables**) et les *tables clustered* (**clustered table**), ces dernières étant organisées en séquentiel indexé (l'index primaire est appelé **clustered index**).

Les pages et les lignes d'une *table en vrac* n'ont pas d'ordre significatif. Chaque ligne est stockée là où il y a de la place. Son *row-id* est stable dans le temps.

Une *table clustered* contient des pages d'index et des pages de base⁵⁹. Les pages de chaque niveau d'index sont doublement chaînées pour permettre le parcours dans l'ordre logique des pages du niveau. Il en est de même des pages de base. Le phénomène de rupture de séquence (page suivante en fin de fichier) est dénommé **fragmentation de l'index**. Les *row-id* des lignes sont hautement instables.

Normalement, la clé d'un index *clustered* est identifiante⁶⁰ (tout identifiant primaire est automatiquement accompagnée d'un *unique clustered index*).

Les index secondaires prennent la forme de **nonclustered index**. Ils peuvent être identifiants ou non identifiants. Leur dictionnaire de valeurs est organisé en séquentiel indexé. Si l'index secondaire est non identifiant, la liste de pointeurs est décomposée en blocs de taille 1 ($R = 1$). En d'autres termes, il existe une entrée d'index au dernier niveau pour chaque ligne de la table (figure 4.35). Cette entrée est formée de la valeur de la clé et d'un seul pointeur. La nature de ce pointeur dépend de l'organisation de la table. Pour une *table en vrac*, le pointeur est un *row-id*. Pour une *table clustered*, ce pointeur est la valeur de la clé de l'index primaire identifiant la ligne, ce qui stabilise les adresses mais alourdit l'accès par index secondaire. Ainsi, dans une *table en vrac*, l'accès est direct tandis que dans une *table clustered*, le parcours de l'index secondaire doit être suivi d'un accès via l'index primaire. Un index secondaire peut être *augmenté*, via les **included columns**.

La clé d'un index primaire ou secondaire compte 16 colonnes et 900 octets au maximum. On peut spécifier le taux de chargement (**fillfactor**) d'un index (τ_i mais aussi τ_b pour une table clustered). Par défaut, il est, malheureusement, de 100%, ce qui est la pire des valeurs possibles dans le cas d'insertions aléatoires !

Poitiers	bx/1208/46	Poitiers	B112
Poitiers	bx/2746/08	Poitiers	C400
Poitiers	bx/2953/25	Poitiers	F010
Poitiers	bx/4452/19	Poitiers	F011

Figure 4.35 - Fragment du dictionnaire de valeurs d'un Index secondaire non identifiant pour une table en vrac (à gauche) et une table *clustered* (à droite)

59. Selon le vocabulaire propre à SQL Server, la *table est un index* dont le dernier niveau est constitué des pages de données. Nous n'adopterons pas cette terminologie dans cette présentation.

60. Si elle ne l'est pas (en l'absence d'identifiant primaire ou de prédicat unique donc), SQL Server ajoute automatiquement un suffixe de 4 octets (non visible par l'utilisateur) rendant les valeurs artificiellement uniques (*uniqueifier*).

• Divers

Dans un index, null est considéré comme une valeur. Il est donc impossible de définir un index identifiant dont la clé comporte une colonne facultative (null = null).

SQL Server n'offre pas d'organisation calculée, d'index bitmap ni de clusters. En revanche, il propose des index spécialisés pour les textes, les documents XML et les données spatiales. Il offre aussi des index partiels, qui peuvent être utilisés pour pallier les lacunes des identifiants facultatifs.

4.16 COMPLÉMENTS SUR LES FONCTIONS DE HACHAGE

4.16.1 Le prédicteur à hachage (filtre de Bloom)

Le *prédicteur à hachage*, ou *filtre de Bloom*,⁶¹ est une technique qui permet de vérifier qu'un élément *n'appartient pas* à un ensemble dont l'accès est coûteux. Examinons l'exemple d'un fichier séquentiel indexé de **Nr** enregistrements dont le temps moyen d'accès par clé est de 35 ms. Pour vérifier si une valeur de clé **k** n'est pas déjà présente dans le fichier (*ne correspond à aucun enregistrement*), il suffit d'accéder à celui-ci via l'index, ce qui conduit à une réussite ou à un échec. Le coût de cette vérification est donc de 35 ms.

Considérons une chaîne **P** stockée en mémoire centrale, appelée **prédicteur**, constituée de **n** bits, tous à 0 au moment de la création du fichier. Soit aussi **h** une fonction de hachage cadrant dans l'intervalle **[1-n]**. La gestion de **P** s'effectue ainsi : lors de l'insertion d'un enregistrement de clé **k**, le bit de rang **h(k)** est mis à 1 :

$$P[h(k)] := 1$$

Idéalement, on devrait pouvoir conclure que l'enregistrement de clé **k** est **présent** dans le fichier si $P[h(k)] = 1$ et qu'il en est **absent** si $P[h(k)] = 0$. Malheureusement, les fonctions de hachage n'étant généralement pas injectives, il est fréquent que plusieurs valeurs de clé activent le même bit, phénomène appelé *collision*. On conclut que la valeur de **k** est **absente** du fichier de façon certaine si $P[h(k)] = 0$ et qu'il est donc *inutile d'accéder au fichier*. En revanche, de la condition $P[h(k)] = 1$ on ne déduit pas grand chose, sinon que la valeur peut être présente dans le fichier ou en être absente (ce qu'on appellera un *faux positif*), incertitude qui *nécessite l'accès au fichier*. On peut augmenter la précision du prédicteur en augmentant la valeur de **n** (par exemple 5 ou 10 fois **Nr**) et en utilisant plusieurs fonctions **h**, activant chacune un bit. La valeur **k** est absente du fichier dès qu'un de ces bits est à 0. On réduit ainsi la probabilité de faux positifs.

Les prédicteurs ont de nombreuses applications. On les retrouve notamment dans les SGBD PostgreSQL et Cassandra, dans des navigateurs web (Google Chrome) et dans la plate-forme BitCoin. Nous examinerons à la section 5.6.4 l'application des prédicteurs à hachage à l'organisation des *fichiers différentiels*.

61. https://en.wikipedia.org/wiki/Bloom_filter (novembre 2017)

4.16.2 Hachage cryptographique (Secure hashing)

Considérons un ensemble **E** d'entités de données de taille et nature quelconques : messages, documents, enregistrements, fichiers, binaires exécutables, mots de passe, etc. Nous voudrions associer, via une fonction **f**, à chaque élément **m** de **E** une courte chaîne d'octets **s**, qui lui soit unique. Cette chaîne **s = f(m)** est appelée la *signature* ou *résumé* (*digest* en anglais) de **m**. Pour compliquer les choses, nous voudrions que **f** jouisse des propriétés suivantes :

1. **s** a une longueur fixe (32 octets par exemple), quelle que soit la longueur de **m**, de quelques octets à plusieurs Go,
2. **s** ne donne aucune information sur **m**; elle apparaît comme une suite aléatoire,
3. **f** ne possède pas de fonction inverse : il est impossible de retrouver **m** (ni une partie de **m**) à partir de **s**,
4. la signature d'un élément **m'** *très légèrement* différent de **m** (une différence d'un bit par exemple) est *très fortement* différente de **f(m)**,
5. et, pour rappel, aucun autre élément de **E** ne produit la signature **s**. **f** doit donc être *injective* : si **f(m) = f(m')**, c'est que **m = m'**.

En outre, l'algorithme associé à **f** doit être public. De telles fonctions, dites *hachage cryptographique*, existent effectivement, ou en tout cas s'en rapprochent très fortement.⁶² Basées sur les techniques de hachage étudiées dans ce chapitre, ces fonctions ont de nombreuses applications, notamment dans le domaine de la sécurité. Citons quelques exemples :

- *Authentification d'un document*. Si ce document est accompagné de sa signature, produite par son émetteur légitime, il suffit au destinataire de recalculer celle-ci pour être rassuré sur l'authenticité de son contenu.
- *Vérification de l'intégrité d'un fichier*. Accompagner le fichier à télécharger (un logiciel ou une sauvegarde de données) de sa signature permet à un utilisateur de vérifier que son contenu n'a pas été corrompu ou altéré, accidentellement ou frauduleusement. Il s'agit d'une pratique désormais courante.
- *Recherche de clones de fichiers*. Considérons une collection de plusieurs milliers de fichiers dont on a précalculé les signatures. On se demande si un fichier inconnu n'est pas déjà présent dans cette collection. Plutôt que de comparer, octet par octet, ce fichier à chacun des membres, on calcule sa signature qu'on compare à celles des membres.
- *Protection de mots de passe*. Dans un système de contrôle d'accès à des services internet, il est habituel de stocker dans un fichier le login et le mot de passe des utilisateurs autorisés. Pratique extrêmement dangereuse : un pirate qui se procurerait ce fichier pourrait accéder frauduleusement aux services. Si la seule fonction utile de ce fichier est de vérifier que le mot de passe soumis par l'utilisateur est correct, alors, on y stockera, non pas ce mot de passe en clair, mais sa signature.

62. https://en.wikipedia.org/wiki/Cryptographic_hash_function (juin 2018)

Ces principes et ces applications imposent que la fonction **f** soit *injective*. Or, sauf cas très particuliers, les fonctions de hachage ne le sont pas et conduisent tôt ou tard à des collisions. Une bonne nouvelle cependant : si la fonction **f** est de bonne qualité (elle génère pour **E** une distribution raisonnablement uniforme des signatures) et si l'espace des signatures **S** est très grand, alors la probabilité d'une collision est pratiquement nulle. On dispose aujourd'hui de fonctions de hachage de qualité. Quant à la taille de **S**, en considérant des signatures de 32 octets, ou 256 bits, elle serait de $2^{256} \approx 1,16 \times 10^{77}$ (proche du nombre estimé d'atomes de l'univers !) On dira qu'une telle fonction est *sûre* vis à vis des collisions, d'où le nom anglais *secure hashing algorithm* ou *SHA*. Celle que nous avons évoquée porte le nom de SHA-256. Elle est considérée aujourd'hui comme l'une des meilleures fonctions de hachage cryptographique.

La qualité de **f** se mesure pratiquement par la difficulté de la mettre en défaut, c'est-à-dire de trouver deux éléments de **E** de même signature. En février 2017, Google a annoncé avoir créé deux documents pdf distincts de même signature via la fonction (déjà ancienne) SHA-1 produisant des signatures de 160 bits. Cette découverte, qui a coûté près de 10^{19} calculs de la fonction SHA-1, confirme que cette fonction n'est plus aujourd'hui considérée comme sûre.

4.16.3 Notion d'identifiant universel (UUID)

Nous verrons dans un chapitre ultérieur (section 12.6.4) que les valeurs de l'identifiant primaire d'un fichier ou d'une table peuvent être arbitraires, sans signification pour les utilisateurs. On parlera dans ce cas d'*identifiant technique*. Tel est le cas par exemple de l'identifiant NCOM de la table COMMANDE. La génération de valeurs uniques peut d'ailleurs être confiée au SGBD, lequel, lors de la création d'un enregistrement, garnira automatiquement le champ identifiant d'une valeur non encore utilisée. Un tel champ est parfois dit *autoincrémenté*.

La tendance à la globalisation des données conduit à une extension de ce concept, l'**identifiant universel** (*Universally Unique Identifier* ou UUID), qui sera d'application dans les bases de données massivement distribuées étudiées au chapitre 10.

Lorsque les données sont distribuées sur plusieurs serveurs, la garantie d'unicité de leur identifiant devient plus complexe. Une première solution consiste à créer un **serveur central d'identifiants**, localisé sur un site maître. Avant l'insertion d'un enregistrement par un serveur, celui-ci demande au serveur central de lui créer une valeur unique, qui sera ensuite stockée dans l'enregistrement. Cette architecture pose trois problèmes : (1) le serveur central est un *point unique d'échec* (*single point of failure*), dont la défaillance paralysera l'ensemble du réseau, (2) ce serveur constitue un goulot d'étranglement, les demandes de génération devant être exécutées séquentiellement, (3) les échanges au travers du réseau ralentissent considérablement l'enregistrement de données.

Une seconde solution consiste à implanter sur chaque serveur un **générateur indépendant** qui fournit des valeurs uniques non seulement localement mais aussi globalement pour l'ensemble des serveurs. Deux techniques :

- La valeur est produite à partir des identifiants de composants matériels du serveur : processeur, carte réseau (adresse MAC), carte-mère, disque magnétique;

à cet identifiant partiel s'ajoute une valeur générée unique (horloge système ou générateur aléatoire par exemple). Le résultat contenant des informations sensibles, on l'anonymise par un hachage cryptographique.

- Sur chaque serveur, un générateur de nombres aléatoires fournit des valeurs successives. Grâce à la qualité du générateur, qui produit une distribution (quasi) uniforme des valeurs, et à la grande taille de l'espace de ces nombres (128 ou 256 bits), on obtient l'unicité globale des identifiants générés.

Ces deux familles de techniques posent quelques problèmes. Le premier est celui de la garantie d'unicité globale : les techniques actuelles de hachage cryptographique et de génération de valeurs aléatoires rendent *hautement improbable*, mais pas impossible, la production de **doublons**. Le deuxième problème concerne les **utilisateurs** : de longs identifiants composés de 32 ou 64 caractères hexadécimaux ne sont pas particulièrement aisés à visualiser, manipuler ou mémoriser. Enfin, la production de valeurs successives aléatoires exclut l'utilisation de techniques d'agrégation efficaces lors du stockage. Les index primaires à organisation séquentielle indexée vont rapidement se dégrader pour offrir des **performances** proches de celles des index secondaires.

4.17 FUTUROLOGIE : UN PEU DE POÉSIE DANS CE MONDE BRUTES !

À quoi ressemblera le monde de demain ? Exercice périlleux que de répondre à cette question, particulièrement dans un domaine technologique aussi dynamique que celui des bases de données.

Bien sûr, certaines extrapolations sont sans risque : *la croissance du volume des données va se poursuivre*⁶³ ou encore, *la capacité et la vitesse d'accès des mémoires va continuer à croître*.⁶⁴

Au delà de telles évidences, on plonge dans la science fiction, voire dans la poésie ! Ouvrons-en quelques portes.

4.17.1 Les mémoires ADN

L'ADN est une molécule présente dans les cellules des êtres vivants. Elle est constituée de deux brins enroulés en (double) hélice. Chaque brin est formé d'une chaîne de composants, appelés nucléotides, dont il existe quatre types. Cette succession de composant constitue le code génétique définissant les caractéristiques spécifiques de

63. Selon IBM, la production **journalière** mondiale de données était en 2017 de 2,5 exaoctets, soit 2.500.000 To [www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WRL12345USEN].

Plus pragmatiquement, Google stocke, outre les 30 000 milliards de pages indexées, pour chacun de ses utilisateurs (moteur de recherche, Gmail, Chrome, Android, Google Drive, YouTube, etc.) plusieurs Go de données.

64. En 2018, commercialisation de disques magnétiques de 14 To par Western Digital et annonce d'un SSD de 60 To chez Seagate.

son propriétaire (chaque caractéristique, telle que la couleur des yeux, est décrite par une sous-chaîne de composants). Le code génétique d'un individu (humain, animal, végétal) est suffisant pour le cloner, c'est à dire pour développer un autre individu physiquement identique. Si le clonage d'un dinosaure à partir de l'ADN de son sang extrait d'un moustique piégé dans la résine reste de la science fiction, celui d'un animal disparu depuis des milliers d'années est, lui, tout à fait réaliste, pourvu que son ADN ait été préservé dans des conditions favorables de température, d'obscurité et de neutralité chimique.

Pour ce qui nous concerne ici, une molécule d'ADN peut être modélisée comme une chaîne de symboles équivalant chacun à 2 bits (quatre valeurs possibles). Cette chaîne est structurée en champs (sous-chaînes), décrivant chacune une propriété. D'un point de vue informationnel, une molécule d'ADN apparaît donc comme un enregistrement constitué d'une suite de champs. On estime qu'un gramme d'ADN pourrait théoriquement contenir plus de 200 000 To.⁶⁵ Microsoft a annoncé avoir réussi le codage ADN de 200 Mo de données multimédia.⁶⁶

L'idée n'est pas neuve (elle date de 1964) mais sa réalisation bute encore aujourd'hui sur un problème majeur, celui de l'*écriture*, qui consiste à construire séquentiellement, par un procédé chimique, la chaîne d'ADN codant les données sources. Chez Microsoft, on atteint en 2017 une vitesse de 400 octets par seconde, encore loin des 100 Mo/s nécessaires à la viabilité commerciale. La *lecture* est aujourd'hui bien maîtrisée, puisqu'elle utilise les techniques de séquençage du génome.

En raison de ses qualités (densité, capacité, robustesse, résistance à l'obsolescence technologique), le domaine d'application envisageable des mémoires ADN sera avant tout celui de l'archivage sécurisé de données à long terme.

4.17.2 Les mémoires atomiques

Des recherches sont actuellement conduites dans la perspective d'utiliser certaines caractéristiques d'un atome pour représenter un bit. Techniquement, on retient trois propriétés atomiques à la fois modifiables (permettant l'*écriture*) et mesurables (permettant la *lecture*) : l'état de *charge électrique* (nombre d'électrons en excès ou en défaut), l'état *magnétique* et la *position* dans une grille atomique. Mentionnons les lignes de recherche basées sur les deux dernières propriétés.

Propriétés magnétiques

En réduisant progressivement la taille d'un aimant, on obtient une forme élémentaire d'un aimant de quelques atomes, voire même d'un seul. Un atome en effet jouit lui aussi de propriétés magnétiques, induites par le mouvement de ses électrons (selon le modèle pré-quantique de l'atome). De même que les mémoires magnétiques, disques et bandes, sont constituées de domaines magnétiques microscopiques que les têtes magnétiques peuvent orienter (*écriture*) et mesurer (*lecture*), il est possible de modifier l'orientation de ce groupe d'atomes ou de cet atome et de mesurer celle-

65. https://fr.wikipedia.org/wiki/Acide_désoxyribonucléique#Gènes_et_génome

66. <https://www.seattletimes.com/business/microsoft/uw-microsoft-claim-big-breakthrough-with-data-storage-using-dna/>

ci. Malheureusement la stabilité de telles mémoires diminue avec leur taille, ce qui conduit à travailler dans des conditions extrêmes : températures de l'ordre de 7°K, ultra vide et absence de vibrations. En 2012, une mémoire d'un bit constitué d'un assemblage de 12 atomes a été réalisée. En 2017, l'École polytechnique de Lausanne et IBM expérimentent une mémoire de 2 atomes à quatre états.⁶⁷

Un atome peut donc implémenter une mémoire d'un bit. En excluant les phénomènes subatomiques, il s'agit de la taille ultime d'une mémoire.

Propriétés positionnelles

Déposés sur une surface de cuivre, des atomes de chlore s'alignent selon une grille dont chaque noeud soit contient un atome de chlore soit reste libre. Il est possible de déplacer chaque atome vers un noeud voisin libre à l'aide d'un microscope à effet tunnel. Il est aussi possible de détecter la présence d'un atome à l'emplacement d'un noeud. Chaque noeud constitue donc une mémoire d'un bit, qu'il est possible de modifier (écrire) et d'examiner (lire). Deux obstacles à l'heure actuelle : cette mémoire doit fonctionner à basse température (77°K maximum) et le processus de lecture et d'écriture est très lent (1 Ko par heure). Un prototype fonctionnel d'un Ko a été réalisé en 2016.^{68,69}

4.17.3 Les bases de données quantiques

Développée dès le début du vingtième siècle, la mécanique (ou *physique* ou *théorie*) quantique postule notamment que certains phénomènes ou grandeurs (p. ex. l'énergie) du niveau atomique évoluent, non de manière continue, mais de manière discrète, par *quanta*. Particulièrement riche, elle est à l'origine des principales grande découvertes de la physique moderne.

Elle se caractérise aussi par des hypothèses et des conclusions jugées par le commun des mortels (dont l'auteur et très probablement la plupart des lecteurs) particulièrement contre-intuitives. D'ailleurs, la mécanique quantique a ceci de commun avec la politique belge: *si vous pensez avoir compris, c'est qu'on vous l'a mal expliquée*.⁷⁰

Cet exergue rassurant posé, nous examinerons quelques particularités de cette discipline qui pourraient, à terme, générer des idées d'applications dans notre domaine.

La superposition. Lorsqu'un système peut prendre deux états, tant qu'on ne connaît pas son état actuel, on considère qu'il est dans ces deux états simultanément. Ce principe est illustré par l'expérience (fictive heureusement) du *chat de Schrödinger*.⁷¹ Un chat est enfermé dans une enceinte dans laquelle un dispositif

67. <https://www.nature.com/news/magnetic-hard-drives-go-atomic-1.21599>

68. <https://www.nature.com/news/atom-wrangers-create-rewritable-memory-1.20269>

69. Kalff, F., E. et al. *A kilobyte rewritable atomic memory*. Nature Nanotechnology, vol. 11, Nov. 2016, pp. 926-929, [<https://www.nature.com/articles/nnano.2016.131>]

70. *I think I can safely say that nobody understands quantum mechanics*, Richard Feynman, célèbre physicien américain (1918-1986). Également : *Anyone who is not shocked by quantum theory has not understood it*. Attribué à Niels Bohr, physicien danois (1885-1962), l'un des pionniers de la mécanique quantique.

mortel peut se déclencher selon une certaine probabilité. Après un certain temps, si on ouvre l'enceinte, on peut observer si le chat est mort ou vivant. Avant cette ouverture, l'état du système *enceinte + chat* est indéterminé, ce qu'on traduit ainsi: le système se trouve simultanément dans deux états, dit **superposés**, assortis chacun d'une probabilité : *chat vivant* et *chat mort*. Cet état combiné s'appelle l'*état quantique* du chat. Le chat est ici un *objet quantique*.

L'intrication (*entanglement* en anglais). Ce phénomène concerne deux objets quantiques dont les états sont liés, de sorte que tout changement d'état de l'un est instantanément propagé vers l'autre, et ce, quelle que soit leur distance, de quelques nanomètres à plusieurs années-lumière.⁷²

La décohérence.⁷³ Un état quantique est préservé tant qu'aucune interaction avec l'environnement ne vient le perturber. Dans le cas contraire, l'état quantique s'*effondre* et se réduit à un seul état. Ce phénomène est dénommé la *décohérence*. Tant que l'enceinte n'a pas été ouverte, le chat se trouve dans deux états superposés. Lorsqu'on l'ouvre (ce qui constitue une perturbation), son état quantique se réduit à un seul état. En physique quantique, la curiosité est un vilain défaut !

Deux remarques : (1) ces phénomènes concernent des objets quantiques atomiques et subatomiques (des photons par exemple) et non des objets de la vie courante tels que des chats, (2) pour certains physiciens, dont Stephen Hawking, ces principes ne décrivent pas la réalité mais modélisent la perception que nous en avons.

La mécanique quantique intéresse les constructeurs d'ordinateurs (dont IBM, Intel et Google), qui développent depuis quelques années des prototypes d'*ordinateur quantiques*. Ceux-ci sont basés sur le principe d'une mémoire qui, selon le principe de superposition, possède plusieurs états simultanés. Ainsi, une mémoire quantique d'un octet (dit *octet quantique* ou *qubyte*) peut posséder au même instant un ensemble de 1 à 256 états, chacun représenté par une des configurations classiques d'un octet. Sa capacité est donc celle de 256 octets. La capacité d'une mémoire quantique de 64 octets quantiques est équivalente à 256⁶⁴ octets classiques, un nombre à comparer au nombre d'atomes de l'univers, estimé à 10⁸⁰. Le traitement de l'information ne se fait pas via des algorithmes classiques mais par des opérateurs appliqués à l'ensemble des états de la mémoire. L'un des plus classiques est l'opérateur de Grover⁷⁴, qui recherche parmi un ensemble (non ordonné) de N éléments ceux qui satisfont un critère donné. Le temps de recherche est en O($\sqrt{N}$) au lieu de O(N) pour les architectures classiques. La mémoire quantique est fragile et instable. Elle doit être protégée des accidents de décohérence, notamment par un travail à très basse température (0,015°K à 1°K), par l'isolation maximum de l'environnement et par une redondance permettant de détecter et de corriger les erreurs. La décohérence de consultation (récupération du résultat d'un traitement) peut se

71. Le premier *lolcat* date donc de 1935.

72. Bien que démontré expérimentalement (récemment), ce phénomène a été contesté par Einstein car il supposerait une vitesse de transmission d'information supérieure à celle de la lumière.

73. La décohérence est parfois définie comme l'intrication accidentelle avec l'environnement.

74. https://fr.wikipedia.org/wiki/Algorithme_de_Grover

gérer comme suit : on assigne au résultat une probabilité maximale de sorte c'est lui qui subsistera suite à l'effondrement.

L'application de ces principes aux bases de données apparaît sans trop de difficultés :

- une mémoire quantique peut accueillir un énorme volume de données, bien au-delà des capacités actuelles
- le traitement quantique des données est essentiellement parallèle par opposition aux processus séquentiels induits par l'architecture de von Neumann des machines actuelles
- le principe d'intrication suggère d'étudier la possibilité de modifier à distance des données répliquées, ce qui simplifierait la synchronisation des bases de données distribuées (voir section 5.7 et chapitre 10).

Notons enfin que certains aspects de la mécanique quantique ont déjà été évoqués dans le domaine des bases de données, mais comme métaphore et non comme technique appliquée.⁷⁵

4.18 QUE RETENIR ?

Une base de données est généralement implantée sur un ou plusieurs **disques magnétiques**. L'espace du disque est découpé en **pages adressables**. La lecture et l'écriture du contenu d'une page aléatoire nécessitent un temps de l'ordre de 12 ms. Un **fichier** (ou espace de stockage) est formé d'une suite de pages. Les **lignes des tables** sont rangées dans les pages d'un fichier. Il est possible d'accéder aux lignes successives d'un fichier.

Pour retrouver rapidement la (ou les) lignes qui possèdent une valeur déterminée d'une colonne, on ajoute au fichier de base un **index** sur cette colonne.

Les **index primaires** sont compacts et offrent un accès séquentiel et par clé rapide. En revanche, ils imposent une disposition spécifique des lignes dans leur fichier. Deux techniques d'index primaire sont utilisées : l'organisation **séquentielle indexée** et l'organisation **calculée**.

Les **index secondaires** sont découplés du fichier de base. Ils n'imposent pas de contraintes sur les lignes du fichier et peuvent être ajoutés et supprimés à tout instant. Deux techniques principales : par liste de pointeurs et par bitmap. Par comparaison avec les index primaires, leur volume et leurs temps d'accès sont supérieurs.

75. Par exemple, la référence *Quantum Database* (Roy, S. et al., *6th Biennial Conf. on Innovative Data Systems Research*, CIDR '13, January 6-9, 2013) s'inspire de la notion de *superposition* pour proposer un type de transaction dans laquelle certaines contraintes sont laissées en suspens, générant au *commit* un ensemble d'états possibles qui sont mémorisés. L'état définitif sera précisé plus tard lorsque des informations complémentaires seront disponibles. Ces mêmes auteurs (accompagnés de quelques autres) ont aussi exploré la métaphore de l'*intrication* pour proposer un modèle de coordination d'un ensemble de requêtes SQL (Gupta, N. et al., *Entangled Queries: Enabling Declarative Data-Driven Coordination*, Proc. ACM SIGMOD, June 2011)

Des techniques d'**agrégation** physique permettent d'améliorer les performances de certaines opérations fréquentes : accès par intervalle et par jointure par exemple.

À chacune de ces techniques est associé un **modèle de calcul** des volumes occupés sur le disque et en mémoire centrale ainsi que des temps d'accès.

Les techniques de hachage trouvent également des applications dans les **prédicteurs** (*filtres de Bloom*), le **hachage cryptographique** (*secure hashing*) et la génération d'**identifiants universels** (*UUID*).

Les recherches se poursuivent dans les techniques capables de stocker de gigantesques volumes de données. L'amélioration des technologies actuelles permettra bien sûr d'en augmenter progressivement la capacité, mais de nouvelles voies de recherche tentent de développer des supports dont la capacité est de plusieurs ordres de grandeurs supérieure à celles des disques magnétiques et SSD actuels. Il apparaît cependant que les voies explorées sont encore longues et semées d'embûches !

4.19 POUR EN SAVOIR PLUS

Ce chapitre n'a fait qu'effleurer l'étude des techniques d'implémentation des structures de données et de leur gestion. Nous n'avons décrit que les éléments qui nous seront nécessaires pour la mise en œuvre physique d'une base de données. Le lecteur trouvera un développement très détaillé des fondements de cette discipline dans la série *The art of computer programming* de D. Knuth, et particulièrement dans le 3^e volume, *Sorting and Searching* [Knuth, 1998]. D'un abord plus aisé, la référence [Garcia-Molina, 2008] sera aussi fortement recommandée.

L'étude des techniques d'implémentation des structures de données et de leur gestion a fait l'objet de très nombreux travaux. Ceux-ci cependant s'adressent plus aux développeurs de SGBD qu'à leurs utilisateurs. On en trouvera les résultats notamment dans les actes des conférences VLDB (*Core track*), ICDT, EDBT et SIGMOD/PODS.

4.20 EXERCICES

- 4.1 On considère une table dont les lignes sont rangées dans un espace qui lui est exclusivement réservé et constitué de pages de 4 Ko. Chaque ligne occupe une seule page. Cet espace est implanté sur un disque de notre modèle de référence (figure 4.6). La table contient 1 000 000 lignes d'une longueur de 200 octets. Le taux d'occupation moyen des pages est de 75 %. Calculer le volume minimal de cet espace de stockage et le temps de lecture séquentielle de toutes les lignes de cette table.
- 4.2 Une table APPEL, représentant des appels téléphoniques, comporte 6 000 000 lignes d'une longueur fixe de 200 octets. Elle dispose d'un identifiant constitué de la colonne ID de 40 caractères, sur laquelle on définit un index. La table est stockée dans un espace qui lui est réservé, implanté sur un disque du modèle de référence, décomposé en pages de 8 Ko. On envisage trois techniques d'implémentation de l'index sur ID : index primaire en séquentiel

indexé, index primaire en calculé, index secondaire sur fichier en vrac. Calculer dans chaque cas le volume minimal de l'espace de stockage et le temps d'accès via ID.

- 4.3 Soit un fichier calculé dont $L_r = 400$ octets, $L_p = 4Ko$ et rempli à 90 %. Quel est le temps moyen de lecture de l'enregistrement $K = k$?
- 4.4 Quel est le taux d'occupation à partir duquel il convient de réorganiser un fichier calculé dont $Mrpp = 20$ si le temps moyen d'accès par clé ne peut dépasser 15 ms ?
- 4.5 Quelle taille de page minimum préconiser pour un fichier calculé dont le temps d'accès ne peut dépasser 1,15 accès physique au taux d'occupation de 90 % ?

Exercices corrigés

- **Exercice 4.1.** Capacité d'une page : $Mrpp = \lfloor L_p / L_r \rfloor = 20$ enregistrements par page. Contenu effectif d'une page : $Nrpp = \tau \times Mrpp = 15$ enregistrements par page. Volume minimal de l'espace : $Np = \lceil Nr / Nrpp \rceil = 66\,667$ pages ou **261 Mo**. Temps de lecture séquentielle de la table (hypothèse de lecture anticipée d'une piste, soit $t_{is1} = 0,184$ ms par page) : $t_{is} = t_{is1} \times Np = 12,27$ s.
- **Exercice 4.2.** On admet que dans les trois organisations le taux d'occupation moyen du fichier de base est $\tau_b = 0,8$. On calcule : $Mrpp = \lfloor L_p / L_r \rfloor = 40$ enregistrements et $Nrpp = \tau_b \times Mrpp = 32$ enregistrements. La taille du fichier de base est donc $Npb = \lceil Nr / Nrpp \rceil = 187\,500$ pages.

Index primaire en séquentiel indexé (SI)

Compte tenu d'un taux de remplissage des pages d'index de $\tau_i = 0,8$ et de pointeurs de page de 4 octets, on calcule $L_i = 44$ octets, $Mipp = \lfloor L_p / L_i \rfloor = 186$ entrées par page et $Nipp = \tau_i \times Mipp = 148,8$ entrées par page. On en déduit $Npi \approx Npi(n) = 1\,261$ pages. Nombre de niveaux : $n = \lceil \log_{Nipp} Npb \rceil = 3$. Taille du fichier $Np = Npb + Npi = 187\,500 + 1\,261 = 188\,761$ pages.

Le temps d'accès via l'identifiant est celui de 4 lectures de pages, soit 49,2 ms. Compte tenu d'un verrouillage des deux premiers niveaux d'index (10 pages) dans le tampon, ce temps tombe à 2 lectures de pages, soit 24,6 ms. Si le tampon peut accueillir tout l'index, le temps d'accès n'est plus que de **12,3 ms**.

Index primaire en calculé

On interprète τ_b comme le taux de chargement τ_{ch} . Les 6 000 000 enregistrements sont envoyés vers $Npb = 187\,500$ pages. Les abaques donnent, pour la courbe $Mrpp = 40$ et pour $\tau_{ch} = 80\%$, un taux d'accroissement $\tau_{ap} = 1,085$ et un coût d'accès $n_{ip} = 1,01$. Le nombre total de pages est $Np = \lceil \tau_{ch} \times Npb \rceil = 203\,438$. Le temps d'accès moyen est de $1,01 \times 12,3 = 12,4$ ms. L'accès ne nécessite pas plus d'une page dans le tampon.

Index secondaire sur fichier en vrac

Le dictionnaire de valeurs de l'index contient $Nv = 6\,000\,000$ entrées de $L_v = 46$ octets (un pointeur d'enregistrement de 6 octets par entrée). On a donc $Mvpp = \lfloor L_p / L_v \rfloor = 178$ entrées par page et $Nvpp = \tau_v \times Mvpp = 142,4$ entrées par page

($\tau_v = 0,8$). On en déduit la taille du dictionnaire : $N_{pv} = \lceil N_v / N_{vpp} \rceil = 42\,135$ pages. On traite ensuite ce dictionnaire comme un fichier de base organisé en séquentiel indexé. Son index primaire se calcule de manière classique. On reprend les grandeurs dont nous disposons déjà : $L_i = 44$; $M_{ipp} = 186$; $N_{ipp} = 148,8$. On a aussi $N_{pi} \equiv N_{pi}(n) = 284$ pages et $n = 3$. Le volume de l'index secondaire est $N_{ps} = N_{pv} + N_{pi} = 42\,419$ pages. La taille totale du fichier est de $N_p = N_{pb} + N_{ps} = 187\,500 + 42\,419 = 229\,919$ pages.

Le temps d'accès via l'identifiant est de 5 lectures de pages (3 pour l'index du dictionnaire + 1 pour le dictionnaire + 1 pour le fichier de base), soit 61,5 ms. Si on verrouille les deux premiers niveaux d'index du dictionnaire dans le tampon, ce temps est de 3 lectures de pages, soit 36,9 ms. Si le tampon contient la totalité de l'index du dictionnaire (287 pages), le temps d'accès est de 2 lectures de pages, soit de 24,6 ms. Il est difficile de diminuer ce temps.

En résumé

	volume index	volume tampon	temps d'accès
Primaire SI	10 Mo	0,08 à 10 Mo	24,6 à 12,3 ms
Primaire calculé	0	0,008 Mo	12,4 ms
Secondaire	332 Mo	0,024 à 2,25 Mo	36,9 à 24,6 ms

- **Exercice 4.3.** La capacité des pages est de $M_{rpp} = \lfloor L_p / L_r \rfloor = 5$ enregistrements. L'abaque nous donne, à l'abscisse 90, le nombre d'accès $n_k = 1,29$, soit un temps d'accès de $n_k \times t_{ia1} = 15,9$ ms. La vitesse de lecture est donc de $1/0,0159 = 62$ enregistrements par seconde.
- **Exercice 4.4.** Un temps de 15 ms correspond à $0,015/t_{ia1} = 1,22$ accès physiques. Pour $M_{rpp} = 20$ et $n_k = 1,22$ l'abaque nous donne un taux d'occupation de 96 %. Ce taux étant très élevé, on réorganisera le fichier bien avant de l'atteindre.
- **Exercice 4.5.** Pour ces données, l'abaque nous indique une valeur M_{rpp} comprise entre 10 et 20 mais plus proche de 20. La fonction $n_k = f(M_{rpp})$ présentant une allure logarithmique inverse, on estime la taille recherchée à $N_{rpp} = 15$.

Des matériaux complémentaires et des exercices additionnels sont disponibles à l'annexe A4.

Chapitre 5

Les systèmes de gestion de bases de données

Ce chapitre décrit les principes des logiciels chargés de gérer les bases de données, les *systèmes de gestion de bases de données* ou SGBD. On y étudie leur architecture et leurs fonctions. On détaille en particulier la notion de transaction monosite et distribuée, la régulation de la concurrence, la protection contre les incidents et le contrôle des accès. On décrit le marché des SGBD et on fournit quelques recommandations sur l'utilisation personnelle d'un SGBD.

5.1 INTRODUCTION

La gestion d'une base de données est à la fois complexe et diversifiée. Maintenir la structure et l'intégrité des données, garantir l'accès aux données en un temps minimum, gérer l'espace occupé de manière optimale, protéger les données des effets des accidents de toute nature, autoriser plusieurs centaines de programmes à accéder aux mêmes données et à les modifier, contrôler les accès selon les autorisations strictement définies sont quelques-unes des fonctions qu'on attend des logiciels (ou **systèmes**) **de gestion de bases de données** ou **SGBD**. Les chapitres précédents ont montré que certaines de ces fonctions pouvaient s'avérer très complexes. Nous verrons en outre lors de l'étude du langage SQL, de nombreuses références aux fonctions et aux composants des SGBD relationnels en particulier, et de tout SGBD en général.

Nous étudierons dans ce chapitre les principes généraux des SGBD, depuis leur architecture jusqu'au détail de certaines de leurs fonctions essentielles. Celles-ci sont généralement liées au concept de transaction, que nous décrirons brièvement.

5.2 LES FONCTIONS DES SYSTÈMES DE GESTION DE BASES DE DONNÉES

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 de langages de programmation qui permettent le développement rapide de programmes complexes. Ces logiciels exigent des ressources importantes en matériel : mémoire centrale, vitesse du processeur, disques rapides et à grande capacité. Ils réclament aussi de la part des utilisateurs et des développeurs qui désirent exploiter leurs fonctions de manière optimale des connaissances plus ou moins approfondies. Ces exigences se conçoivent aisément quand on sait qu'une base de données peut comporter plusieurs milliers de tables et plusieurs dizaines de milliers de colonnes, et qu'elle peut desservir plusieurs milliers d'utilisateurs selon des modalités personnalisées. 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 ou une colonne dans une base de données. Des contraintes d'intégrité, standard ou personnalisées, peuvent être définies sur les données. Les SGBD permettent de les exprimer via différentes techniques dont les déclencheurs et les procédures SQL (chapitre 9).

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 (telles que les index) qui seront utilisées pour atteindre ces données. 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 résultat 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) Accès par programme

Les commandes SQL de définition, d'extraction et de modification de données sont généralement transmises au SGBD par des programmes d'application. Il existe plusieurs modes d'interaction entre un programme et le SGBD qui seront étudiés dans la section 9.11.

e) Personnalisation des données extraites : les vues

Le schéma d'une base de données peut être très volumineux et très complexe. Un utilisateur particulier désirera généralement qu'on ne lui en présente que les parties qui sont réellement pertinentes. Ces parties forment une vue sur les données. Une vue inclut des tables et des colonnes contenant les données extraites de la base de données mais aussi des données dérivées, correspondant aux besoins réels de l'utilisateur. Le SGBD gère les correspondances entre une vue et les tables réelles.

f) 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. Le SGBD permet aussi de retirer un privilège accordé.

g) Protection contre les accidents

Des mécanismes du SGBD de protection contre les accidents garantissent que les données ne seront pas corrompues à la suite d'événements affectant le fonctionnement de la configuration informatique (serveurs, réseau, postes de travail, logiciels).

h) Gestion des accès concurrents

Le SGBD permet à plusieurs programmes et utilisateurs de consulter et de modifier simultanément les mêmes données sans que cette concurrence entraîne de détérioration des données.

i) Distribution et réplication des données

Pour des raisons d'accessibilité et de sécurité, on peut être amené à répartir les données et à les répliquer sur plusieurs serveurs, éventuellement distribués géographiquement. Le SGBD assure la gestion synchronisée et l'interrogation de ces données, et ce, de manière transparente pour les applications clientes.

j) **Optimisation et autonomie de fonctionnement**

La complexité des SGBD, conséquence des exigences croissantes en termes de fonctionnalités, de performances et de sécurité, rend leur maîtrise par les ingénieurs de plus en plus difficile. Certains SGBD intègrent aujourd'hui des fonctions autonomes de contrôle, d'optimisation, de maintenance et d'évolution.

k) **Dictionnaire de données**

Le SGBD gère un *dictionnaire de données* (ou *catalogue*), qui contient la description précise des composants du schéma de la base de données. Ce dictionnaire est utilisé par le SGBD mais est également accessible à l'utilisateur.

5.3 ARCHITECTURE DES SGBD

Un SGBD est ce qu'on appelle un *serveur*. Il exécute des *services* pour le compte de programmes d'application, dits *programmes clients*, dont le fonctionnement nécessite l'enregistrement, la consultation et, plus généralement, la gestion du contenu d'une base de données.

Classiquement, un programme d'application P s'exécute sur une machine dénommée *serveur d'application*, tandis que le SGBD tourne sur une autre machine appelée *serveur de données*. Ces deux machines ont des besoins en logiciel et en matériel assez différents, de sorte que leurs configurations sont le plus souvent également différentes. Dans cette architecture, dite **client-serveur**, il existe généralement une troisième machine, la *station cliente*, sur laquelle tourne un navigateur via lequel un utilisateur (humain) interagit avec le programme d'application. On parle alors d'une architecture *à trois couches* ou *3-tiers architecture*. Rien n'interdit cependant qu'on installe le programme d'application, le SGBD et le navigateur sur deux machines seulement, voire sur une seule : l'architecture garde néanmoins sa structure *à trois couches*. Notons que les concepts de *client* et de *serveur* correspondent à des rôles : P est un client du SGBD puisqu'il lui demande des services mais il est aussi un serveur pour la station cliente¹, puisqu'il lui offre des services.

Ces trois composants logiciels dialoguent au travers de canaux de communication de nature quelconque (figure 5.1). Chaque programme d'application P envoie au SGBD des *demandes de services*, par exemple sous la forme de requêtes SQL ("select NOM from CLIENT where NCLI = 'C400'"). En réponse, et si cette demande est correcte et légitime, le SGBD l'exécute et retourne au programme le résultat de cette exécution, soit les données demandées ('FERARD') et un diagnostic sur le déroulement de l'opération (OK).

1. Le terme de *serveur* désigne tantôt un logiciel ou un processus assurant un ensemble de services, tantôt la machine qui exécute ce logiciel.

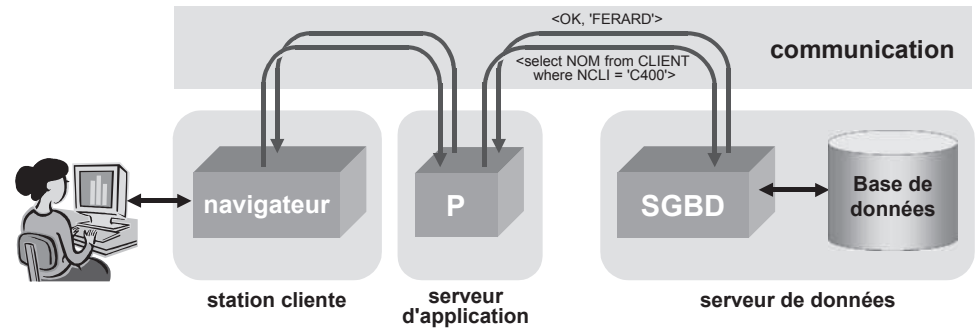


Figure 5.1 - Architecture typique d'une application à trois composants

Examinons à présent le fonctionnement du SGBD lorsqu'il reçoit une requête. Considérons à titre d'exemple la demande d'exécution par le programme client P de la requête de consultation mentionnée précédemment.

Observons d'abord que la base de données est constituée de deux composants. Les **données** proprement dites, organisées comme nous l'avons vu au chapitre précédent, et les tables du **catalogue** (figure 5.2). Ces dernières forment une petite base de données dont le contenu décrit le schéma de la base de données principale. On y trouve une table dont chaque ligne décrit une table, une table dont chaque ligne décrit une colonne, une table dont chaque ligne décrit un identifiant ou une clé étrangère, etc. Nous étudierons ces tables plus en détail à la section 9.10 (figures 9.4 et 9.5). Le SGBD va consulter le catalogue pour analyser et traiter les requêtes qu'il reçoit. L'exécution d'une requête se passe comme suit.

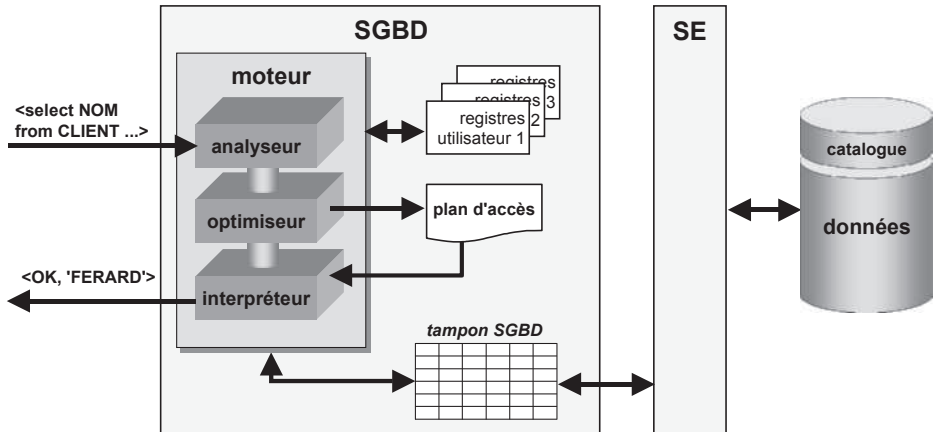


Figure 5.2 - Composants du SGBD et flux de données

1. P envoie la requête au SGBD.
2. Le SGBD **analyse** la requête. Il vérifie si elle est syntaxiquement correcte, puis si elle est pertinente : il existe bien une table nommée "CLIENT", celle-ci possède des colonnes nommées "NOM" et "NCLI", les valeurs de NCLI sont des chaînes

d'au plus 4 caractères, P a l'autorisation de lire ces données. Ces renseignements sont fournis par les tables du catalogue.

3. Le SGBD **traduit** la requête en un **plan d'accès** efficace constitué d'une suite d'opérations sur les données. Ce plan profite de l'existence d'un index primaire associé à la table CLIENT et défini sur la colonne NCLI. L'optimiseur consulte lui aussi les tables du catalogue.
4. Le plan d'accès est **exécuté** par un composant qui en *interprète* les opérations. Il comporte notamment l'accès à l'index primaire, suivi de l'accès à la page de la table qui contient la ligne, suivi de l'analyse du contenu de la page pour en extraire la valeur de NOM de la ligne identifiée.
5. La valeur extraite et le diagnostic sont transmis à P.

Il est utile de rappeler le rôle du tampon. Lorsque le SGBD envoie au système d'exploitation (SE) une demande de lecture ou d'écriture d'une page de la base de données, cette page est d'abord copiée dans le tampon. Ceci concerne l'accès aux pages de données et à celles du catalogue, lesquelles sont utilisées de manière intensive. Chaque session de travail ouverte par un programme client est décrite sous la forme d'un ensemble de registres. Ceux-ci mémorisent l'état actuel de la session : l'identification de la session, celle du programme client, la requête en cours d'exécution, l'état des curseurs, les espaces de stockage en ligne, etc.

Cette architecture se décline de différentes manières selon les SGBD. Par exemple, la phase de traduction (analyse et optimisation) est dans certains cas effectuée préalablement à l'envoi de la requête. Tel est le cas d'une interface SQL statique (voir section 9.11.1), qui fait intervenir un précompilateur spécifique des programmes clients. Le plan d'accès est alors précalculé et stocké dans le catalogue. Les registres peuvent être en tout ou en partie attachés au programme client et non localisés dans le SGBD. Enfin, étant donné son utilisation intensive, les tables du catalogue sont parfois stockées dans un espace dédié du SGBD plutôt que de manière anonyme dans le tampon. Les principes cependant restent similaires. Ajoutons encore que certaines fonctions assurées par le SGBD, et que nous étudierons plus loin dans ce chapitre, réclament des composants et des données spécifiques que nous avons ignorés. On pense en particulier à la gestion de journaux (5.6), à la gestion des verrous (5.5) et au contrôle des accès (5.8).

5.4 LES TRANSACTIONS

Un utilisateur ou un programme d'application interagit avec la base de données via des appels au SGBD exprimés sous forme de requêtes SQL. Le SGBD garantit une *qualité de service* absolue dans l'exécution de ces requêtes, garantie qui comporte quatre propriétés : *atomicité*, *cohérence*, *isolation* et *durabilité* (ou ACID).

- **Atomicité.** L'exécution d'une requête est atomique si cette dernière, quoi qu'il arrive, est exécutée complètement (en cas de réussite) ou pas du tout (en cas d'échec ou d'incident). Cette propriété peut sembler anodine, mais c'est loin

d'être le cas, et sa garantie réclame des techniques complexes et coûteuses. Par exemple, la suppression d'une seule ligne de la table CLIENT peut exiger la lecture et l'écriture de plusieurs centaines de pages sur le disque (pensons à l'intégrité référentielle en mode cascade et aux multiples index en jeu). Un incident qui se produirait au milieu de cette séquence d'opérations affecterait immanquablement l'intégrité des données sur le disque. Le SGBD garantit que, si un tel incident devait survenir, soit les données sont restaurées dans leur état précédant l'exécution de la requête, soit des modifications complémentaires sont effectuées après l'incident pour achever le travail.

- **Cohérence.** Une requête de modification exécutée sur des données cohérentes laisse celles-ci dans un état final également cohérent. Concrètement, le SGBD garantit le respect de toutes les contraintes d'intégrité imposées aux données. En cas de tentative de violation, le SGBD annule les modifications (mode *bloquant*) ou effectue des modifications complémentaires pour atteindre un nouvel état cohérent (mode *cascade* par exemple).
- **Isolation.** Les opérations sur les données sont exécutées comme si chaque requête disposait de la base de données pour elle seule. Il n'y a (en apparence du moins) aucune interférence nuisible entre les exécutions simultanées de requêtes, même lorsqu'elles traitent les mêmes données. Cette propriété simplifie fortement l'écriture des programmes, qui n'ont pas à se préoccuper des opérations concurrentes, qu'il s'agisse de lectures ou d'écritures.
- **Durabilité.** Lorsque le SGBD a confirmé qu'une mise à jour a été effectuée avec succès, il garantit qu'elle est permanente. Aucun incident survenant ultérieurement, quelle qu'en soit la gravité (y compris la destruction physique du disque, voire du serveur !), ne peut entraîner l'annulation de cette modification.

Pour essentielles qu'elles sont, ces garanties associées à l'exécution des requêtes individuelles sont insuffisantes en pratique. Montrons-le par une analyse de la demande de service adressée à une banque et qui consiste à **transférer** un montant M d'un compte C1 vers un compte C2. Le langage SQL n'autorisant pas la formulation en une seule requête d'une opération aussi complexe, celle-ci va se décomposer en une séquence d'opérations plus élémentaires : lire la ligne du compte C1, lui retirer le montant M, recopier cette ligne, lire la ligne du compte C2, lui ajouter le montant M, recopier cette ligne. Ce développement est particulièrement vulnérable aux incidents qui pourraient se produire. Par exemple, une panne de serveur intervenant après la première recopie mais avant la seconde entraînerait une grave détérioration des données, au détriment du propriétaire du compte C1, qui aurait perdu le montant M, retiré de C1 mais pas encore transféré sur C2.

C'est pour éviter de tels problèmes que le concept de **transaction** a été développé. Une transaction est une suite d'opérations constituant une unité logique de traitement et pour laquelle on exige **globalement** les garanties ACID. Une transaction correspond donc à une opération complexe *atomique, cohérente, isolée* et à effet *durable*. Le transfert d'un montant entre deux compte se traduit par une transaction formée des opérations élémentaires décrites précédemment. Contrairement aux opérations élémentaires, le SGBD est incapable d'identifier les transactions

dans un flux de requêtes émises par un utilisateur ou un programme. Il faut donc lui signaler le début et la fin de chaque transaction à l'aide de requêtes spéciales. En pratique un programme contiendra une section de code telle que la suivante, exprimée dans un pseudo-code qu'on considérera comme intuitif²:

```
getInputData (C1, C2, M) ; [1]
begin_transaction; [2]
    select MONTANT into :M1 from COMPTE where NCPT = :C1; [3]
    M1 := M1 - M; [4]
    update COMPTE set MONTANT = :M1 where NCPT = :C1; [5]
    select MONTANT into :M2 from COMPTE where NCPT = :C2; [6]
    M2 := M2 + M; [7]
    update COMPTE set MONTANT = :M2 where NCPT = :C2; [8]
commit_transaction; [9]
<suite du programme> [10]
```

Ce fragment demande à l'utilisateur de lui fournir deux numéros de compte (C1 et C2) et un montant (M) [1]. Il ouvre ensuite une transaction [2], dans laquelle il lit la ligne du compte C1 [3], retire M de son montant [4], recopie la ligne dans la base de données [5], puis il lit la ligne du compte C2 [6], ajoute M à son montant [7] et recopie la ligne dans la base de données [8]. Il indique enfin la fin de la transaction [9]. Une transaction réelle est souvent plus complexe, comportant notamment des opérations de validation des données. En cas d'erreur, le programme peut demander l'annulation de la transaction (instruction `abort_transaction`). Nous allons à présent examiner dans les sections 5.5 et 5.6 comment le SGBD garantit les qualités ACID d'une transaction.

5.5 RÉGULATION DE LA CONCURRENCE

On dit que des transactions³ sont **concurrentes** lorsqu'elles accèdent (ou sont susceptibles d'accéder) en même temps aux mêmes données. Si l'une de ces transactions effectue une modification de ces données partagées, alors des problèmes d'interférences nuisibles risquent de survenir : corruption des données ou perception erronée des données par une transaction. Nous décrirons deux des scénarios parmi les plus fréquents, *la mise à jour perdue*, qui conduit à la corruption des données et *les données instables*, qui induisent une perception erronée des données, laquelle peut ensuite corrompre les données.

2. Les instructions SQL utilisées dans les exemples qui vont suivre seront décrites en détail dans les chapitres suivants. Elles sont cependant suffisamment simples pour que le lecteur puisse dès maintenant deviner leur structure.

3. En toute rigueur, il faut distinguer le *code d'une transaction* (la séquence d'instructions) de l'*exécution d'une transaction* (une séquence d'opérations). Les praticiens ont tendance à utiliser le terme de transaction pour désigner ces deux concepts. Le contexte permet en général de lever cette ambiguïté.

5.5.1 Mise à jour perdue

Considérons une transaction T1 qui ajoute une somme de 1 000 au compte C1 et une transaction T2 qui ajoute une somme de 500 à ce même compte.

Au total une somme de 1 500 doit être ajoutée à C1. Le code de ces transactions est constitué comme suit.

```
select MONTANT into :M1 from COMPTE where NCPT = :C1;      [1]
M1 := M1 + <somme>;                                         [2]
update COMPTE set MONTANT = :M1 where NCPT = :C1;          [3]
```

On peut envisager une **exécution séquentielle** de ces transactions : la transaction T1 est exécutée complètement avant que la suivante, T2, soit démarrée. Les opérations correspondraient à l'exécution des instructions suivantes :

```
select MONTANT into :M1 from COMPTE where NCPT = :C1;      [1:1]
M1 := M1 + 1000;                                           [1:2]
update COMPTE set MONTANT = :M1 where NCPT = :C1;          [1:3]

select MONTANT into :M1 from COMPTE where NCPT = :C1;      [2:1]
M1 := M1 + 500;                                           [2:2]
update COMPTE set MONTANT = :M1 where NCPT = :C1;          [2:3]
```

Le résultat global est parfaitement correct : le compte C1 a été augmenté de 1 500. Une exécution séquentielle conduit à des performances médiocres, de sorte qu'en pratique les transactions s'exécutent **en parallèle**, de manière concurrente. Leurs opérations élémentaires font l'objet d'une **exécution entrelacée**, dans laquelle les opérations des transactions sont mélangées, par exemple comme dans la séquence suivante :

```
select MONTANT into :M1 from COMPTE where NCPT = :C1;      [1:1]
M1 := M1 + 1000;                                           [1:2]
select MONTANT into :M1 from COMPTE where NCPT = :C1;      [2:1]
M1 := M1 + 500;                                           [2:2]
update COMPTE set MONTANT = :M1 where NCPT = :C1;          [1:3]
update COMPTE set MONTANT = :M1 where NCPT = :C1;          [2:3]
```

On constate immédiatement que le résultat global est erroné, le montant du compte n'étant augmenté que de 500. La seconde mise-à-jour [2:3] a annulé la première [1:3], qui a donc été **perdue**. Le SGBD ne se limite pas à une exécution séquentielle pour des raisons de performances mais il ne peut laisser se dérouler de manière non contrôlée les exécutions entrelacées pour des raisons d'intégrité des données.

5.5.2 Données instables⁴

Une transaction T1 se propose d'ajouter une somme (par exemple 500) au compte d'un client. Cependant, après que la modification ait eu lieu, la transaction, observant qu'une condition <C> n'est pas satisfaite, se ravise et annule cette modification. Son code pourrait être le suivant :

4. En anglais : *dirty data* ou *read uncommitted*.


```

select MONTANT into :M1 from COMPTE where NCPT = :C1;      [1]
M1 := M1 + 500;                                           [2]
update COMPTE set MONTANT = :M1 where NCPT = :C1;         [3]
if not <C> then abort-transaction                         [4]

```

Une autre transaction T2, plus simple, se contente d'ajouter la somme de 300 au compte d'un client. Son code se présente comme suit :

```

select MONTANT into :M1 from COMPTE where NCPT = :C1;      [1]
M1 := M1 + 300;                                           [2]
update COMPTE set MONTANT = :M1 where NCPT = :C1;         [3]

```

Considérons à présent l'exécution entrelacée ci-après, dans laquelle les transactions accèdent au même compte :

```

select MONTANT into :M1 from COMPTE where NCPT = :C1;      [1:1]
M1 := M1 + 500;                                           [1:2]
update COMPTE set MONTANT = :M1 where NCPT = :C1;         [1:3]
select MONTANT into :M1 from COMPTE where NCPT = :C1;      [2:1]
not <C> => abort-transaction                             [1:4]
M1 := M1 + 300;                                           [2:2]
update COMPTE set MONTANT = :M1 where NCPT = :C1;         [2:3]

```

Suite à l'annulation de T1, le compte aurait dû n'être augmenté que de 300 mais l'a été de 800. L'erreur est due au fait que T2 a été autorisée à lire l'état instable, non confirmé, d'une donnée traitée par T1 : perception erronée des données entraînant une corruption de celles-ci.

5.5.3 Régulation de la concurrence par verrouillage

Les SGBD disposent de plusieurs techniques pour empêcher des transactions concurrentes de travailler sur les mêmes données. Nous allons décrire la plus répandue, celles du **verrouillage**.

Une transaction peut demander au SGBD de poser un verrou sur une donnée D (une valeur, une ligne, une page, une table, etc.). Il existe principalement deux types de verrous : *partagés* (ou en lecture) et *exclusif* (ou en modification). Les règles d'attribution et de gestion de verrous sont les suivantes :

- Avant de *lire* une donnée D, la transaction doit poser un verrou partagé ou exclusif sur D.
- Avant de *modifier* une donnée D, la transaction doit poser un verrou exclusif sur D.
- Lorsqu'une transaction n'utilise plus une donnée D, elle demande la suppression du verrou posé sur D.
- Une transaction peut poser un verrou partagé sur D si aucun verrou exclusif n'y a été posé par une autre transaction.
- Une transaction peut poser un verrou exclusif sur D si aucun verrou (partagé ou exclusif) n'y a été posé par une autre transaction.

- des verrous peuvent être posés sur un ensemble de lignes spécifié par une clause *where* (par exemple, les comptes des clients de Toulouse). On les nomme *verrous d'intervalle* (*range lock*).

Donc, si une transaction a posé sur D un verrou partagé, une autre transaction peut lire D mais pas la modifier. Si une transaction a posé sur D un verrou exclusif, aucune autre transaction ne peut lire ni modifier D tant que ce verrou n'a pas été levé.

En pratique, une transaction T1 qui se voit refuser un verrou sur D, parce qu'une autre transaction T2 y a déjà posé un verrou, est **suspendue** par le SGBD jusqu'à la libération par T2 de la donnée demandée. Il n'y a en général pas de primitive explicite de demande de verrouillage d'une donnée. La demande est implicite lors de l'exécution des requêtes de lecture et de mise à jour. Cependant, une transaction qui lit une donnée dans l'intention de la modifier par la suite peut demander un verrou exclusif au moment de la lecture⁵. Lors d'une demande de modification d'une données déjà lue, le verrou partagé est, si la chose est possible, transformé en verrou exclusif. La demande de levée d'un verrou est également implicite : tous les verrous sont levés à la fin de l'exécution de la transaction.

5.5.4 Sérialisabilité et verrouillage en 2 phases (2PL)

La technique du verrouillage ne garantit malheureusement pas qu'un ensemble de transactions concurrentes se déroulent de manière correcte. On introduit alors la notion d'**exécution sérialisable** d'un ensemble $T = \{T1, T2, \dots, Tn\}$ de transactions concurrentes. Ce terme désigne une exécution (entrelacée ou non) des opérations des transactions de T dont l'effet global est le même qu'une quelconque des exécutions séquentielles de T (par exemple T2, puis T1, ... et enfin Tn).

Il existe plusieurs méthodes d'organisation de l'exécution des transactions qui en garantissent la sérialisabilité. Le protocole de **verrouillage en 2 phases** (*2-phase locking*) est le plus connu. La gestion des verrous au sein d'une même transaction procède en deux phases : une *phase croissante* de demandes de verrous (dont aucun n'est relâché) suivie d'une *phase décroissante* de déverrouillage (sans nouvelle demande de verrouillage). On ne peut donc plus demander de verrous après qu'on en ait levé un. Ce protocole est peu coûteux et efficace mais il est souvent plus contraignant qu'il n'est nécessaire, réduisant inutilement le parallélisme des transactions. Dans la variante la plus souvent implémentée par les SGBD (*strong strict two-phase locking*), la phase 2 est exécutée au moment de la clôture de la transaction. Elle est donc implicite.

5.5.5 Interblocage

La technique de sérialisation par verrouillage peut entraîner des anomalies d'exécution des transactions, dont l'**interblocage**⁶ est le plus spectaculaire. Le cas de base

5. `declare cursor for <expression SFW> for update.` Voir section 9.11.1

6. Ou, plus poétiquement, *verrou mortel* (*deadlock*) ou *étreinte mortelle* (*deadly embrace*).

est celui de deux transactions T1 et T2 demandant chacune la modification des données D1 et D2. Le scénario critique se déroule comme suit :

1. T1 demande un accès à D1 avec verrou exclusif : demande satisfaite ;
2. T2 demande un accès à D2 avec verrou exclusif : demande satisfaite ;
3. T1 demande un accès à D2 avec verrou exclusif : demande refusée, T1 en attente ;
4. T2 demande un accès à D1 avec verrou exclusif : demande refusée, T2 en attente.

La situation est bloquée : chaque transaction est suspendue, dans l'attente de la terminaison de l'autre. Le traitement se fait par *prévention* ou *correction*. Dans le premier cas, le SGBD évite les situations qui risquent de provoquer un interblocage (ici, T2 ne pourra pas démarrer). Selon l'approche corrective, la plus fréquente, le SGBD laisse l'interblocage se produire, le détecte puis annule (abort forcé) une des transactions bloquées de manière à laisser les autres poursuivre leur exécution. La transaction annulée est ensuite reprise après un certain délai.

5.5.6 Les niveaux d'isolation

Les protocoles garantissant la sérialisabilité, particulièrement assortis du verrouillage en deux phases strict (libération au `commit`), sont particulièrement sévères et peuvent entraîner le blocage de données pendant une durée excessive, réduisant par là le parallélisme des transactions concurrentes. Par exemple, le simple calcul du total des montants des commandes par localité va poser des verrous en lecture sur la plupart des lignes des quatre tables⁷ de notre base de données de référence, immobilisant ainsi toutes les opérations de modification pendant plusieurs minutes. Le standard SQL2 définit, outre la sérialisabilité, trois niveaux d'isolation plus faibles, qui nécessitent moins de verrous et qui permettent donc plus de parallélisme mais qui offrent moins de protection contre les interférences nuisibles.

- **lecture reproductible** (*repeatable read*) : au sein de la transaction, plusieurs lectures de la même donnée doivent livrer la même valeur. Cependant, ce niveau ne gère pas les verrous d'intervalle, de sorte qu'au cours de l'exécution de la transaction, une transaction concurrente peut créer de nouveaux enregistrements qui ne seront pas pris en compte. Cette *ignorance* porte le nom de *lecture fantôme* (*phantom read*).
- **lecture confirmée** (*read committed*) : au sein de la transaction, plusieurs lectures de la même donnée peuvent livrer des valeurs différentes si une transaction concurrente l'a modifiée puis confirmée entretemps (`commit`).
- **lecture non confirmée** (*read uncommitted*) : la transaction est autorisée à lire une données qui a été modifiée mais pas encore confirmée par une transaction concurrente.

La définition de ces niveaux est critiquée pour son ambiguïté.⁸ En conséquence, les SGBD s'autorisent à les redéfinir et à les implémenter de manière spécifique, et même à proposer d'autres protocoles d'interaction. La **gestion de la concurrence**

7. En pratique, lorsque le nombre de verrous de lignes d'une table est trop grand, le SGBD procède à un *upgrade* de ces verrous, qui sont remplacés par un unique verrou assigné à la table.

8. A Critique of ANSI SQL Isolation Levels [<http://www.cs.umb.edu/~poneil/iso.pdf>]

multiversions (*multiversion concurrency control* ou *MVCC*) en est un exemple. Selon cette famille de protocoles sans verrouillage, les lectures accèdent aux données dans leur état confirmé (*committed*) le plus récent au moment du démarrage de la transaction. Durant son exécution, celle-ci ignore les états ultérieurs de ces données qui auraient été modifiées et confirmées par des transactions concurrentes. La transaction peut elle-même modifier une donnée lue. Dans ce cas, le SGBD mémorise les versions successives de cette donnée. Si, à la clôture de la transaction, le SGBD observe qu'une transaction concurrente a créé une version plus récente de la donnée, alors la transaction doit être annulée (*abort*).

Le niveau d'isolation est spécifié lors de la connexion à la base de données ou à l'ouverture de la transaction. Dans ce dernier cas, les relations avec des transactions concurrentes de niveaux d'isolation différents sont assez complexes [Gray, 1993].

5.6 PROTECTION CONTRE LES ACCIDENTS

Tous les composants d'un système d'information sont vulnérables : le disque, les processeurs, la mémoire centrale et les cartes mères des serveurs, l'alimentation, le courant électrique, l'eau de refroidissement, le système d'exploitation, le SGBD, les programmes d'application, le réseau, le navigateur, voire l'utilisateur (par maladie ou malveillance), peuvent chacun, à leur manière, présenter des dysfonctionnements ou tomber en panne. Ces accidents risquent d'affecter la disponibilité des données et leur intégrité. Il est donc nécessaire de pallier leurs effets pour garantir la robustesse de la base de données et en particulier les propriétés d'atomicité et de durabilité des transactions.

5.6.1 Sauvegardes et journaux

Les techniques de protection et de restauration en cas de panne sont basées sur des données redondantes : les copies de sauvegarde et les journaux. Une **copie de sauvegarde** (*backup*) est un instantané de la base de données pris à un instant de référence, par exemple le 1^{er} jour de chaque mois. À partir de cet instant, le SGBD démarre un **journal**. Celui-ci est un fichier historique dans lequel le SGBD consigne tous les événements affectant l'état des données. Il y inscrit en particulier les ouvertures, clôtures et annulations des transactions, ainsi que chaque modification effectuée. Ces informations sont accompagnées de l'état des données avant (*images avant*) et/ou après (*images après*) la modification. Elles sont aussi complétées par l'identification du processus client et par l'instant de l'événement. Sauvegardes et journaux sont des fichiers robustes⁹ et fortement protégés. Ils sont enregistrés sur d'autres supports que la base de données, voire sur d'autres sites. Le journal, bien que généralement très volumineux¹⁰, est parfois dupliqué. Examinons le comportement du SGBD en cas d'incident selon deux scénarios : reprise à froid et reprise à chaud..

9. Par exemple, un journal doit rester exploitable (ouvrable et lisible) après qu'un incident se soit produit, alors que le SGBD n'a pas eu le temps de le fermer proprement en recopiant sur disque les pages du tampon modifiées (relire 4.3.3).

5.6.2 Reprise à froid

Un *incident grave* a provoqué l'arrêt du système. On constate que les données sont corrompues au point d'être irrécupérables. L'exploitation est suspendue. Le SGBD restaure la dernière copie de sauvegarde. Ensuite, il parcourt le journal et réapplique dans la base de données toutes les *images après* des modifications des transactions qui se sont terminées correctement (pour lesquelles il existe un `commit_transaction`)¹¹. Lorsque le processus se termine, la base de données est dans le dernier état correct défini par les transactions qui se sont clôturées avec succès. L'exploitation peut alors reprendre.

5.6.3 Reprise à chaud

Un *incident mineur* affecte l'exécution d'une ou plusieurs transactions ou de programmes d'application mais n'entraîne pas l'arrêt de l'exploitation de la base de données. L'effet des transactions incomplètes au moment de l'incident doit être annulé. Pour ce faire, le SGBD recherche dans le journal les traces de l'ouverture de ces transactions (`begin_transaction` non suivis d'un `commit_transaction` ou d'un `abort_transaction`), puis réapplique dans la base de données les *images avant* des données modifiées. Pendant ce processus d'annulation, les autres transactions et programmes poursuivent normalement leur exécution. Les transactions et programmes arrêtés peuvent alors être relancés.

Le volume des données échangées avec le disque étant important, le SGBD profite au maximum des optimisations offertes par la gestion des tampons. Plus particulièrement, lors d'un `begin_transaction`, il procède comme suit.

- Une trace du `begin_transaction` est écrite dans le tampon du journal.

Lors d'une modification d'une donnée, le SGBD procède comme suit.

- Une *image avant* est écrite dans le tampon du journal.
- La modification est réalisée dans le tampon de la BD.
- Une *image après* est écrite dans le tampon du journal.

Lors d'un `commit_transaction`, le SGBD procède comme suit.

- Les pages de la transaction (trace du `begin` et images) dans le tampon du journal sont recopiées sur le disque.
- Une trace du `commit_transaction` est écrite dans le journal sur le disque.
- En principe les pages de la transaction dans le tampon de la base de données sont recopiées sur le disque. Cependant, une optimisation courante consiste à retarder cette recopie, le contenu du journal étant suffisant pour les reprises à froid et à chaud en cas d'incident. La recopie peut être exécutée de différentes

10. Pour des raisons d'efficacité, l'état d'une donnée avant et après modification est le plus souvent représenté par une copie de la page qui contient celle-ci. La modification d'un seul caractère dans une ligne d'une table peut entraîner l'écriture dans le journal de 2 x 4 Ko.

11. En pratique, il parcourt le journal en arrière de manière à n'appliquer que les dernières images des données modifiées.

manières : au commit de chaque transaction, après un certain nombre de transactions clôturées, lorsque le journal a atteint une certaine taille, après une période déterminée, lorsqu'un programme d'application la demande ou à la fermeture de la base de données. Cette synchronisation des contenus du journal et de la base de données s'appelle *checkpoint*.

Une gestion particulière des journaux à images *après* modification permet d'exécuter la reprise à chaud sans l'aide des images *avant*. En effet, tant que les modifications d'une transaction non clôturée n'ont pas été effectuées dans la base de données :

- en cas d'abandon de la transaction (*abort_transaction*), il suffit d'invalider les images *après* du journal;
- en outre, les processus lecteurs concurrents peuvent éviter le blocage s'ils travaillent sur les données en cours de modification; s'ils le désirent, ils peuvent accéder au dernier état stable des données en cours de modification.

Il n'est en revanche plus possible d'annuler des transactions clôturées. Cette famille de protocoles est connue sous le nom *write-ahead logging* ou *WAL*.

5.6.4 Les fichiers différentiels

La technique des fichiers différentiels [Severance, 1976] est une variante évoluée du *write-ahead logging*. Elle exploite deux principes a priori intuitifs : (1) un fichier non modifiable, en lecture seule, est peu vulnérable et facile à protéger et (2) un fichier est d'autant plus facile à protéger et à restaurer qu'il est petit.

Considérons un fichier de grande taille (**Nr** enregistrements) fréquemment mis à jour et son état **F** au temps **t0**. On lui associe un fichier annexe **D**, dit *fichier des différences*, vide en **t0**. Dès cet instant, les données sont gérées comme ceci :

- lors d'une mise à jour d'un enregistrement de **F**, la nouvelle version de celui-ci est rangée dans **D**; soit elle *remplace* la version précédente soit elle *s'y ajoute*,
- lors d'une demande de lecture d'un enregistrement, celui-ci est d'abord recherché dans **D**, puis, s'il est absent, est recherché dans **F**.

Le fichier **D** contient donc la dernière version (ou les dernières versions) des enregistrements modifiés. Il est organisé de manière que l'accès à la dernière version des enregistrements soit rapide (index primaire sur la dernière version).

Cette structure (**F**, **D**) offre d'importants avantages : (1) le fichier de base **F**, volumineux, n'est jamais modifié et peut être localisé sur un support robuste, en lecture seule, (2) le fichier des différences **D** est vulnérable mais, étant de petite taille, est facile à protéger et à restaurer, par exemple par réplication sur des supports distincts, (3) la structure physique de **F** n'étant pas perturbée par les mises à jour, ses accès sont très rapides (pages remplies à 100% par exemple), (4) en raison de la petite taille de **D**, les accès à celui-ci sont très rapides (petite taille des index par exemple), enfin (5) la journalisation des images avant et après est intégrée au fichier lui-même.

En revanche, le prix à payer est le double accès à **D** puis à **F** lors de la lecture d'un enregistrement. C'est ici qu'intervient la technique des *prédicteurs à hachage* étudiée à la section 4.16.1. En associant un prédicteur au fichier **D**, on évite dans la

plupart des cas l'accès à ce dernier lorsque l'enregistrement recherché en est absent. La figure 5.3 illustre le comportement du prédicteur en mesurant la proportion de *faux positifs* **%fp** (bit à 1 alors que l'enregistrement n'est pas présent dans **D**, ce qui exige un double accès inutile) selon la taille du prédicteur (**M** x **Nr** bits) et la proportion de modifications (**%u** = $100 \times \text{Nu} / \text{Nr}$, où **Nu** est le nombre d'enregistrements modifiés).

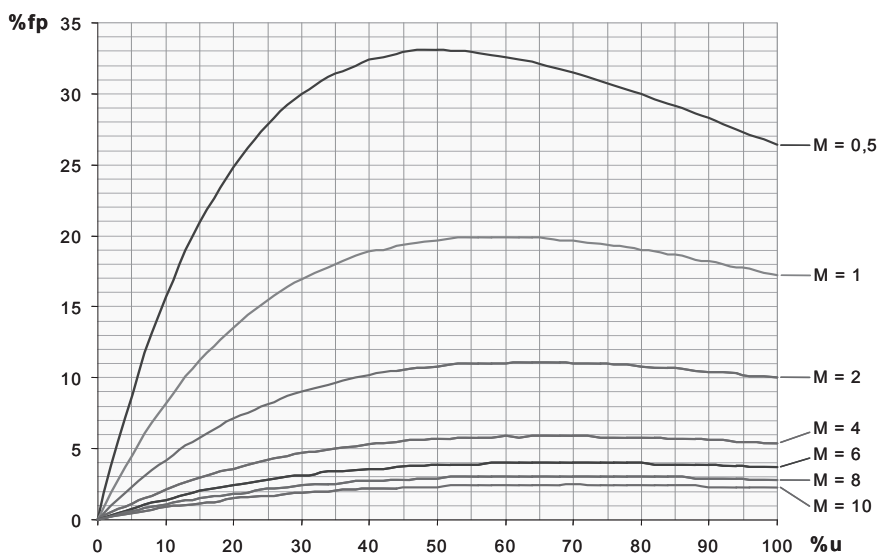


Figure 5.3 - Proportion de faux positifs dans un fichier différentiel

Pour **M** = 2, un prédicteur de 2 x **Nr** bits provoquera un double accès inutile pour 7,2% des enregistrements lorsque 0,2 x **Nr** modifications auront été effectuées.

Lorsque le fichier des différences devient trop volumineux, on met à jour le fichier **F** à partir de **D** et on repart avec une structure propre et performante.

Une description plus détaillée de cette organisation est disponible à l'annexe A5.

5.7 LES BASES DE DONNÉES DISTRIBUÉES

Nous avons jusqu'ici considéré **une** base de données gérée par **un** SGBD sur **un** serveur. On parle dans ce cas d'une **base de données centralisée**. Les systèmes d'information modernes possèdent des données distribuées géographiquement, ce qui nous conduit à de nouvelles architectures dans lesquelles un programme d'application travaille sur des données réparties géographiquement, sur plusieurs serveurs, éventuellement gérées par des SGBD différents.

5.7.1 Un premier exemple

Examinons à nouveau le cas d'un transfert d'un compte bancaire vers un autre, étudié dans la section 5.4. Imaginons que la demande émane du site de Paris, que le

compte source est géré à Poitiers et le compte destinataire à Genève, ces trois sites disposant chacun de leur propre base de données. Nous savons qu'une telle opération est encapsulée dans une transaction de manière à garantir les quatre propriétés ACID. Soit **T** cette transaction. Elle s'adresse au SGBD du site de Paris, mais celui-ci, ne disposant pas des données localement, va envoyer aux sites de Poitiers et de Genève une requête de mise à jour des comptes source et destinataires. Ces requêtes sont elles aussi encapsulées chacune dans une transaction de manière à garantir, au niveau local, les propriétés ACID. La transaction globale **T** déclenche ainsi les transactions locales **T1** et **T2** (figure 5.4). On dira que **T** est une **transaction distribuée**.

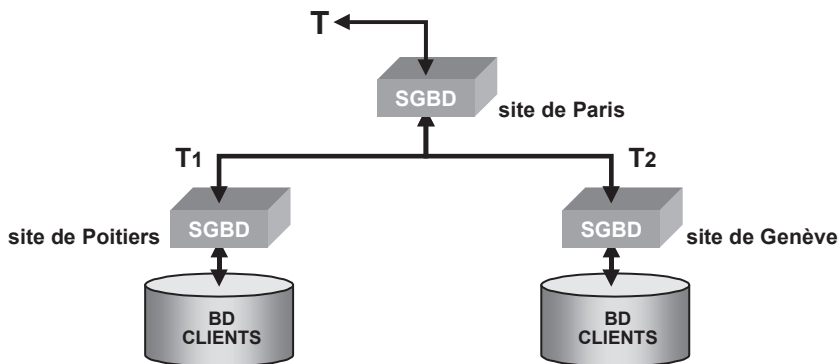


Figure 5.4 - Une base de données distribuée sur (au moins) deux sites

C'est ici que les choses se compliquent. On admet que **T** se termine avec succès, et peut donc exécuter un `commit_transaction`, lorsque **toutes** les transactions locales **T1** et **T2** se sont terminées avec succès, ayant, en théorie, exécuté un `commit_transaction`. Au contraire, **T** échoue dès qu'**une** des transactions locales échoue. Admettons que **T1** échoue et **T2** réussisse. Dans ce cas, **T** échoue, mais il est impératif que **T2** échoue également, malgré son succès, à défaut de quoi la transaction globale **T** laisserait les données de la banque incohérente.¹² La séquence standard `begin-update-commit` des transactions n'est donc plus appropriée.

Un deuxième problème risque de survenir lorsque le site de Genève, vu du site de Paris, devient inaccessible parce que les communications sont coupées ou parce que ce site est en panne. La transaction distribuée **T** ne peut se clôturer car le site de Paris est dans l'incertitude quand à l'issue de **T2**. Une solution simple serait d'attendre la reconnexion ou le redémarrage de Genève. Cependant, on va alors se heurter à un problème dérivé : le temps d'attente peut être long, de plusieurs heures dans les cas graves, temps durant lequel tous les sites intervenant dans **T** doivent maintenir le verrouillage des données impliquées.

12. À la grande satisfaction du (ou des) client(s) !

5.7.2 Les types de distribution

Analysons plus en détail le scénario ci-avant. Le monde de l'informatique bancaire n'étant pas particulièrement fantaisiste, on peut imaginer que toutes les bases de données locales obéissent au **même schéma** et qu'elles sont gérées par le **même SGBD**. La table COMPTE d'un site comprend la description des comptes des clients locaux, et d'eux seulement. Les données sont réparties sans redondance entre sites d'une manière analogue à celle de l'organisation RAID 0 entre disques (section 4.4.1). Elles constituent une base de données répartie **homogène** (même schéma et même SGBD).

Les données peuvent être **fragmentées** sans réplication¹³ comme dans notre exemple, non fragmentées et **répliquées** (analogue au RAID 1), fragmentées et répliquées (analogue au RAID 10).

La fragmentation peut être horizontale ou verticale¹⁴. Dans la **fragmentation horizontale**, tous les sites ont le même schéma mais les données sont distribuées sur une base généralement géographique, comme dans notre exemple. Dans la **fragmentation verticale** c'est le schéma (et les données correspondantes) qui est distribué entre les sites, généralement sur un critère fonctionnel, chacun se spécialisant dans un sous-système de l'entreprise. Ces deux types de fragmentation ne sont évidemment pas exclusifs. Il est nécessaire que chaque site dispose du schéma complet de la base de données distribuée, ainsi que des informations de localisation des données.

La **fragmentation** des données répond à quatre objectifs.

1. Minimiser les coûts d'accès en rapprochant les données de ses utilisateurs principaux : les données des clients de Genève sont localisées à Genève, ce qui évite ou minimise le transfert de ces données par le réseau.
2. Gérer la croissance des bases de données de grande taille de manière modulaire : il est plus facile d'ajouter un fragment que d'augmenter la taille d'une base de données centralisée.
3. Améliorer la disponibilité des données. En cas de défaillance d'un serveur, seul un fragment est indisponible et en cas de défaillance du réseau, les données locales restent disponibles pour les applications locales.
4. Paralléliser l'exécution de certaines requêtes. Nous en reparlerons ci-après à propos du pattern *MapReduce*.

En revanche, en l'absence de réplication, l'exécution de certaines requêtes peut être plus coûteuse. On pense par exemple à une jointure de deux tables situées sur deux sites. Les objectifs de la **réplication** peuvent être décrits ainsi :

1. Réduire le coût d'accès aux données non locales.

13. *Fragmentation* et *réplication* sont les termes standard dans les bases de données distribuées pour désigner la *distribution* et la *redondance*. Un *réplica* est une copie de données sources.

14. Ces qualificatifs font référence aux modes de décomposition d'une table. Une décomposition *horizontale* s'obtient par sélection, la table source se reconstruisant par une union. Une décomposition *verticale* s'obtient par projection, la table source se reconstruisant par une jointure.

2. Améliorer la disponibilité des données. En cas de défaillance du site source des données répliquées ou d'une partie du réseau, les réplicas restent disponibles.
3. Améliorer la résistance aux pannes. En cas de défaillance grave d'un site, ses données locales peuvent être reconstruites à partir des réplicas.¹⁵

Les principaux inconvénients sont l'augmentation du volume global des données et la multiplication des mises à jour dans les réplicas, entraînant quant à elle augmentation des coûts et complexité de synchronisation.

À l'opposé des bases de données **homogènes**, on trouvera des configurations **hétérogènes** dans lesquelles les données locales sont gérées par des SGBD différents, éventuellement selon des schémas distincts. Tel sera le cas d'une base de données distribuée construite à partir de bases de données locales préexistantes autonomes. Ces cas de figure sont souvent appelés **bases de données fédérées** s'il existe une gestion commune et **multibases** si la gestion reste locale.

Examinons d'un peu plus près les parties d'une base de données qui sont répliquées dans plusieurs sites. On distingue deux organisations de ces réplicas selon la manière dont les données sont modifiées. Dans une organisation **hiérarchique**, un site est déclaré **maître**, les autres étant déclarés **dépendants**. Toute requête de modification émise par un programme doit être adressée au site maître, lequel se charge ensuite, soit immédiatement (réplication **synchrone**), soit plus tard (réplication **asynchrone**) de propager les mises à jour vers tous les sites dépendants. En revanche, les requêtes de lecture peuvent être adressées à tous les sites, qu'ils soient maître ou dépendants. Cette organisation est simple et efficace tant que le site maître est actif et accessible de tous les autres sites. Si le site maître est défaillant, seule les requêtes de lectures sont permises. On peut cependant prévoir de promouvoir dynamiquement un site dépendant au statut de maître en attendant la restauration du site maître officiel. Dans l'organisation **non hiérarchique**, tous les sites ont le même statut. Une modification peut être adressée à l'un quelconque d'entre eux, lequel se charge de propager les mises à jour en mode pair-à-pair. Oracle peut gérer une organisation mixte comportant plusieurs sites maîtres communiquant en pair-à-pair.

Ajoutons enfin que la fragmentation et la distribution des données doivent être transparentes pour les programmes d'application. Les requêtes s'expriment sur le schéma global, le SGBD se chargeant de gérer transactions et communications de manière automatique. Ce n'est pas toujours le cas, certaines configurations exigeant du programme d'application de préciser la localisation des données.

5.7.3 Partitionnement du réseau

La partitionnement du réseau est un accident grave dans lequel, suite à la désactivation de certains tronçons du réseau ou à la défaillance de certains sites, le réseau est

15. Nous avons vu (section 4.4) que la reconstruction d'un disque suite à une panne peut prendre un temps considérable. Il en est de même d'une base de données multidisques. La reconstruction à partir des sites voisins se fera dès lors, non pas par transfert intégral de réplicas, mais à partir de données synthétiques extraites des journaux.

décomposé en parties disjointes. Si certains sites de la base de données distribuée se trouvent dans des parties différentes, la synchronisation entre ces sites est impossible durant un temps a priori indéterminé.

5.7.4 Le `commit` en deux phases (2PC)

La gestion des transactions doit évidemment être affinée pour tenir compte de la distribution des données. Nous décrirons les principales caractéristiques de l'algorithme de coordination le plus connu, dénommé *commit en deux phases*, ou 2PC. Nous reprenons notre exemple de transaction distribuée **T** exécutée par le site **S** et qui génère des requêtes adressées aux sites **S1** et **S2** qui les exécutent sous la forme des transactions locales **T1** et **T2**. Sur le site **S**, le gestionnaire de transaction **G** assure la coordination de l'exécution de **T**. **G** transmet à **S1** la première requête d'`update` de **T** et à **S2** la seconde. Les deux sites exécutent leur requête sous la forme des transactions locales **T1** et **T2**, mais suspendent celles-ci avant l'exécution de leur `commit_transaction`. La suite se déroule en deux phases.

1. **Phase de vote.** Chaque site envoie à **G** le diagnostic de réussite ou d'échec de ses opérations locales.
2. **Phase de commit.** Si tous les sites ont répondu positivement, **G** leur envoie un message leur ordonnant d'exécuter un `commit_transaction` local. Si au moins un site a répondu négativement, **G** envoie à tous les sites un message leur ordonnant d'exécuter un `abort_transaction` local. Dans les deux cas, chaque site retourne à **G** un message de confirmation. **G** peut alors informer le programme qui a demandé l'exécution de **T** de l'issue de cette transaction : réussite ou échec.

Si le protocole 2PC limite les cas de dysfonctionnement de transactions réparties, il ne permet pas de contrôler certaines défaillances. Par exemple, lorsqu'un site a effectué son `commit_transaction` local mais échoue à transmettre à **G** le message de confirmation, **G** est dans l'incertitude quant à la réussite globale de la transaction. Il faut parfois lever cette incertitude par une résolution manuelle. La défaillance du site du coordinateur **G** va en outre entraîner le blocage plus ou moins long de toutes les ressources verrouillées sur les sites locaux en attendant sa restauration. La mesure du temps de réponse peut dans certains cas tenir lieu de déclaration d'échec : si, dans la phase de vote, un site ne répond pas dans le délai imparti, **G** considère sa réponse comme négative. Le protocole 2PC ne peut donc pas garantir les propriétés ACID dans tous les cas. Des variantes du protocole 2PC sont disponibles notamment chez Oracle, DB2, SQL Server et MySQL Cluster.

5.7.5 Distribution des traitements

En principe, les requêtes SQL sont rédigées sans référence à la localisation des données et à l'éventuelle existence de réplicas. L'analyseur de requêtes localise les données concernées grâce à la consultation d'un dictionnaire de localisation puis décompose la requête en fragments destinés chacun à un des sites contributeurs. Cette décomposition vise à minimiser le coût d'exécution de la requête notamment en minimisant le volume de données transitant par le réseau. Par exemple, le calcul

d'une moyenne sera exécuté sur le site où résident des données brutes tandis qu'un produit relationnel sera plus efficacement exécuté sur le site d'origine (**S**) après avoir reçu les données brutes via le réseau (les données utilisées sont moins volumineuses que le résultat de la requête).

Dans bien des cas cependant, en particulier dans les SGBD non relationnels, la décomposition optimale est de la responsabilité du programme d'application. L'un des modèles de traitement fréquents est l'extraction de statistiques de grand volumes de données distribuées sur un grand nombre de sites. Ce type de traitement en apparence assez simple repose sur un modèle particulier de transaction distribuée. Il inclut donc notamment la décomposition du traitement en unités propres à chaque site, la distribution des traitements locaux, la récupération des résultats partiels, l'agrégation des résultats, l'équilibrage dynamique de la charge (un site rapide peut prendre en charge les données d'un site plus lent) et la gestion des défaillances. Google a proposé en 2004 un canevas de programmation, dénommé **MapReduce**¹⁶, permettant de spécifier de manière abstraite des traitements d'extraction de données synthétiques parallélisables sur un grand nombre de serveurs. Hadoop, de la fondation Apache, est une des implémentations les plus populaires de MapReduce.

5.8 CONTRÔLE DES ACCÈS

La base de données rassemble des données issues d'une communauté d'utilisateurs. On admet souvent qu'un utilisateur ait le droit d'effectuer toutes les opérations qu'il juge utiles sur **ses propres données** : les créer, les consulter, les modifier et les supprimer. On admet aussi qu'un utilisateur puisse être autorisé à effectuer certaines opérations sur des données dont il n'est pas propriétaire. L'enregistrement, la gestion et le contrôle strict de ces droits est une des fonctions essentielles des SGBD, dénommée **contrôle d'accès**. Une partie du langage SQL est consacrée à la création et à la modification des descriptions des utilisateurs (et des groupes d'utilisateurs) et des droits dont ils jouissent.

5.8.1 Les modèles de contrôle d'accès

Il existe trois grandes classes de modèles de contrôle d'accès : *mandataires*, *discretionnaires* et *à base de rôles* [Bertino, 1998].

- Les **modèles mandataires**¹⁷ (*mandatory access control* ou *MAC*) sembleront familiers aux amateurs de romans d'espionnage¹⁸. On associe à chaque utilisateur

16. Ce terme est issu des concepts de la programmation fonctionnelle. En Python par exemple, la fonction `map(f, L)` applique la fonction `f` à chaque élément de la liste (ou *itérable*) `L` et retourne la liste des résultats. `reduce(f, L)` applique la fonction `f` aux deux premiers éléments de la liste (ou *itérable*) `L`, puis itérativement au résultat courant et à l'élément suivant. Par exemple, `reduce(lambda x, y: x+y, L)` renvoie la somme des valeurs de `L`.

17. Traduction habituelle mais douteuse, *mandatory* signifiant *obligatoire*.

18. ... où un document est classé *public*, *confidentiel*, *secret*, *top secret*, etc.

(le *sujet*) un niveau de sécurité **Ls**. De même, chaque objet (table, ligne, colonne, etc.) est caractérisé par un niveau de sécurité **Lo**. On établit alors des règles d'accès telles que les suivantes¹⁹ :

- un sujet **s** ne peut modifier un objet **o** que si $Ls \leq Lo$ (*no write-down*),
- un sujet **s** ne peut consulter un objet **o** que si $Ls \geq Lo$ (*no read-up*).

La gestion des droits, centralisée, est très simple. On observe que le créateur d'un objet ne possède aucun droit sur celui-ci.

- Les **modèles discrétionnaires** (*discretionary access control* ou *DAC*) sont mieux connus des informaticiens. Le créateur d'un objet en est le propriétaire. Il a le droit d'effectuer toute action sur cet objet. Le propriétaire d'un objet peut concéder, à sa discrétion, certains droits à d'autres utilisateurs. Il peut également *déléguer*, c'est-à-dire autoriser un utilisateur à transmettre à d'autres certains droits qui lui ont été accordés sur cet objet. Ce modèle comporte certains raffinements tels que les suivants. *Droit négatif* ou interdiction : un utilisateur ne pourra jamais exercer un certain droit, même s'il le reçoit d'un autre utilisateur. *Droit contextualisé* : il n'est valable que pendant certaines périodes de temps (les jours ouvrables, de 8h30 à 17h) et à partir d'un certain endroit (son bureau). La plupart des SGBD ayant adopté un modèle de contrôle d'accès discrétionnaire, celui-ci sera étudié plus en détail à la section 9.2). Le modèle discrétionnaire s'avère complexe dès que le nombre d'objets et d'utilisateurs devient important.
- Les **modèles à base de rôles** (*role-based access control* ou *RBAC*). Dans ce modèle à deux niveaux, approprié aux grandes organisations, les droits sont accordés (et retirés), non pas directement à des utilisateurs, mais à des *rôles*, qui constituent des catégories remarquables d'utilisateurs : *guichetier*, *vendeur*, *consultant*, *administrateur*, *gérant de filiale*, etc. Un rôle peut en impliquer d'autres; par exemple, tout *guichetier* est un *vendeur*; les guichetiers jouissent donc aussi des droits des vendeurs. On attribue à un utilisateur un ou plusieurs rôles. Les droits de cet utilisateur sont constitués de l'union des droits des rôles qui lui ont été attribués. Certains SGBD offrent également un modèle à base de rôle (section 9.2).

5.8.2 Protection des données privées - Inférence statistique

La **protection des données privées** (*privacy*) est une extension du contrôle d'accès qui prend aujourd'hui une importance croissante, notamment dans tous les organismes gérant des données statistiques. Elle consiste à empêcher des personnes non autorisées d'obtenir des données sensibles relatives à des individus. On observe que le contrôle d'accès standard est insuffisant pour protéger ces données. Montrons-le par quelques exemples d'**inférence statistique**.

Admettons que tous les utilisateurs puissent consulter les colonnes de la table CLIENT, sauf la colonne COMPTE, dont les valeurs sont classées confidentielles. Il n'est donc pas possible d'exécuter une requête telle que :

19. On décrit ici le protocole de Bell et LaPadula. http://en.wikipedia.org/wiki/Bell-LaPadula_model [février 2009]

```
select COMPTE from CLIENT where NCLI = 'C400';
```

Admettons encore que les requêtes statistiques soient autorisées sur des données sensibles. On autorise donc la requête :

```
select sum(COMPTE) from CLIENT where LOCALITE = 'Poitiers';
```

Ceci est cependant dangereux. En effet, sachant que C400 est le seul client de Poitiers de catégorie B2, je pourrais demander innocemment :

```
select sum(COMPTE) from CLIENT
where LOCALITE = 'Poitiers' and CAT = 'B2';
```

pour obtenir en toute légalité l'information confidentielle désirée. On décide alors que le SGBD n'accepte d'exécuter une requête statistique que si elle porte sur un échantillon assez grand mais pas trop grand, par exemple entre 25% et 75% de la population totale. Ce n'est malheureusement pas suffisant. Il existe un procédé, laborieux certes mais efficace, permettant de contourner cette protection. Choisissons une condition d'apparence anodine, qu'on appelle un **traceur** (*generalized tracker*), satisfaite par près de la moitié de la population. La condition $NOM < 'M'$ convient très bien. Calculons respectivement les grandeurs **S1** et **S2** comme suit :

```
select sum(COMPTE) into :S1 from CLIENT where NOM < 'M';
select sum(COMPTE) into :S2 from CLIENT where NOM >= 'M';
```

S = S1 + S2 nous donne la somme des comptes de tous les clients, information pourtant confidentielle (calculée sur plus de 75% de la population). Continuons :

```
select sum(COMPTE) into :C1 from CLIENT
where (NOM < 'M') or (LOCALITE = 'Poitiers' and CAT = 'B2');
select sum(COMPTE) into :C2 from CLIENT
where (NOM >= 'M') or (LOCALITE = 'Poitiers' and CAT = 'B2');
```

La grandeur **C = C1 + C2 - S** est exactement la valeur du compte du client C400, soit 350 ! Notre seul mérite est de connaître une combinaison de conditions identifiant notre victime²⁰ et, bien sûr, d'avoir imaginé un traceur.

La mauvaise nouvelle est qu'on sait de longue date²¹ qu'il n'existe pas de procédés généraux de protection véritablement satisfaisants contre ce genre d'attaque et qu'en particulier l'anonymisation des données (substitution d'identifiants arbitraires aléatoires aux véritables identifiants) est inutile. Mentionnons trois techniques de protection.

- Les requêtes statistiques sont évaluées non pas sur l'ensemble des données individuelles mais sur un échantillon aléatoire extrait soit dynamiquement au moment de l'exécution des requêtes, soit anticipativement et enregistré dans une base de données dérivée.
- Les requêtes statistiques portent sur une base de données dérivée, construites à partir de la base de données principale, mais dans laquelle on a introduit des perturbations aléatoires de moyenne nulle dans les valeurs des colonnes sensibles,

20. Tester le caractère identifiant de cette combinaison est aisé, par le même procédé.

21. Voir par exemple [Denning, 1979] pour plus d'information.

de sorte que les données individuelles soient très imprécises mais que les statistiques soient d'autant plus exactes que l'échantillon sur lequel elles portent est important.

- Dans cette base de données dérivée, on permute aléatoirement les données sensibles entre les enregistrements (on permute les valeurs de COMPTE entre les lignes NUMCLI = 'C400' et NUMCLI = 'B512') mais de manière à préserver les statistiques selon certains critères. Par exemple, des permutations de COMPTE effectuées au sein de chaque classe de lignes de valeurs de LOCALITE et CAT déterminées préserveront les statistiques sur ces deux critères mais pas selon tous les autres.

Le danger n'est d'ailleurs pas lié à la présence de données individuelles mais existe aussi pour des données statistiques précalculées, comme en témoignent les problèmes mis en évidence lors des recensements américains aux siècles derniers. Jusqu'en 1840, les résultats statistiques étaient rendus publics. Observant que certaines informations individuelles pouvaient se déduire de ceux-ci, cette politique a subi des restrictions de plus en plus fortes²². En 1920, le *Census Act* impose la confidentialité des statistiques pouvant conduire à des informations sur des individus identifiables. On imagine aisément l'importance de cette question aujourd'hui face à la collecte non contrôlée de données individuelles, parfois prétendument anonymisées, par les réseaux sociaux et les entreprises de distribution.

5.9 LES BASES DE DONNÉES *IN-MEMORY*

L'application ultime de la technique des tampons (section 4.3.3) consiste à charger la totalité de la base de données en mémoire centrale. Cette base de données est dite *in-memory*.²³ Les accès à la mémoire externe étant tous traduits par des transferts internes à la mémoire centrale, la lecture et l'écriture des données sont par conséquent plus rapides que si les données résidaient sur disque. Cette architecture n'est pas neuve. Il existait en effet, dans les années soixante-dix, une variante du SGBD IMS d'IBM dénommée *IMS Fast Path*.²⁴ Aujourd'hui, en raison de l'augmentation de la taille des mémoires centrales, plusieurs SGBD (Oracle 12c, Hekaton de SQL Server, DB2 BLU d'IBM et SQLite par exemple²⁵) offrent une version *in-memory* permettant de gérer des bases de données de grande taille (plusieurs To). Leur usage va des bases de données à haut débit transactionnel (OLTP) jusqu'aux entrepôts de données destinés à l'extraction de statistiques (OLAP).²⁶ On citera VoltDB et

22. À tel point que certains chercheurs se plaignent des erreurs que ces mesures de protection provoquent dans leurs résultats. Voir par exemple l'article du Wall Street Journal du 6 février 2010, *Census Bureau Obscured Personal Data - Too Well, Some say*. Le Census Bureau américain utilise les trois techniques décrites.

23. Il faut reconnaître que la traduction française *base de données en mémoire* n'aurait aucune allure !

24. D'ailleurs toujours disponible sous z/OS.

25. Voir http://en.wikipedia.org/wiki/List_of_in-memory_databases (juin 2015)

26. Les excellentes performances de certains SGBD du type NoSQL (chapitre 10) sont dues à l'usage de bases de données *in-memory*. Tel est le cas de Redis par exemple.

Apache Ignite parmi les nombreux exemples de SGBD spécialisés dans cette approche.

L'utilisation du tampon pour y charger la base de données ne serait en fait qu'un palliatif imparfait. Par exemple, la répartition des données dans une suite de pages et les techniques de stockage des enregistrements dans ces pages sont dues à la géométrie hétérogène des disques. La mémoire centrale étant linéaire (une simple suite d'octets, de temps d'accès uniforme), ces structures complexes ne sont plus nécessaires, de sorte que leur gestion par le SGBD peut se simplifier considérablement.

Si l'exploitation d'une base de données en mémoire est très rapide, le chargement initial des données, et, si des modifications ont été effectuées, la recopie de ces données sur disque peuvent être très coûteux. Par exemple, le **chargement en streaming** du contenu de notre disque de référence de **500 Go** prendra plus de **2 heures 30 minutes**.²⁷

5.10 LE MARCHÉ DES SGBD RELATIONNELS

Le marché des SGBD commerciaux est actuellement dominé par *Oracle*, *DB2* d'IBM, *SQL Server* de Microsoft et la version commerciale de *MySQL*. Les deux premiers peuvent être considérés comme *historiques* (introduits respectivement en 1979 et 1982) alors que le troisième dérive de Sybase, lequel est aujourd'hui diffusé sous le nom de *Adaptive Server Enterprise* (ASE).

Le monde Open Source propose également des SGBD relationnels de bonne qualité. On citera en particulier Firebird, ou son grand frère InterBase 6, PostgreSQL et MariaDB²⁸. Ce dernier, malgré sa popularité, notamment auprès des développeurs d'applications web, n'a pas atteint le niveau de maturité de ses concurrents²⁹. On notera encore SQLite, qui jouit d'une position très particulière. Très compact et efficace, il adopte une architecture rudimentaire (le SGBD et programme d'application sont intégrés) et non pas l'architecture client-serveur de la section 5.3. SQLite est probablement le SGBD le plus répandu à l'heure actuelle.

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. L'interface SQL d'Access est cependant délicate à

27. Pour un disque plus moderne tel que WD Ultrastar 7K2000 de 2 To et 7 200 rpm, comportant 291 000 cylindres de 7 pistes, le temps est de 16 968 s. de transfert + 291 s. de déplacement unitaires du bras, soit au total 4 heures 45 minutes. La répartition des données sur *n* disques, ensuite relus en parallèle, divise ce temps par *n*.

28. MySQL, initialement Open Source, a été acquis par SUN, lequel a été ensuite absorbé par Oracle. MySQL étant devenu un produit commercial, ses concepteurs ont développé une nouvelle branche, MariaDB.

29. Il n'y a pas si longtemps (version 3), la documentation de MySQL justifiait l'absence de clé étrangère par cet argument surprenant : *il suffit de faire attention à ne pas introduire de données erronées*. On frémit à l'idée qu'un tel raisonnement s'applique un jour au domaine médical et à la sécurité routière et aérienne !

utiliser, notamment par son caractère non standard, ses lacunes et la difficulté de trouver une documentation précise sur ce langage.

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 ISO. La première norme a été publiée en 1986 (SQL-86). La suivante, SQL-89, introduit notamment l'intégrité référentielle. 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. 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 au chapitre 9 quelques-unes des principales extensions qu'apporte la norme 1999 par rapport à celle de 92. Malheureusement, les éditeurs de SGBD montrent peu d'empressement à respecter l'intégralité des 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.

Les bases de données NoSQL

Cette famille récente de SGBD sera étudiée au chapitre 10.

5.11 POUR EN SAVOIR PLUS

Du point de vue historique, une étude particulièrement fouillée des premiers âges (1963 aux années quatre-vingts) a été publiée sous la référence [Haigh, 2006], aussi disponible à l'adresse (<http://tomandmaria.com/tom/Writing/VeritableBucketOfFacts-SIGMOD.pdf>). De T. Haigh toujours, le rapport des entretiens avec Ch. Bachman, pionnier des SGBD, préliminaire à l'étude mentionnée précédemment³⁰. Une chronologie intéressante de 1969 à 2008 est disponible à l'adresse <http://fadace.developpez.com/sbgdcmp/story/>.

On trouvera une étude très détaillée des fonctions des SGBD dans [Gray, 1993] et [Silberschatz, 2011] pour ce qui est des transactions mono et multisites, et dans [Garcia-Molina, 2008] pour l'ensemble des fonctions. [Elmasri, 2006] couvre également le sujet mais de manière plus généraliste.

L'historique et l'évolution du langage SQL et de ses normes sont proposés dans [Melton, 2002]. Les normes SQL sont disponibles 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>).

30. Charles W. Bachman interview: September 25-26, 2004 ; Tucson, Arizona, (<http://portal.acm.org/citation.cfm?id=1141880.1141882>).

Chapitre 6

Le langage SQL DDL

Ce chapitre ainsi que les deux suivants présentent, selon une approche progressive et raisonnée, le langage de base de données SQL (*Structured Query Language*), et en particulier les deux parties importantes que sont les sous-langages DDL (*Data Definition Language*) et DML (*Data Manipulation Language*). On examine dans ce chapitre les fonctions de création et modification des structures de données qui ont été étudiées dans le chapitre 2, et la manière dont ces fonctions sont mises en œuvre dans le langage SQL DDL.

6.1 INTRODUCTION

Nous nous servons dans ce chapitre et les deux suivants de la norme SQL2 du langage, largement disponibles dans la plupart des SGBD. Les extensions offertes par la norme SQL3 (ou SQL:1999) seront étudiées au chapitre 9. Le lecteur trouvera sur le site de l'ouvrage la syntaxe précise du langage SQL2.

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 constitue une **requête**¹, 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

1. Il est à noter cependant que, dans certains ouvrages, le terme de *requête* est limité à l'instruction d'extraction de données *select-from-where*.

apparaît à l'écran. Cette requête peut également être envoyée par un programme (écrit en Pascal, C, COBOL, Basic, PHP 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.

Comme déjà mentionné à la section 2.15, il est vivement recommandé de travailler les matériaux de cet ouvrage en les expérimentant au moyen d'un SGBD. Si le lecteur n'a pas accès à un moteur *lourd* tel qu'Oracle ou DB2 (versions *Education* gratuites), on pourra recommander l'interface SQL d'Access, InterBase, livré avec Delphi de Borland ou Firebird, (Open Source) ou SQLite. Le site web de l'ouvrage contient des exemples rédigés en SQLite, via l'interface SQLfast.

6.2 LE LANGAGE SQL DDL

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

6.3 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

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 (la syntaxe dépend du SGBD et du langage de programmation), ce qui ouvre une session de travail :

```
connect to ADMIN\SERV01\GESCOM.db as CON_21Dec2015_002  
user 'jlh' password 'jlhclicom';
```

À la fin du travail, on demande la déconnexion, ce qui clôt la session :

```
disconnect CON_21Dec2015_002;
```

Durant une session, l'utilisateur peut exécuter des instructions SQL de création de structures et de manipulation de données. Nous examinerons de manière plus concrète au chapitre 9 les détails de l'interface entre Java et les bases de données.

6.4 LES NOMS SQL

La formation des noms des tables, des colonnes et des autres objets est soumise à des règles très strictes. Tout nom *régulier* est une chaîne formée de lettres (sans accent), de chiffres et de signes #, \$ et _. Un nom ne peut commencer par un chiffre. Donc pas d'espaces, de tirets ni de lettres accentuées. Les noms du langage SQL sont aussi interdits (table, integer, create, index, etc.) En revanche, les lettres majuscules et minuscules sont équivalentes : TAUX et taux sont considérés comme des noms identiques. Il est cependant possible de contourner ces contraintes en entourant un nom *irrégulier* de guillemets (" ") ou de crochets ([]). Dans ce cas, "TAUX" et "taux" ne sont pas équivalents. Par exemple, les noms suivants sont valides:

CLIENT, NUM_TVA, #CLI, Prix, PopulationTotale, "Prix à l'unité", [Dérivée], "index"

6.5 CRÉATION D'UNE TABLE

Cette opération produit une table (vide, c'est-à-dire sans lignes) dont le schéma est conforme aux indications données par la requête `create table`. On y spécifie le nom de la table et la description de ses colonnes. On spécifiera en outre 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) ) ;
```

Pour prendre en compte le fait que la table existe déjà, on utilisera les formes `create or replace table CLIENT` ou `create table if not exists CLIENT`.

Si une table n'est utile que durant la session courante, elle sera déclarée temporaire. Elle disparaît lors de la déconnexion de la base de données :

```
create temporary table RESULTAT ( ) ;
```

6.6 LES COLONNES ET LEURS TYPES

SQL offre divers types de données, dits *de base*, dans la déclaration d'une colonne d'une table. On citera les principaux :

- `SMALLINT` : entiers signés courts (par ex. 16 bits) ;

- **INTEGER** (ou **INT**) : entiers signés longs (par 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** ou **REAL**) : 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 1000^e de seconde) ;
- **TIMESTAMP** : date + temps ;
- **INTERVAL** : intervalle en années/mois/jours entre dates ou en heures/minutes/secondes entre instants².

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 séquences 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, tel le type *sequence* d'Oracle, les attributs *autoincrement* de SQLite et MS Access, *auto_increment* de MySQL et *identity* de SQL Server.

En résumé, les types SQL de base se classent selon le tableau de la figure 6.1. 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 modifier³.

```
create domain MONTANT decimal(9,2);
```

2. 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.

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

```
create domain MATRICULE char(10);
create domain LIBELLE char(32);
```

numérique	exact	smallint, integer, decimal, numeric
	approché	real, float, double
chaîne de	bits.....longueur fixe ...	bit
	longueur variable	bit varying
	longueur indéfinie	BLOB
caractères.....	longueur fixe ...	char
	longueur variable	character varying
	longueur indéfinie	CLOB
temporel	date.....	date
	instant	time
	instant daté	timestamp
	intervalle	interval
booléen		boolean

Figure 6.1 - Classification des types SQL de base

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

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',
. . . DATECOM date not null default current_date4,
```

SQL3 introduit le concept plus puissant de *type* (section 9.6.2).

6.7 CARACTÈRE OBLIGATOIRE/FACULTATIF D'UNE COLONNE

Par défaut (c'est-à-dire si on ne spécifie rien) toute colonne est facultative. Le caractère obligatoire d'une colonne se déclarera par la clause *not null* :

```
create table OFFRE ( NUMFL char(10) not null,
                   NUMPL char(15) not null,
                   PRIX decimal(8),
```

4. La date courante au moment de l'insertion d'une ligne; voir 7.3.2.

```
primary key (NUMFL, NUMPL),
foreign key (NUMFL) references FOURNISSEUR,
foreign key (NUMPL) references PIECE );
```

6.8 LES IDENTIFIANTS

a) 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) );
```

b) Identifiant secondaire

Les autres identifiants seront déclarés via le prédicat unique :

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

6.9 LES CLÉS ÉTRANGÈRES

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

```
create table DETAIL ( NCOM  char(12),
                     NPRO  char(15),
                     QCOM  decimal(8),
                     primary key (NCOM,NPRO),
                     foreign key (NCOM) references COMMANDE,
                     foreign key (NPRO) references PRODUIT );
```

Par défaut, l'identifiant *visé* par la clé étrangère dans la table cible est l'identifiant primaire de celle-ci. Il est possible, mais ceci est déconseillé en toute généralité, de définir une clé étrangère *visant* un identifiant secondaire de la table cible. Dans ce cas, les composants de cet identifiant doivent être mentionnés dans la déclaration :

```
foreign key (PROPRIO) references PERSONNE(NUM_REGISTRE)
```

a) Comportement lors des mises à jour des données

La description d'une clé étrangère doit également spécifier le type de comportement choisi lors des mises à jour, comme discuté à la section 2.10.3. 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 suit⁵ :

```
create table COMMANDE ( NCOM char(12) not null,
                        NCLI char(10) not null,
                        ...,
                        foreign key (NCLI) references CLIENT
                                on delete no action);
```

- **propagation** : la suppression est effectuée, 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,
                        ...,
                        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 `NCLI`⁶. Ce comportement est typique des clés étrangères cycliques telles que celle de la figure 8.1. Il est spécifié comme suit :

```
create table COMMANDE ( NCOM char(12) not null,
                        NCLI char(10),
                        ...,
                        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 : 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,
                        ...,
                        foreign key (NCLI) references CLIENT
```

5. La norme spécifie une variante, disponible dans certains SGBD : le mode **restrict**. La différence est très subtile et en général mal comprise. Le mode **restrict** rejette l'`update` si, **avant** exécution, il existe des lignes dépendant de la ligne modifiée. Le mode **no action** rejette (annule) l'`update` si, **après** exécution, des lignes dépendantes sont devenues orphelines. Les situations dans lesquelles ces comportements diffèrent sont rares.

6. Il existe une variante **set default**, par laquelle la valeur par défaut déclarée pour cette colonne se substitue à la valeur `null`.

```
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);

create table COMMANDE ( NCOM char(12) not null primary key,
                        NCLI char(10) not null,
                        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.

c) Les clés étrangères partiellement facultatives

La clé étrangère comporte autant de colonnes que l'identifiant cible et ces colonnes sont de même type, prises deux à deux. Si au moins une des colonnes d'une clé étrangère est facultative, alors il est possible de préciser, via la clause **match**, la manière dont l'intégrité référentielle sera vérifiée en présence de valeur(s) null.

- **match simple** : si **toutes** les colonnes de la clé étrangère possèdent une valeur, la contrainte référentielle est évaluée, sinon elle est ignorée.
- **match full** : si les colonnes sont toutes null, la contrainte est ignorée. Si elles sont toutes non null, elle est évaluée. Dans les autres cas, elle n'est pas satisfaite.
- **match partial** : la contrainte est évaluée pour les colonnes non null. La table cible doit contenir au moins une ligne dont l'identifiant comporte les valeurs non null de la clé étrangère.

Exemple :

```
foreign key (NCLI, NFOURN) references ACHAT (NCLI, NFOURN)
on delete cascade
match full,
```


6.10 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 être 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);
```

6.11 EXEMPLE COMPLET

On peut ainsi donner la définition du schéma de la base de données décrite à la figure 2.7 sous la forme d'une suite de requêtes (ou *script*) SQL :

```
create table CLIENT (  NCLI      char(10) not null primary key,
                      NOM       char(32) not null,
                      ADRESSE   char(60) not null,
                      LOCALITE  char(30) not null,
                      CAT       char(2),
                      COMPTE    decimal(9,2) not null);

create table PRODUIT ( NPRO      char(15) not null primary key,
                      LIBELLE   char(60) not null,
                      PRIX      decimal(6) not null,
                      QSTOCK    decimal(8) not null);

create table COMMANDE (NCOM      char(12) not null primary key,
                      NCLI      char(10) not null,
                      DATECOM   date not null,
                      foreign key (NCLI) references CLIENT
                      on delete no action on update cascade);

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
                      on delete cascade on update cascade,
                      foreign key (NPRO) references PRODUIT
                      on delete no action on update cascade);
```

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.

6.12 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 2.10 sur l'impact des contraintes d'intégrité sur les opérations de modification des données. Pour éviter les erreurs si la table n'existait pas, on écrira par prudence :

```
drop table if exists DETAIL;
```

6.13 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 une caractéristique d'une colonne :

```
alter table CLIENT alter column CAT set default '00';  
alter table CLIENT alter column CAT drop default;
```

Un domaine peut être modifié et supprimé lorsqu'il n'est plus utilisé :

```
alter domain MONTANT set default -1.0;  
drop domain MATRICULE;
```

6.14 AJOUT ET RETRAIT D'UNE CONTRAINTES

Nous avons rencontré jusqu'ici quatre types de contraintes : les identifiants primaires et secondaires, les colonnes obligatoires et les clés étrangères. Ces contraintes sont généralement déclarées lors de la création de la table qui y est soumise. Il est cependant possible de les ajouter et de les retirer 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);
alter table CLIENT add unique (NOM, ADRESSE, LOCALITE);
```

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

```
alter table CLIENT alter CAT not null;
alter table CLIENT alter ADRESSE null;
```

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 nommée :

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

6.15 LES STRUCTURES PHYSIQUES

Les structures physiques décrites au chapitre 4 sont créées 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, si l'index est trié, l'ordre de rangement des valeurs dans l'index (`asc` = ascendant, `desc` = descendant). L'ordre par défaut est *ascendant*. Si la colonne de l'index est facultative, la position des valeurs null peut être spécifiée par la clause `nulls first` ou `nulls last`.

```
create index XCLI_LOC 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 XDET_NCOM_NPRO
on DETAIL (NCOM asc, NPRO desc);
```

Cette forme a été utilisée dans les premières versions des SGBD relationnels pour définir des identifiants. Elle est aujourd'hui considérée comme obsolète.⁷

Il existe des variantes d'index disponibles dans la plupart des SGBD : les index *partiels*, réduit à un sous-ensemble de lignes de la table (section 4.10.4), les index *sur expression* (section 4.10.5) et les index *augmentés* (section 4.10.6). Les index partiels permettent en particulier de pallier l'absence d'identifiant facultatifs dans certains SGBD, tels que SQL Server (voir annexe A06).

Il est en outre possible de supprimer un index considéré comme inutile,

```
drop index XCLI_LOC;
```

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

```
create dbspace CLI_PRO_DAT;
```

On spécifiera ensuite pour chaque table l'espace de stockage dans lequel ses lignes doivent être stockées :

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

6.16 QUE RETENIR ?

Le sous-langage DDL de SQL permet de créer et modifier les structures d'une base de données. Il est ainsi possible de créer, supprimer et modifier un domaine, une table, une contrainte, un index et un espace de stockage.

6.17 POUR EN SAVOIR PLUS

Le langage décrit est celui de la norme SQL2. La version suivante, SQL3 (ou SQL:1999), introduira nombre de concepts supplémentaires qui seront présentés au chapitre 9.

L'annexe A6 complète certains aspects de ce chapitre. En particulier, il décrit la création en SQLfast de la base de données de référence de la figure 2.8.

7. Les SGBD créent automatiquement un index pour chaque identifiant primaire (et parfois secondaire). La raison est compréhensible : il serait absurde d'imposer une contrainte d'unicité dont la validation serait exagérément coûteuse (voir chapitre 4).

Chapitre 7

Le langage SQL DML (1)

Ce chapitre, ainsi que le suivant, décrivent la deuxième partie importante du langage SQL : le sous-langage DML (*Data Manipulation Language*). On y examine les principes de l'extraction de données d'une table. Outre les formes simples de la requête d'extraction `select`, on étudie les fonctions SQL, les fonctions agrégatives, les sous-requêtes et les quantificateurs ensemblistes.

7.1 INTRODUCTION

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

L'extraction¹ 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, même partielle, exigera non seulement la totalité de ce chapitre, mais aussi la plus grande partie du chapitre 8 et plusieurs sections du chapitre 9. Bien que la connaissance du langage SQL soit l'un des objectifs de ces chapitres, nous voudrions surtout insister sur l'expression rigoureuse de raisonnements corrects, ce que le langage courant est souvent loin de permettre. Dans la suite de cet exposé, nous aborderons principalement :

1. Le terme *extraction* est à prendre dans une acception qui diffère de celle qui est utilisée en dentisterie. Une donnée extraite reste dans la base de données. On en extrait en réalité une copie. Pour extraire une donnée au sens propre, on utilisera la commande *delete*.

1. l'extraction de données d'une seule table, indépendamment des autres tables (sections 7.2, 7.3, 7.5) ;
2. l'extraction de données d'une seule table, mais en corrélation avec les données d'une autre table via les sous-requêtes (section 7.6).

Les nombreuses variantes de l'instruction **select** de SQL permettent d'extraire des données d'une table ou de plusieurs 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 dans ce chapitre suppose une utilisation interactive : l'utilisateur introduit la requête au terminal et en demande l'exécution au SGBD qui affiche les données extraites à l'écran².

Nous venons de préciser que l'exécution d'une requête **select** produit un résultat qui est une table. Ce fait est important, car il implique naturellement qu'on puisse remplacer, dans une requête, le nom d'une table par une requête **select**. Nous y reviendrons au chapitre 9.

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 la table ou les tables desquelles le résultat tire ses valeurs,
- la clause **where** spécifie 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. Nous serons assez rapidement mis en présence de plusieurs types de tables, qui obéissent à des règles différentes. Nous en retenons quatre.

1. **Les tables de base.** Celles-ci sont stockées de manière permanente dans une mémoire externe et sont accessibles à tous les utilisateurs, moyennant le respect des règles d'accès. La figure 2.8 représente une base de données comportant quatre tables de base.
2. **Les résultats de requêtes SFW.** La table résultant de l'exécution d'une requête d'extraction est volatile. Ses lignes sont soit envoyées à l'écran, soit utilisée comme sous-requête dans une autre requête, soit livrées une par une au programme qui en a demandé l'extraction, puis elle disparaissent.
3. **Les tables dérivées.** Il est possible de stocker le résultat d'une requête dans une table définie à cet effet (requête **insert into**, étudiée au chapitre 8). En général, de telles tables sont temporaires et leur usage est limité à un utilisateur pendant une durée relativement courte (quelques jours).

2. Contrairement aux apparences, ce mode d'interaction obéit à l'architecture client-serveur décrite à la section 5.3. En effet, l'utilisateur ne dialogue pas directement avec le SGBD mais avec un petit programme très élémentaire, généralement écrit en Java, dont le rôle consiste à saisir une requête que l'utilisateur écrit dans un champ, à la transmettre au SGBD, à recevoir les lignes du résultat et à les afficher sous une forme tabulaire à l'écran.

Elles sont parfois appelées *instantané* ou *snapshot*.

4. **Les vues.** Une vue est une table virtuelle dont le contenu est défini comme le résultat d'une requête SFW qui lui est attachée. Seule sa définition est stockée, son contenu étant recalculé lors de chaque consultation. Les vues seront étudiées à la section 9.3.

La manipulation expérimentale des fonctions étudiées dans ce chapitre est essentielle. Le lecteur trouvera des indications précieuses sur l'installation et l'utilisation de SGBD relationnels dans la section 2.15 et sur le site de l'ouvrage. Sauf mention contraire, tous les exemples de requêtes et leurs résultats sont relatifs à la base de données illustrée à la figure 2.8. Les expressions SFW seront introduites dans ce chapitre de manière progressive, par ordre de complexité croissante.

7.2 REQUÊTES ÉLÉMENTAIRES

Cette section introduit des formes simples de la requête SFW qui livrent des valeurs de colonnes extraites de lignes sélectionnées par des conditions élémentaires.

7.2.1 Extraction simple

La requête la plus simple consiste à demander l'affichage de valeurs de certaines colonnes des 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 suit³ :

```
select *
from   CLIENT;
```

7.2.2 Extraction de lignes sélectionnées

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

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

Le résultat est la table suivante :

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, l'expression `ch1 < ch2`, pour toutes chaînes `ch1` et `ch2`, s'interprète comme : *ch1 est avant ch2* selon l'ordre lexicographique (celui du dictionnaire). Attention, selon cet ordre, le caractère "a" n'est pas égal à "A". Plus précisément, "Z" < "a" et "12" < "2". 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 ;

3. Cette forme synthétique est déconseillée pour une requête apparaissant dans un programme d'application (ou une procédure SQL, un déclencheur ou dans la définition d'une vue). 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.

- 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'); certains SGBD admettent les formes "rue de l'Eté" ou 'rue de l\' 'Eté' ;
- dates : '2016-02-14' (standard SQL2⁴) ou '2015/02/14'.

Petite remarque en passant : la décomposition d'une requête en deux ou trois lignes la rend plus lisible mais n'est pas obligatoire. On peut en effet écrire :

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

7.2.3 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, si 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
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 éliminera les lignes en double par la clause `distinct`⁵ :

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

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

4. En fait, le standard préconise la forme `date '2016-02-14'` ; pour des raisons de concision, nous adopterons la forme courte, souvent tolérée par les SGBD.

5. L'expression "`select LOCALITE ...`" est une forme abrégée de "`select all LOCALITE ...`".

LOCALITE

Namur
Poitiers

Ainsi qu'on l'a préconisé dans le chapitre 2, nous travaillerons sur des tables de base 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;
```

L'existence de lignes en double dans une table peut poser des problèmes 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 d'un identifiant* de la table, l'unicité des lignes du résultat est garantie. Il est donc inutile⁶ d'inclure `distinct`.

7.2.4 Des conditions de sélection plus complexes

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

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

Pourquoi les formes spéciales (`is`, `is not`) pour exprimer l'égalité ou la non égalité dans le cas des valeurs `null` et non les formes classiques `CAT = null` et `CAT <> null` ? Nous examinerons plus en détail la question de l'information incomplète aux sections 7.6 et 9.12, et surtout l'annexe A24. Nous y verrons que ces deux conditions, quelle que soit la valeur de `CAT`, ont toujours la valeur `unknown`⁷. En effet, la question *est-ce que "ce que je ne connais pas" est égal à "ce que je ne connais pas"* ne peut avoir d'autre réponse que *"je ne sais pas"*, traduit par `unknown` en SQL.

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

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

7. Sauf, hélas, en Access, où `CAT = null` se comporte comme `CAT is null`. Cette confusion n'existe pas dans les autres SGBD.

```
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. Un masque est une chaîne de caractères agissant comme un filtre. Le signe "_" y 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 toute valeur de ADRESSE contenant le mot "Neuve", c'est-à-dire constituée d'une suite quelconque (%) suivie de "Neuve" suivie d'une suite quelconque (%).

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 suivant, 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 2015⁸

```
DATECOM like '2015%'
```

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

8. Cette forme est souvent acceptée mais le standard SQL recommande les formes complètes *cast (DATECOM as char) like '2015%'* ou *extract (year from DATECOM) = 2015*.

'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)`. Voir aussi la fonction `trim` ci-après.

Une condition peut enfin porter sur un n-uplet (liste de composants) :

```
(LOCALITE, CAT) = ('Genève', 'B1')
```

7.2.5 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;
```

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

- **where P and Q** sélectionne les lignes qui vérifient simultanément P et Q ;
- **where P or Q** sélectionne les lignes qui vérifient P ou Q ou les deux ;
- **where not P** sélectionne les lignes 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)
```

SQL n'offre pas les trois opérateurs complémentaires classiques *ou exclusif* ($\oplus$), *implication* ($\Rightarrow$) et *équivalence* ($\Leftrightarrow$). Ceux-ci devront donc être simulés à l'aide des trois opérateurs de base :

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

Les principes décrits dans cette section supposent que les conditions prennent l'une des valeurs {true, false}. Or, dans certaines circonstances, une condition SQL peut prendre une des trois valeurs {true, false, unknown}, relevant ainsi de la logique

ternaire. Celle-ci sera évoquée à la section 9.12.3 et étudiée de manière plus détaillée dans l'annexe A24.

7.3 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

Dans le 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 assignera alors explicitement un nom à cette colonne, ce qu'on appelle un **alias de colonne**, comme dans l'exemple suivant :

```
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

7.3.1 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. La bibliothèque des fonctions SQL peut en outre être enrichie de nouvelles fonctions développées par les utilisateurs, sous la forme de *User Defined Functions* (UDF). Des expressions faisant usage de fonctions peuvent apparaître dans la clause `select` et dans la clause `where`⁹, mais aussi, comme nous le verrons plus tard, dans les instructions `update` et `insert`. Nous citerons les principales¹⁰.

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 absolue.

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* ;
- `ch1 || ch2` : construit une chaîne composée de la concaténation (*mise bout-à-bout*) des chaînes *ch1* et *ch2* ;
- `lower(ch)` et `upper(ch)` : construit une chaîne formée des caractères de *ch* transformés respectivement en minuscules et majuscules ;
- `substring(ch from I for L)` : construit une chaîne formée des *L* caractères de la chaîne *ch* partant de la position *I* ;
- `trim(ec 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(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 chaîne de bits *ch*.

d) Fonctions de conversion

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

Exemples

```
cast(ATECOM as char(12))
```

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

10. 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.

```
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.

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 exp1`
`when c2 then exp2`
`...`
`else expn`
`end`
renvoie la valeur de l'expression *exp1* si la condition *c1* est vraie, *exp2* si *c2* est vraie, ..., sinon renvoie *expn*, ou *null* si cette dernière clause est absente;
- `case exp`
`when v1 then exp1`
`when v2 then exp2`
`...`
`else expn`
`end`
variante de la précédente pour des conditions de la forme *exp = vi* ;
- `coalesce(exp1, exp2, ..., expn)` : renvoie la première expression différente de *null*, soit *null* s'il n'y en a pas ;
- `nullif(exp1, exp2)` : renvoie *exp1* si celle-ci est non *null*, *exp2* sinon.

Exemple

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

7.3.2 Les 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 le premier du mois :

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

7.4 ORDRE DES LIGNES DU RÉSULTAT

Par construction, l'ordre des lignes d'une table ou d'un résultat est arbitraire. Il est possible cependant d'imposer un ordre défini des lignes du résultat d'une requête via la clause **order by**, comme illustré par la requête suivante :

```
select NCLI,NOM,LOCALITE
from   CLIENT
order by LOCALITE,NCLI;
```

Les clients apparaissent par valeurs croissantes de `LOCALITE`, et au sein de chaque groupe de même valeur de `LOCALITE`, par valeurs croissantes de `NCLI`. En toute généralité, les critères d'ordre sont des expressions calculables pour chaque ligne du résultat : nom de colonne, alias de colonne, fonctions, etc. Il n'est pas nécessaire que les colonnes citées apparaissent dans la liste `select`. On peut donc écrire :

```
order by LOCALITE,upper(NOM),1/(COMPTE-1000)
```

On peut spécifier l'ordre croissant (**asc**, par défaut) ou décroissant (**desc**):

```
order by LOCALITE,upper(NOM) asc,COMPTE desc
```

La notion d'ordre dans les valeurs textuelles est cependant plus complexe qu'il n'y paraît. Elle dépend en effet de l'ordre lexicographique (l'ordre des mots du dictionnaire) adopté par le SGBD, lui-même basé sur l'ordre alphabétique des symboles utilisés. Par défaut, les SGBD adoptent le plus généralement l'ordre des codes ASCII étendu ou UTF-8 pour les caractères de base, c'est-à-dire :

```
'a' < 'b' < ... < 'A' < 'B' < ... < 'Z' < ... < 'à' < ...
```

Malheureusement, cet ordre n'est pas conforme à l'ordre alphabétique auquel nous sommes habitués, dans lequel les caractères 'a', 'à' et 'A' sont consécutifs ou du moins ont même position.

La plupart des SGBD offrent une collection de **collations**, nom technique des ordres alphabétiques disponibles. L'utilisation d'une collation est spécifiée par la clause

```
collate utf8_CI_AI11
```


associée soit à la base de données, soit à une colonne, soit à un index, soit, dans une requête, à un opérateur de comparaison, à la clause `order by` ou la clause `group by`. Quelques exemples d'application :

```
ADRESSE char(60) collate utf8_CI_AI not null,
NOM >= 'MERCIER' collate utf8_CI_AI
order by ADRESSE collate utf8_CI_AI
group by LOCALITE collate utf8_CI_AI
```

À défaut de disposer d'une collation adéquate, on utilisera des fonctions SQL natives :

```
order by lower(ADRESSE)
```

ou définies par l'utilisateur :

```
group by deaccent(ADRESSE)12
```

7.5 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-après 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.

11. Selon les conventions de SQL Server, cette collation est basée sur le codage UTF-8, ignore la casse (case insensitive) et ignore les accents (accent-insensitive).

12. Fonction de SQLfast éliminant les accents de son argument.

Remarque

L'argument des fonctions agrégatives peut être une expression quelconque, 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 évidemment le même résultat.

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

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

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

L'usage du modificateur `distinct` est généralisable à plusieurs fonctions agrégatives. 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 Numeros,
       count(NOM) as Noms,
       count(LOCALITE) as Localites,
       count(CAT) as Categories
from   CLIENT;
```

Numeros	Noms	Localites	Categories
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 Numeros,
       count(distinct NOM) as Noms,
```

```
count(distinct LOCALITE) as Localites,
count(distinct CAT) as Categories
from CLIENT;
```

Numeros	Noms	Localites	Categories
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 null (c'est-à-dire inconnu) pour toutes les autres fonctions.

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

Nombre	Somme	Max
0	<null>	<null>

Le comportement des ensembles vides peut parfois s'avérer surprenant. Si la requête suivante renvoie naturellement un ensemble vide,

```
select COMPTE from CLIENT where LOCALITE = 'Alger';
```

<u>Compte</u>

il n'en est pas de même de la requête suivante, qui renvoie un résultat d'une ligne :

```
select sum(COMPTE) from CLIENT where LOCALITE = 'Alger';
```

<u>sum(Compte)</u>
<null>

7.6 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'autre 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*.

7.6.1 Condition d'association par sous-requête

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 SFW 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')));
```

Si la sous-requête renvoie **une seule ligne**, il est permis d'utiliser les opérateurs de comparaison classiques, comme dans la requête suivante, qui sélectionne les clients dont la valeur du compte est supérieure à celle du client C400 :

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

On remarquera au passage que cette requête mentionne deux fois la table CLIENT. Il n'y a ici aucune confusion possible entre requête principale et sous-requête, la règle étant qu'un nom de colonne est relatif à la (sous-)requête la plus emboîtée dont la table contient une colonne de ce nom. Ainsi, le nom NCLI désigne une colonne de la table CLIENT de la sous-requête et non celle de la requête principale.

7.6.2 Sous-requête et clé étrangère composite

On observera que les sous-requêtes examinées exploitent les liens entre tables formés par le couplage d'un identifiant et d'une clé étrangère. Qu'en est-il des *identifiants* et *clés étrangères composites* ? Considérons à titre d'exemple le schéma de la figure 3.8, et recherchons les comptes auxquels ont été imputés des achats faits le 2017-09-12. On pourra écrire la requête suivante, qui est basée sur une sous-requête qui définit plus deux colonnes :

```
select *
from COMPTE
where (NCLI,NFOURN) in (select NCLI,NFOURN
                        from ACHAT
                        where DATEA='2017-09-12');
```

Tous les SGBD n'acceptent cependant pas cette syntaxe. Nous verrons plus loin d'autres formulations permettant d'exprimer une telle requête (sections 7.7 et 8.8.1).

7.6.3 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 partition de l'ensemble des commandes en quatre sous-ensembles (figure 7.1) :

- C1 : les commandes qui n'ont pas de détails (cas rare !),
- C2 : les commandes qui ont au moins un détail mais dont aucun 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 n'ayant qu'un seul détail, celui-ci spécifiant PA60.

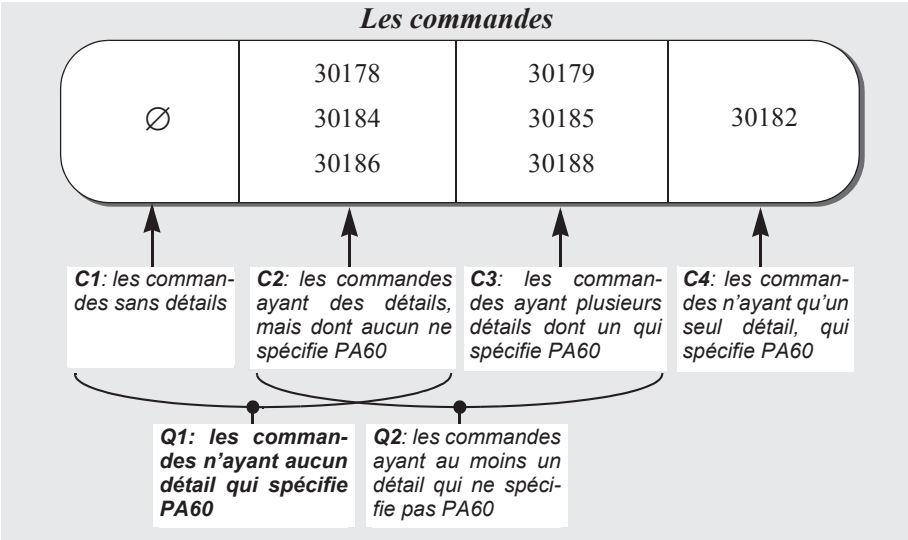


Figure 7.1 - Une partition de l'ensemble des commandes explicitant la portée des requêtes Q1 et Q2. Dans un souci de généralité, on admet qu'une commande puisse n'avoir aucun détail

La requête **Q1** correspond aux catégories C1 et C2, tandis que la requête **Q2** 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 2015*.

1. Les deux ensembles sont A = {les clients de Poitiers} et B = {les commandes de 2015}.
2. Ils s'expriment respectivement par "CLIENT where LOCALITE = 'Poitiers'" et "COMMANDE where DATECOM like '%2015'".
3. Les colonnes en correspondance sont respectivement NCLI dans CLIENT et NCLI dans COMMANDE.
4. D'où la requête :

```
select * from CLIENT
where LOCALITE = 'Poitiers'
and NCLI not in (select NCLI
                  from COMMANDE
                  where DATECOM like '%2015')
```

7.6.4 Références multiples à une même table

La section 7.6.1 comprenait une requête dans laquelle la table CLIENT était citée deux fois. Nous approfondir ce phénomène dans la présente section.

a) Colonne qualifiée et alias de table

Dans une requête comportant une sous-requête, il est possible de mentionner à plusieurs endroits un même nom de colonne, alors que ce nom désigne des colonnes différentes, comme dans l'exemple ci-après :

```
select NCOM, DATECOM
from COMMANDE
```

```
where NCLI in (select NCLI
                from CLIENT
                where LOCALITE = 'Poitiers');
```

Pour éviter les éventuelles ambiguïtés (ce qui n’est pas le cas ici) on peut qualifier les colonnes en préfixant leur nom du nom de leur table comme suit :

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

De même qu’on peut assigner un surnom à une colonne d’un résultat (alias de colonne), on peut aussi assigner un surnom à une table de la clause from, ce qu’on appelle un **alias de table**. Cet alias remplacera le nom de la table, là où il devra apparaître (le terme **as** est optionnel) :

```
select NCOM, COM.DATECOM
from COMMANDE as COM
where COM.NCLI in (select CLI.NCLI
                  from CLIENT as CLI
                  where LOCALITE = 'Poitiers');
```

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. À 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 CLI
where COMPTE > (select avg (COMPTE)
                from CLIENT
                where LOCALITE = CLI.LOCALITE);
```

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

Dans cette requête, l’alias CLI désigne chaque ligne de la table CLIENT de la requête principale, ligne pour laquelle on va évaluer la condition **where**. L’expression LOCALITE de la sous-requête désigne la valeur de LOCALITE de la ligne de CLIENT

dans la sous-requête tandis que l'expression `CLI.LOCALITE` désigne la valeur de `LOCALITE` de la ligne de `CLIENT` en cours d'examen dans la requête principale.

c) 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 trois détails.

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

d) null = null ?

C'est l'occasion d'examiner d'un peu plus près les mœurs 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 par principe comparable à rien, même pas à lui-même, puisqu'il est généralement interprété comme *inconnu*. Nous rediscuterons de cette question plus tard (section 9.12).

7.7 LES QUANTIFICATEURS ENSEMBLISTES

Les quantificateurs ensemblistes permettent d'imposer qu'un ensemble défini possède *au moins un élément* ou *au moins un élément qui satisfait une condition particulière*.

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 sont commandés pour l'instant, c'est-à-dire ceux pour lesquels il existe au moins une ligne de `DETAIL`¹³. 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 `P.NPRO`.

13. Il existe d'autres formulations de cette requête qui utilisent la relation de comparaison `in` ou la fonction agrégative `count()`. Quelles sont-elles ?

```

select NPRO, LIBELLE
from   PRODUIT as P
where  exists (select *
                from   DETAIL
                where  NPRO = P.NPRO);

```

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 commandes spécifiant la plus petite quantité du produit PA60, c'est-à-dire les commandes pour lesquelles il n'existe pas de détails spécifiant une quantité inférieure :

```

select distinct NCOM
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 suivante, 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 composites

Reportons-nous une fois de plus au schéma de la figure 3.8, et recherchons, comme nous l'avons déjà fait à la section 7.6.2, *les comptes auxquels ont été attachés des achats faits le 2015-09-12*. Les quantificateurs ensemblistes nous permettent d'exprimer cette requête d'une manière élégante :

```

select *
from   COMPTE C

```

```

where exists (select 1
              from   ACHAT
              where   NCLI = C.NCLI
              and     NFOURN = C.NFOURN
              and     DATEA = '2015-09-12')) ;14

```

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 offrir le quantificateur universel `for all` (traduisant $\forall$). Ce dernier n'existe cependant pas, de sorte qu'il nous faudra tourner ce type de condition d'une autre manière.

Pour traduire une condition qui serait basée sur ce quantificateur, nous exploiterons une équivalence remarquable de la logique :

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

Par exemple, pour rechercher *les commandes qui réfèrent tous les produits*, nous sélectionnerons les commandes selon la règle suivante :

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.**

Cette règle pourrait encore être exprimée d'une autre manière : considérons une situation dans laquelle **un produit n'est pas référencé par M** ; nous affirmons qu'**une telle situation ne peut exister**. Nous étudierons cette traduction de manière plus approfondie à la section A7.1 de l'annexe A7.

7.8 EXERCICES

Exprimer en SQL les requêtes suivantes.

- 7.1 Afficher la liste des localités dans lesquelles il existe au moins un client.
- 7.2 Afficher le numéro, le nom et la localité des clients de catégorie C1 n'habitant pas à Toulouse.
- 7.3 Donner le numéro, le nom et le compte des clients de Poitiers et de Bruxelles dont le compte est positif.
- 7.4 Quelles catégories de clients trouve-t-on à Toulouse ?
- 7.5 Afficher le numéro, le nom et la localité des clients dont le nom précède alphabétiquement la localité où ils résident.

14. On a remplacé * par 1 (ou toute autre constante). En réalité, le seul fait qui nous intéresse c'est un ensemble vide ou non vide. Qu'il soit constitué de lignes ou de constantes est indifférent.

- 7.6 Afficher les localités des clients qui commandent le produit de numéro 'CS464'.
- 7.7 Quels sont les produits en sapin qui font l'objet d'une commande ?
- 7.8 Afficher la valeur totale des stocks (compte non tenu des commandes actuelles).
- 7.9 Combien y a-t-il de commandes spécifiant un (ou plusieurs) produit(s) en acier ?
- 7.10 Afficher le numéro et le nom des clients qui n'ont pas commandé de produits en sapin.
- 7.11 Quels sont les produits (numéro et libellé) qui n'ont pas été commandés en 2015 ?
- 7.12 Rechercher les clients qui ont commandé tous les produits.
- 7.13 Rechercher les localités dont aucun client n'a passé de commande.
- 7.14 Rechercher les localités dont tous les clients ont passé au moins une commande.

Exercices corrigés

- Exercice 7.1

```
select distinct LOCALITE from CLIENT;
```

- Exercice 7.2

```
select NCLI,NOM,LOCALITE from CLIENT  
where CAT = 'C1' and LOCALITE <> 'Toulouse';
```

- Exercice 7.3

```
select NCLI,NOM,COMPTE from CLIENT  
where LOCALITE in ('Poitiers','Bruxelles') and COMPTE > 0;
```

- Exercice 7.4

```
select distinct CAT from CLIENT  
where LOCALITE = 'Toulouse' and CAT is not null;
```

- Exercice 7.5

```
select NCLI,NOM,LOCALITE from CLIENT  
where NOM < LOCALITE;
```

- Exercice 7.6

```
select distinct LOCALITE from CLIENT  
where NCLI in (select NCLI from COMMANDE  
               where NCOM in (select NCOM from DETAIL  
                              where NPRO = 'CS464'));
```

- **Exercice 7.7**

```
select NPRO from PRODUIT
where LIBELLE like '%SAPIN%'
and exists (select * from DETAIL
            where NPRO = PRODUIT.NPRO);
```

ou

```
select NPRO from PRODUIT
where LIBELLE like '%SAPIN%'
and NPRO in (select NPRO from DETAIL);
```

- **Exercice 7.8**

```
select sum(QSTOCK*PRIX) as TOTAL from PRODUIT;
```

- **Exercice 7.9**

```
select count(*) from COMMANDE
where NCOM in (select NCOM from DETAIL
               where NPRO in
                 (select NPRO from PRODUIT
                  where LIBELLE like '%ACIER%'));
```

- **Exercice 7.10**

```
select NCLI,NOM from CLIENT where NCLI
not in (select NCLI from COMMANDE where NCOM
        in (select NCOM from DETAIL where NPRO
            in (select NPRO from PRODUIT
                where LIBELLE like '%SAPIN%'))));
```

- **Exercice 7.11**

```
select NPRO,LIBELLE from PRODUIT where NPRO
not in (select NPRO from DETAIL where NCOM
        in (select NCOM from COMMANDE
            where DATECOM like '2015%'));
```

- **Exercice 7.12**

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.

```
select NCLI from CLIENT C
where not exists (select * from PRODUIT where NPRO
                  not in (select NPRO from DETAIL
                          where NCOM
                            in (select NCOM from COMMANDE
                                where NCLI = C.NCLI))));
```

- **Exercice 7.13**

```
select distinct LOCALITE from CLIENT where LOCALITE
not in (select LOCALITE from CLIENT C
        where exists (select * from COMMANDE
                      where NCLI = C.NCLI));
```

- **Exercice 7.14**

```
select distinct LOCALITE from CLIENT C
where not exists (select * from CLIENT
```

```
where  LOCALITE = C.LOCALITE  
and    NCLI not in (select NCLI  
                        from  COMMANDE) );
```

Des exercices et solutions additionnels, ainsi que des matériaux complémentaires (scripts SQLfast), sont disponibles dans l'annexe A7

Chapitre 8

Le langage SQL DML (2)

Ce chapitre poursuit l'étude du langage SQL DML. On y aborde plus particulièrement l'extraction de données de plusieurs tables (via les jointures), les opérateurs ensemblistes, les extensions de la requête SFW, le traitement des structures de données cycliques, l'extraction de données groupées et les opérateurs de modification de données et de structures de données.

8.1 INTRODUCTION

Nous compléterons dans ce chapitre l'examen et l'application du langage SQL DML en abordant des variantes plus puissantes de la requête `select` et la question de la modification des données et de leur structure. 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. Nous étudierons en outre une classe d'applications particulièrement utile bien que délicates, basées sur les structures de données cycliques.

Dans la suite de cet exposé, nous aborderons principalement :

1. l'extraction de données de plusieurs tables au moyen de l'opérateur de jointure (8.2),
2. les opérateurs ensemblistes (union, intersection, différence) (8.3),
3. le traitement des structures de données cycliques, dans lesquelles une ligne fait, directement ou indirectement référence à une ligne de la même table (8.7),

4. l'extraction de données groupées, par lesquelles on ne s'intéresse plus à des lignes individuelles mais à des groupes de données (8.9),
5. les extensions de la requête SFW lui assurant une plus grande régularité (8.10),
6. la modification des données (insertion, suppression, modification de lignes) (section 8.13) et des structures de données (8.13 et 8.14).

8.2 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.

8.2.1 La jointure de plusieurs tables

Considérons la table suivante :

NCOM	NCLI	DATECOM	NOM	LOCALITE
30178	K111	2015/12/21	VANBIIST	Lille
30179	C400	2015/12/22	FERARD	Poitiers
30182	S127	2015/12/23	VANDERKA	Namur
30184	C400	2015/12/23	FERARD	Poitiers
30185	F011	2016/01/2	PONCELET	Toulouse
30186	C400	2016/01/2	FERARD	Poitiers
30188	B512	2016/01/3	GILLET	Toulouse

Ses données sont manifestement issues de notre base de données. Les colonnes NCOM, NCLI et DATECOM sont extraites de la table COMMANDE tandis que les colonnes NCLI, NOM et LOCALITE ont été tirées de la table CLIENT. La colonne NCLI est commune aux deux tables. Ses valeurs ont été utilisées pour mettre en correspondance les lignes des deux tables d'origine.

Cette table a été obtenue par une **jointure** (*join*) entre la table COMMANDE et la table CLIENT. Cette jointure est un couplage ligne par ligne des deux tables. Deux lignes sont couplées si elles ont la même valeur de NCLI. La jointure s'obtient en SQL par une requête de la forme suivante :

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

On y précise d'abord les tables sources (clause **from**), la règle d'association des lignes de ces deux tables (clause **where**) et les valeurs extraites des deux tables (clause **select**). La règle d'association se présente généralement sous la forme de l'égalité des valeurs de deux colonnes. 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 (fictive) 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 précédemment, 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 les deux conditions de 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`).

8.2.2 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 associations 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 31 décembre 2015.

```
select NCOM, CLIENT.NCLI, DATECOM, NOM, ADRESSE
from COMMANDE, CLIENT
where COMMANDE.NCLI = CLIENT.NCLI
and CAT = 'C1' and DATECOM < '2014-12-23';
```

8.2.3 Lignes célibataires et jointures externes

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 correspondant 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. Si nous voulons que chaque ligne de CLIENT apparaisse dans le résultat de la jointure, nous devons y ajouter les lignes célibataires, qui s'obtiennent par la requête ci-dessous :

```
select null as NCOM,NCLI,null as DATECOM,NOM,LOCALITE
from   CLIENT
where  not exists (select * from COMMANDE
                  where  NCLI = CLIENT.NCLI) ;
```

On observe que les données non pertinentes ont été remplacées par des valeurs null.

Il existe une variante spéciale de la jointure, appelée **jointure externe** (*outer join*), dont la particularité est de compléter la jointure naturelle¹ par les lignes célibataires d'une ou des deux tables sources. On écrira :

```
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.

8.3 LES OPÉRATEURS ENSEMBLISTES

Rappelons d'abord qu'un *ensemble* est une collection d'éléments distincts. Un ensemble de lignes ne peut donc contenir deux lignes dont toutes les colonnes, considérées deux à deux, ont la même valeur. Une collection de lignes dont on n'est pas certain que les éléments sont distincts constitue un *multi-ensemble*². Une requête dont la liste d'éléments de la clause **select** n'inclut pas tous les éléments d'un identifiant renvoie un multi-ensemble de lignes, qu'on réduira si nécessaire à un ensemble par le modifieur **distinct**.

L'opérateur SQL **union** permet de réunir dans un même ensemble les lignes de deux tables distinctes, de base ou calculées. On insiste sur le fait qu'il produit un

1. C'est le nom de la jointure décrite précédemment. Elle se distingue de la *jointure interne* par le fait qu'elle ne conserve qu'un seul exemplaire des colonnes de jointure.

2. Certains parleront d'*ensembles avec doublons*. Le terme est intuitif mais impropre.

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 bien sûr qu'une seule fois dans l'union. Plus curieux cependant : les arguments de l'opérateur **union** peuvent être des multi-ensembles de lignes, auquel cas leurs doublons sont également éliminés. À titre d'illustration, on comparera la requête :

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

LOCALITE
Poitiers
Namur
Poitiers
Namur
Namur

avec la suivante, qui lui *ajoute* des éléments :

```
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 doublons, on utilisera l'opérateur **union all**. Une même ligne qui apparaît **n1** fois dans le premier membre et **n2** fois dans le second apparaîtra **n1+n2** fois dans le résultat, comme le montre la requête suivante.

```
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 *n1* et *n2* exemplaires dans chaque argument,

- ***intersect all*** produira $\min(n1, n2)$ exemplaires de cette ligne,
- ***except all*** produira $\max(n1 - n2, 0)$ exemplaires de cette ligne.

On trouvera à la section A24, une étude détaillée de ces opérateurs.

Remarque sur l'intersection et la différence ensemblistes

Les opérateurs d'intersection et de différence **entre deux ensembles de lignes** ne sont pas strictement indispensables (d'ailleurs certains SGBD ne les proposent pas) dans la mesure où ils peuvent être exprimés simplement par les requêtes standards.

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

La *différence ensembliste* 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* ou la jointure externe (on sélectionne les lignes dont les valeurs complémentaires sont null).

8.4 LE 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). Cette opération, qui porte le nom de ***produit relationnel***, nous a permis de comprendre le mécanisme de la jointure, mais n'offre pas grand intérêt en soi. Citons cependant deux applications intéressantes 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 à présent tous les couples *effectifs*, indiquant 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 très intéressé par les **produits qu'on ne commande pas** dans chaque commune, et pour lesquels un effort d'information serait utile :

```

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

```

Le second exemple implique une table TVA(TAUX) d'une seule ligne fournissant un paramètre de calcul pour la seconde table :

```

select NPRO, PRIX, PRIX*TAUX as PRIX_HTVA
from PRODUIT, TVA

```

8.5 VARIANTES SYNTAXIQUES DE LA 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 8.2, 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, si toutes les autres colonnes portent des noms différents dans les deux tables, et si on ne retient qu'une seule fois les colonnes de jointure, on peut écrire :

```

select *
from CLIENT natural join COMMANDE
where ...

```

dont la formulation classique est :

```

select C.NCLI, NOM, ADRESSE, LOCALITE, CAT, COMPTE, NCOM, DATECOM
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
and    ...;
```

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 précédente, 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 standard SQL 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.NCLI³. Il est permis d'utiliser le qualifieur inner, facultatif :

```
from CLIENT inner join COMMANDE using (NCLI)
```

Conclusion

Ces formes n'apportent aucune valeur ajoutée par rapport à la formulation classique d'une jointure mais peuvent entraîner une complexité inutile et donc un risque d'erreurs. Sauf dans les jointure simples, on leur préférera autant que possible les

3. Cette remarque n'a en soi que peu d'intérêt, mais illustre la complexité inutile induite par ces formes de jointure.

expressions régulières proposées en 8.2. A titre d'exemple, on examinera la requête ci-dessous tirée d'une application Access fournie dans la distribution d'Office. Les noms de tables et de colonnes ont été anonymisés mais la structure a été conservée.

```
select distinct *
from S inner join (P inner join ((E inner join
(C inner join A on C.I = A.CI) on E.I = A.EI)
inner join B on A.I = B.OI) on P.I = B.PI) on S.I = A.S;
```

8.6 LES TABLES STATIQUES

De même que dans une expression arithmétique telle que $x + 5$ interviennent des variables (x) et des constantes (5), une expression ensembliste de tables peut inclure des *tables constantes*, qualifiées de *statiques*. Une table statique comprenant les deux lignes ('CS464', 25) et ('PA60', 12) est définie comme suit :

```
values ('CS464', 25), ('PA60', 12)
```

forme qu'on rencontre dans la requête `insert`. Cette table et ses colonnes ne portent pas de nom. Elle apparaîtra généralement comme composante de requêtes plus complexes, telles que la suivante :

```
select distinct NPRO,QCOM from DETAIL
union
values ('CS464', 25), ('PA60', 12);
```

Une table d'une ligne peut aussi s'exprimer par une forme dégénérée de la requête `select` (*zero-table select*) :

```
select 'CS464', 25
```

8.7 REQUÊTES SUR DES STRUCTURES CYCLIQUES

On qualifie de **cyclique** (ou *réursive*) une structure de données qui fait, directement ou non, référence à elle-même. La table PERSONNE illustrée à la figure 8.1 présente une structure cyclique car la colonne RESPONSABLE constitue une clé étrangère vers la table PERSONNE elle-même. Le rôle de cette colonne facultative est de désigner le responsable direct de chaque personne, s'il existe.



Figure 8.1 - Un schéma cyclique : la table PERSONNE se référence elle-même

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

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

La figure 8.2 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.

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 8.2 - Un exemple de contenu de la table PERSONNE⁴

On peut donner du contenu de cette table une représentation graphique qui rend compte de la structure hiérarchique⁵ que ces données définissent (figure 8.3).

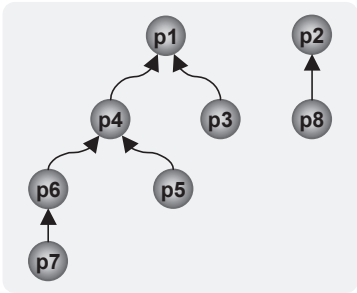


Figure 8.3 - Représentation graphique du contenu de la table 8.2

4. Dans un but de lisibilité, les valeurs <null> ont été représentées par "--".
5. On admettra qu’une personne ne peut être son propre responsable, ni directement ni indirectement. Par conséquent, le graphe ne peut comporter de circuits. En fait, cette contrainte ne peut être explicitement 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). Nous reviendrons sur ce point à la section 16.12.4.

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. En SQL :

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

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**.

Pour l'interpréter, supposons qu'on dispose de deux tables décrivant les personnes (figure 8.4). 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 précédemment, ces tables contiendraient les données de la figure 8.4. 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;
```

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

NPERS	NOM	(RESPONSABLE)
p1	Mercier	--
p2	Durant	--
p4	Dupont	p1
p6	Dupont	p4

Figure 8.4 - 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.

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';
```

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. Pour ce type d'extraction, nous aurons besoin des requêtes récursives⁶, qui seront décrites au chapitre 9, section 9.5.

Un autre exemple classique de structure cyclique décrit une **nomenclature de produits** (*bill of materials* ou *BOM*). 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 8.5 illustre cette structure.

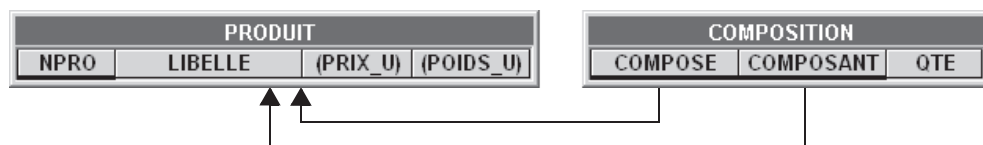


Figure 8.5 - Une structure indirectement cyclique :
nomenclature d'une gamme de produits

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'avons 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 (h, b, q) 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 p_2 contient 8 unités du produit p_7 et 2 unités du produit p_8 . La figure 8.6 représente un exemple de nomenclature, représentée graphiquement à la figure 8.7.

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.

6. Le lecteur intéressé par les requêtes (récursives) appliquées aux structures de graphes consultera utilement les chapitres 26 et 27 de [Celko,2000].

La requête suivante donne des informations sur le produit p4, ainsi que sur sa composition⁷ :

```
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' ;
```

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

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 8.6 - 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

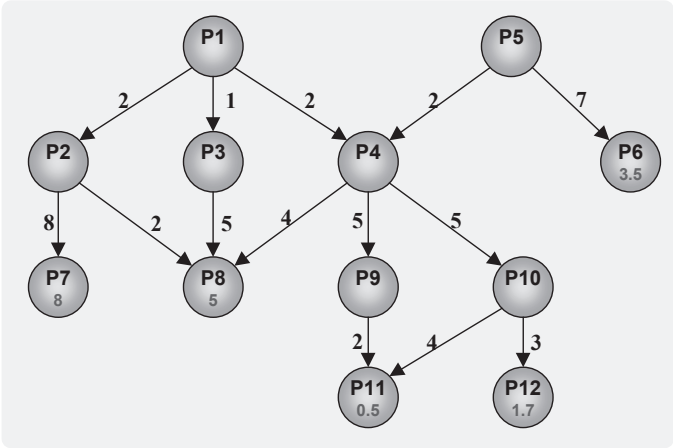


Figure 8.7 - Représentation graphique du contenu des tables 8.6

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

est-elle équivalente à :

```
select distinct 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 8.8.3.

En revanche, certaines structures de sous-requête **ne peuvent pas s'exprimer** par une jointure. Tel serait le cas de la recherche des commandes qui **ne spécifient pas** PA60, vue auparavant (section 7.6.3) et qu'on rappelle sous ses deux formes :

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

```
select NCOM, DATECOM, NCLI
from   COMMANDE C
where  not exists (select 1 from DETAIL
                  where  NCOM = C.NCOM);
```

Ces requêtes ne sont en effet absolument pas équivalentes aux tentatives suivantes :

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

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

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**, laquelle exige une sous-requête (not in ou not exists).

Cependant, il est possible d'exprimer cette inexistence par le mécanisme de jointure externe. Les clients *qui n'ont pas passé de commande* correspondent aux lignes

célibataires d'une jointure naturelle, lesquelles sont présentes dans une jointure externe :

```
select C.NCLI,NOM
from   CLIENT C left outer join COMMANDE M
      on (C.NCLI = M.NCLI)
where  M.NCOM is null;
```

b) Sous-requête et clé étrangère composite

Cette question a déjà été abordée dans la section 7.7. Nous pouvons ici lui apporter une réponse supplémentaire. Il s'agit de rechercher (figure 3.8) *les comptes auxquels ont été attachés des achats faits le 2015-09-12*. 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 = '2015-09-12';
```

8.8.2 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;
```

8.8.3 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 8.8, qui comporte une table VENTE dont chaque ligne (c, p, x) indique que dans un magasin de la chaîne c , le produit p est vendu au prix x , ainsi qu’une table IMPLANTATION dont chaque ligne (c, v) 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 IMPLANTATION.

VENTE		
CHAINE	PRODUIT	PRIX

IMPLANTATION	
CHAINE	VILLE

Figure 8.8 - Deux tables logiquement corrélées via les colonnes CHAINE, mais sans clés étrangères

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. L’expression :

```
select distinct PRODUIT, VILLE, PRIX
from   VENTE V, IMPLANTATION I
where  V.CHAINE = I.CHAINE;
```

indique, pour chaque ligne (p, v, x) 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 8.9 établit des intervalles successifs de valeurs de compte (MIN_CPT et MAX_CPT), et attribue un code à chacun (CODE_CPT). Par exemple, la valeur de COMPTE 5600 reçoit le code B.

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 8.9 - Table de classification des valeurs des comptes des clients

Nous aimerions associer à chaque client le code de son compte. C'est ce que réalise la requête suivante pour les clients de catégorie C1 :

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

NCLI	NOM	CODE_CPT
B112	HANSENNE	D
C123	MERCIER	X
F010	TOUSSAINT	G
L422	FRANCK	G
S127	VANDERKA	X

8.8.4 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.

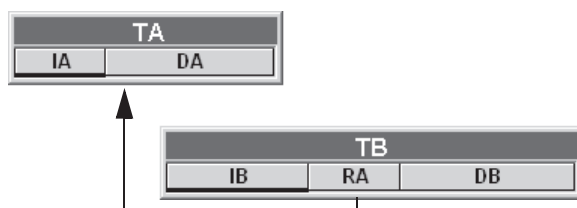
a) Quelles entités représente le résultat d'une jointure ?

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 :

```
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 ? Formulons la question autrement : 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 commandes ? Aussi étrange que cela paraisse, la réponse est que *cette requête représente des commandes*. 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, et dont 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**.

À titre d'exemple supplémentaire, on vérifie que 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;
```

L'interprétation d'une jointure multiple se fait de manière itérative : on évalue une première jointure, dont le résultat est joint à la table suivante⁸, 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 multiple, 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 2.7, dont la racine est DETAIL. Cette règle est aussi applicable à une jointure PRODUIT * COMPOSITION * PRODUIT (figure 8.5) qui associerait ses composants à chaque produit, et qui représenterait des *compositions*.

Intuitivement toujours, et plus généralement, la jointure est formée de fragments qui obéissent à un des patterns 1, 2 ou 3 de la figure 8.10, où un rectangle représente une table apparaissant 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)

La règle d'interprétation est d'application pour les patterns 1 à 3, mais pas pour le pattern 4, où une jointure est définie entre les tables A, B et C, telles que A est réf-

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

rencée par B et C. Cette jointure, plutôt rare, représente des entités nouvelles, résultant du croisement des entités de B et des entités de C via A⁹.

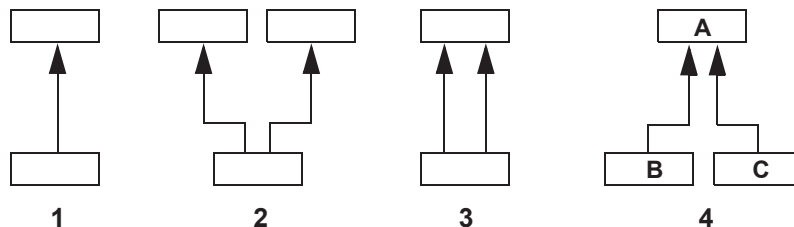


Figure 8.10 - Une jointure fournit des informations sur la racine de son graphe si celui-ci est constitué de fragments des types 1, 2 et 3, mais pas du type 4

b) Identifiant d'une jointure

Ce que nous venons de discuter conduit à une règle simple : 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 principe¹⁰ 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 (curieusement) les détails de commande, mais pour chacun d'eux on ne retient que la localité du client de la commande et 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 ($v, 1$) de ce résultat représenterait alors le fait que dans la localité v , on a commandé au moins une fois un produit de libellé 1.

9. 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*.

10. 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.

8.9 EXTRACTION DE DONNÉES GROUPÉES

Les requêtes examinées dans les sections précédentes¹¹ 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.

8.9.1 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 (figure 8.11), ou selon leur catégorie, ou encore selon leur nom. Les grouper selon leur identifiant NCLI constituerait des groupes d'une seule ligne, ce qui n'aurait guère d'intérêt

CLIENT			
NCLI	NOM	LOCALITE	COMPTE
F400	JACOB	Bruxelles	0
B332	MONTI	Genève	0
K111	VANBIST	Lille	720
S127	VANDERKA	Namur	-4580
L422	FRANCK	Namur	0
C123	MERCIER	Namur	-2300
B062	GOFFIN	Namur	-3200
S712	GUILLAUME	Paris	0
F010	TOUSSAINT	Poitiers	0
B112	HANSENNE	Poitiers	1250
C400	FERARD	Poitiers	350
C003	AVRON	Toulouse	-1700
B512	GILLET	Toulouse	-8700
F011	PONCELET	Toulouse	0
K729	NEUMAN	Toulouse	0
D063	MERCIER	Toulouse	-2250

Figure 8.11 - Un extrait de la table CLIENT dont les lignes ont été regroupées selon les valeurs de LOCALITE

La requête suivante donne, pour chaque groupe de clients *classés* ou *regroupés par localité*, le nom de celle-ci, le nombre de clients dans le groupe et la valeur moyenne des comptes des clients du groupe.

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

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

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

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 constantes¹².

On observe cependant que certains SGBD (MySQL et SQLite notamment) violent cette règle. Ils autorisent ainsi dans la requête précédente, la clause **select** suivante :

```
select LOCALITE,NOM,count (*) ... as MOYENNE_COMPTES
```

Puisqu’il y a plusieurs valeurs de NOM par groupe et que la requête n’en affiche qu’une seule, celle-ci est en réalité **tirée au hasard**. Le seule situation où cette liberté serait acceptable est le cas d’une colonne surnuméraire *dépendant fonctionnellement* des colonnes du critère de groupement, garantissant ainsi une seule valeur par groupe.

Les fonctions agrégatives calculables sur les groupes sont celles qui ont été décrites à la section 7.5. Dans le chapitre suivant, nous étudierons une nouvelle fonction particulièrement utile, **group_concat**, qui a pour but de transformer une table en liste.

8.9.2 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 avant groupement. Dans la requête suivante, on ne retient par exemple que les groupes d’au moins trois clients :

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

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

ce qui donne :

LOCALITE	Nombre	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. L'usage d'alias de colonne donne une forme plus élégante :

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

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

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

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

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

8.9.3 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`.

```

select NCLI, count(*) as Nombre, sum(QCOM) as Quantite
from   COMMANDE M, DETAIL D
where  M.NCOM = D.NCOM
and    NPRO = 'PA45'
group by NCLI
having Nombre >= 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 ',M.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;

```

8.9.4 Composition de la liste `select`

Nous voudrions ajouter au résultat précédent le nom de chaque client. Ce désir est *logiquement légitime* puisqu'il n'y a qu'une seule valeur de NOM par groupe. Cependant, nous savons que seuls des éléments du critère de groupement (outre les fonctions agrégatives) sont autorisés dans la clause `select`. Nous sommes donc forcé d'écrire la requête comme suit, ce qui, il faut l'admettre, n'est guère élégant :

```

select 'Montant dû par ',M.NCLI,NOM, ' = ',sum(QCOM*PRIX)
...
group by M.NCLI,NOM;

```

La requête suivante calcule la quantité qui resterait en stock pour chaque produit si on en 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'agrégation¹³. 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;

```

13. Variante suggérée par un lecteur : `select P.NPRO, avg(QSTOCK) - sum(QCOM) as SOLDE . . . group by P.NPRO`

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

8.9.5 Composition du critère de groupement

Nous venons de le voir, le critère de groupement peut inclure plusieurs noms de colonne. L'exemple supplémentaire ci-après 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 Cat1, count(*) as N
from   CLIENT
group by substring(CAT from 1 for 1);
```

Cat1	N
<null>	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 suivante.

```
select int(COMPTE/1000)*1000
+ case sign(COMPTE) when -1 then -1000 else 0 end as Min,
int(COMPTE/1000)*1000
+ case sign(COMPTE) when -1 then -1 else 999 end as Max,
count(*) as N
from   CLIENT C group by int(COMPTE/1000);
```

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

8.9.6 Attention aux groupements multiniveaux

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** (outre le fait que les clients sans commande 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 comptabilisé trois fois, pour un total de 1 050 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, soit procéder en deux étapes indépendantes, soit utiliser une sous-requête dans la clause `select` (voir section 8.10.1).

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 suivante 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;

```

8.9.7 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 50 unités en 2015. 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 '2015%'
group by D.NPRO
having sum(QCOM) > 50;

```

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 '2015%')) > 50;

```

8.10 EXTENSION DE LA STRUCTURE DES REQUÊTES SFW

SQL autorise certaines formes présentant une plus grande régularité dans l'écriture des requêtes en admettant l'usage généralisé de requêtes SFW comme expression de désignation de valeurs individuelles et de tables.

8.10.1 Extension de la clause *select*

La clause `select` permet de spécifier les données 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`. Toute expression qui renvoie une valeur pour chaque ligne est valide : nom de colonne, constante, fonction. Il est aussi possible d'y inclure une requête SFW pourvu qu'elle renvoie une seule valeur. Pour obtenir le montant des commandes de montant supérieur à 1 000, on pourra écrire :

```

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;

```

La colonne MONTANT est définie par une sous-requête apparaissant dans la liste **select**. Le même résultat pourrait s'obtenir par une jointure et un groupement, mais l'expression serait sans doute jugée moins naturelle.

8.10.2 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. Il est cependant possible d'y spécifier des tables dérivées, résultant d'opérations produisant des tables. Nous en examinerons deux : la requête SFW elle-même et les expressions ensemblistes.

a) Les requêtes SFW

L'évaluation d'une requête SFW a pour résultat une table d'une ou plusieurs colonnes. Il est permis d'utiliser cette table comme source de lignes pour une autre requête SFW sans qu'il soit nécessaire de la stocker dans une table temporaire ou de la définir comme une vue (section 9.3.6). La requête suivante calcule la moyenne des montants des commandes¹⁴ :

```

select avg (MONTANT)
from   (select NCOM, sum(QCOM*PRIX) as MONTANT
        from   DETAIL D, PRODUIT P
        where  D.NPRO = P.NPRO
        group by NCOM) ;

```

b) 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**. Considérant les tables BON_CLIENT (décrivant un sous-ensemble des clients) et PROSPECT (reprenant les clients potentiels), de même schéma que CLIENT, on pourra écrire¹⁵ :

```

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';

```

14. On ne peut pas écrire : `avg (sum (QCOM*PRIX) )`, expression qui serait ambiguë.

15. 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.

On peut associer un alias à la table dérivée de la clause `from` ainsi qu'à ses colonnes, ce qui permet de les utiliser dans les clauses `select` et `where` :

```
select NPRO, TOTAL_QTE
from ( select NPRO, sum(QCOM) from DETAIL
      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;
```

8.11 LIMITATION DE LA TAILLE D'UN RÉSULTAT

Lorsque le résultat d'une requête comporte un grand nombre de lignes, il peut être utile de n'en recevoir qu'un sous-ensemble défini par le rang de ses lignes. On voudrait par exemple recevoir les 10 premières lignes, puis les 10 suivantes, ou encore les 5 dernières. Le standard SQL propose de définir l'intervalle des lignes à recevoir comme suit :

```
select * from CLIENT
order by NCLI offset 4 rows fetch first 5 rows only;
```

Les 4 premières lignes sont sautées, puis les 5 suivantes sont transmises. On reçoit donc les lignes de rang 5 à 9. Elle est adoptée par SQL Server 2012 et Oracle 12c, mais d'autres (MySQL, SQLite) proposent une expression plus légère :

```
select * from CLIENT
order by NCLI limit 5 offset 4;
```

Cette spécification d'intervalle de lignes n'a évidemment de sens que si le résultat est ordonné par une clause `order by`. Il est à noter que le résultat n'est pas déterministe si le critère de tri n'est pas identifiant : avec une clause `order by LOCALITE`, les lignes de même valeur de `LOCALITE` se présentent de manière aléatoire.

8.12 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 `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`.

À titre d'exemple, on indique, pour la requête suivante, 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

```

8.13 MODIFICATION DES DONNÉES

On présente les opérateurs qui permettent l'introduction (`insert`), la suppression (`delete`) et la modification (`update`) des données d'une table puis on rappelle le comportement du SGBD en présence de contraintes référentielles.

8.13.1 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 (`default`) 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 lors de sa déclaration.

En toute généralité, l'argument de l'instruction `insert` est une *table statique* telle que nous l'avons décrite à la section 8.6. On peut donc aussi écrire :

```
insert into DETAIL values ('30178','PA45',12),
                          ('30178','CS262',5);
```

Chaque valeur peut être exprimée sous la forme d'une constante, comme illustré ici, mais plus généralement de toute expression dont l'évaluation donne une valeur : expression calculée ou requête SFW par exemple :

```
insert into DETAIL
values ('30178','PA45',(select avg(QCOM) from DETAIL
                        where NPRO = 'PA45'));
```

Il est possible d'insérer dans une table existante les lignes d'une *table dérivée* définie par une expression SFW. Dans la requête suivante, on ajoute à la table `CLI_TOULOUSE` (NUMERO,NOM,ADRESSE) des données extraites de `CLIENT` :

```
insert into CLI_TOULOUSE
select NCLI, NOM, ADRESSE
from   CLIENT
where  LOCALITE = 'Toulouse';
```

À partir du moment où les données ont été insérées, la table `CLI_TOULOUSE` devient indépendante de la table `CLIENT`. Une mise à jour des lignes de `CLIENT` n'aura pas d'effet sur le contenu de la table `CLI_TOULOUSE`, ni inversement.

Dans tous les cas, les données insérées doivent respecter les contraintes d'intégrité (unicité, 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 de consultations après leur remplissage.

On appelle parfois *instantané* (*snapshot*) ou *vue matérialisée* ce type de table. 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 ceux-ci.

Note

En principe, l'insertion d'une ligne violant une contrainte d'unicité est rejetée. Il existe cependant une variante, dite *upsert* (*update/insert*), de cette requête qui, lorsqu'il existe déjà une ligne de même identifiant, en effectue la modification¹⁶.

16. Le verbe `insert` est alors remplacé par `merge`, `insert or update`, `insert on duplicate key`, etc., selon le SGBD.

8.13.2 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 = 'B062';
```

Cet ordre supprime de la table `CLIENT` la ligne qui décrit le client numéro `B062`. L'instruction suivante supprime de la table `DETAIL` les lignes, quel qu'en soit le nombre, qui spécifient un produit en rupture de stock.

```
delete from DETAIL
where NPRO in (select NPRO
                from   PRODUIT
                where  QSTOCK <= 0);
```

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.

8.13.3 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 augmente de 5% le prix des produits en sapin :

```
update PRODUIT
set     PRIX = PRIX * 1.05
where   LIBELLE like '%SAPIN%';
```

Ces valeurs peuvent également provenir de la base de données elle-même. La requête ci-après déduit de la quantité en stock de chaque produit la somme des quantités actuellement en commande.

```
update PRODUIT
set     QSTOCK = QSTOCK - (select sum(QCOM)
                          from   DETAIL
                          where  NPRO = PRODUIT.NPRO)
where   exists (select * from DETAIL
                where  NPRO = PRODUIT.NPRO);
```

On notera l'usage du préfixe "`PRODUIT.`" 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 le déclenchement d'erreurs. En effet, pour un produit P qui n'a pas de détails :

- l'ensemble défini par DETAIL where NPRO = P.NPRO est vide,
- l'expression sum(QCOM) appliquée à cet ensemble vide renvoie null,
- l'expression QSTOCK - null vaut null,
- et donc l'opération sera refusée puisque la colonne QSTOCK a été déclarée not null. Si QSTOCK avait été facultative, sa valeur aurait été détruite pour tous les produits sans détail !

Plus simplement, on peut éviter la condition de protection en remplaçant "sum(QCOM)" par "coalesce(sum(QCOM), 0)".

Lorsque des données de remplacement de plusieurs colonnes sont issues de la même source, la forme de la requête peut devenir compliquée, voire inefficace. On pourra alors appliquer l'assignation de n-uplets, comme suit :

```
update PRODUIT
set (DESCRIPTION,PRIX) = (select DESCR,PRIX_U
                           from REFERENCE
                           where REFID = PRODUIT.NPRO);
```

8.13.4 À propos de la gestion des verrous

Nous avons vu que la consultation de données entraînait la pose de verrous partagés sur celles-ci et que si, par la suite, ces données faisaient l'objet d'une modification (update ou delete) ces verrous devenaient exclusifs (section 5.5.2). Cette transformation peut entrer en conflit avec d'autres verrous partagés. Pour éviter cette situation, on demandera le verrouillage exclusif des données dès leur consultation :

```
select CAT,COMPTE from CLIENT where NCLI = 'C400' for update;
```

8.13.5 Mise à jour et contraintes d'unicité

L'exécution des requêtes insert et update est susceptible d'entraîner une violation de contraintes d'unicité. Dans un tel cas, le SGBD va refuser l'opération. Le cas d'un identifiant secondaire (prédicat unique) est un peu plus délicat si certaines de ses colonnes sont *facultatives*. Le standard SQL2 et les suivants stipulent que la contrainte d'unicité s'applique seulement parmi les lignes pour lesquelles **toutes** les colonnes ont une valeur. Deux lignes ayant pour cet identifiant les mêmes valeurs peuvent coexister pour autant qu'au moins une de ces valeurs soit null.

Comme nous l'avons vu à la section 2.6, le problème est que certains SGBD (SQL Server par exemple) ne se conforment pas à cette interprétation, considérant que deux valeurs null sont égales, ce qui est une erreur grave (voir section 9.12). Il n'est donc pas possible d'enregistrer deux lignes dont les valeurs de l'identifiant sont null. Il est possible de pallier cette faiblesse par un *index identifiant partiel* (section 6.15) bien que cette construction soit jugée obsolète aujourd'hui.

8.13.6 Mise à jour et contraintes référentielles

Une requête de modification du contenu de la base de données (`insert`, `delete`, `update`, etc.) 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 2.10.3 et sa traduction en SQL à la section 6.9.

8.13.7 Déterminisme des requêtes SQL

Une même requête exécutée sur une même base de données produit-elle toujours le même résultat ? La question a de quoi intriguer puisque la réponse devrait être résolument positive, à défaut de quoi nous serions dans une totale insécurité. La réalité nous oblige à nuancer cette certitude. Les modes comportementaux liés à la modification des données peuvent en effet poser des problèmes délicats auxquels on devra être attentif. Nous en mentionnerons trois.

a) Interaction entre colonnes dans un `update`

L'effet d'une modification faisant intervenir les mêmes colonnes à gauche et à droite des clauses `set` dépend malheureusement d'options d'implémentation propres au SGBD. Aussi, la requête suivante¹⁷ aura-t-elle un effet différent selon le SGBD :

```
update CLIENT set ADRESSE = LOCALITE,  
                LOCALITE = ADRESSE;
```

En MySQL et en InterBase, cette requête copiera, pour chaque ligne, la valeur de `LOCALITE` dans la colonne `ADRESSE`, détruisant ainsi les valeurs de cette dernière. En Oracle, PostgreSQL et SQLite, elle échangera les valeurs de ces deux colonnes pour chaque ligne. On appréciera la différence !

b) `delete` dans une structure cyclique

Reprenons, en la complétant, la définition de la table `PERSONNE` de la figure 8.1 :

```
create table PERSONNE ( NPERS char(4) not null primary key,  
                        NOM    char(25) not null,  
                        RESPONSABLE char (4),  
                        foreign key (RESPONSABLE) references PERSONNE  
                        on delete no action);
```

Imaginons ensuite les deux requêtes suivantes, exécutées dans cet ordre :

```
delete from PERSONNE where NPERS = 'p2';  
delete from PERSONNE where NPERS = 'p8';
```

17. On se gardera bien de considérer que cette requête est stupide. Elle est au contraire très utile dans les processus de nettoyage de données lorsque l'encodeur risque de se tromper de champs dans les écrans de saisie.

Il est clair que la première va échouer alors que la seconde va effectivement supprimer la ligne "p8". Si on permute l'ordre d'exécution, les deux lignes p2 et p8 seront supprimées. Regroupons à présent ces requêtes sous la forme suivante :

```
delete from PERSONNE where NPERS in ('p2','p8');
```

Si les lignes sont traitées dans l'ordre (p2, p8), alors l'exécution échouera et aucune ligne ne sera supprimée. En revanche, si elle sont prises en compte dans l'ordre (p8, p2), les deux lignes seront supprimées. Pire, si le SGBD décidait de parcourir séquentiellement les lignes de la table pour supprimer celles qui satisfont la condition de sélection, alors le résultat dépendra de l'ordre dans lequel les lignes seront lues, le plus souvent celui dans lequel elles sont stockées et qui nous est inconnu. Une telle insécurité est bien sûr dommageable. Qu'en est-il en pratique ? Considérons la requête suivante, où C représente une condition quelconque :

```
delete from PERSONNE where C;
```

Il existe trois stratégies d'exécution de cette requête : *séquentielle*, *snapshot* et *point fixe*. Appelons PC l'ensemble des lignes vérifiant la condition C. La **stratégie séquentielle** accède aux lignes de PC de manière séquentielle et les supprime au fur et à mesure. Si une ligne de PC, au moment où on y accède, possède encore une ligne dépendante, alors toute l'opération est annulée. Le résultat dépend donc de l'ordre de parcours des lignes. Cette approche *non déterministe* très dangereuse est implémentée dans InterBase, FireBird et MySQL. La **stratégie snapshot** calcule à l'avance l'ensemble PC. Si une seule des lignes de cet ensemble possède une ligne dépendante, alors toute l'opération est annulée. Cette approche *déterministe* n'a pas été observée dans les SGBD examinés du moins pour cette requête. La **stratégie point fixe** consiste à parcourir les lignes de PC, supprimant celles qui n'ont pas de lignes dépendantes, puis à réitérer le processus jusqu'à ce qu'aucune ligne ne puisse plus être supprimée. Cette approche *déterministe* est implémentée dans Oracle, DB2, SQL Server et PostgreSQL.

c) Interférence entre les modes on delete de clés étrangères

Considérons quatre tables A, B, C et D. La table B possède une clé étrangère (RA) vers A, la table C une clé étrangère (RA) vers A, et la table D les clés étrangères (RB) vers B et (RC) vers C. Ces tables, en résumé, sont déclarées comme suit :

```
create table A(IA primary key, ...);
create table B(IB primary key, RA, ...,
  foreign key (RA) references A on delete cascade);
create table C(IC primary key, RA, ...,
  foreign key (RA) references A on delete cascade);
create table D(ID primary key, RB, RC, ...,
  foreign key (RB) references B on delete cascade,
  foreign key (RC) references C on delete no action);
```

Conformément aux modes on delete cascade, la suppression d'une ligne a de A entraîne la suppression de toutes les lignes de B dépendantes de a et la suppression

de toutes les lignes de C dépendantes de **a** pour lesquelles il n'existe pas de lignes de D dépendantes (on delete no action). La suppression en cascade des lignes de B provoque la suppression des lignes de D dépendantes.

Imaginons à présent une ligne **d** de D dépendant transitivement de **a** à la fois via **b** de B et via **c** de C; en outre, **d** est la seule ligne dépendant de **c**¹⁸. Suite à la suppression de la ligne **a**, le SGBD peut poursuivre par la suppression de **b** d'abord ou par celle de **c** d'abord. Dans le premier cas, les quatre lignes **a**, **b**, **c** et **d** seront supprimées tandis que dans le second cas aucune ligne ne sera supprimée. Il est donc impossible de prévoir l'effet de la suppression de **a** !

d) **Constructions SQL non déterministes**

Les clauses `offset` et `limit` décrites à la section 8.11, qui sélectionnent les lignes selon leur rang peuvent produire un résultat aléatoire si le critère de tri n'est pas identifiant.

La fonction agrégative `group_concat`, qui sera étudiée à la section 9.6.1, construit des listes dont l'ordre est aléatoire si la clause `order by` de la fonction n'a pas été spécifiée. Tel est le cas de la version incomplète offerte par SQLite par exemple.

e) **Réduction de l'indétermination**

Lorsqu'un SGBD exécute une action de manière inadéquate, on proposera de forcer le comportement désiré par exemple en stockant un instantané des données avant leur modification dans une table temporaire et en exécutant les opérations de modification à partir de cette table.

8.14 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 non vide 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.

- *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`).
- *Suppression d'une colonne.* Cette colonne ne peut intervenir dans la composition d'un identifiant ou d'une clé étrangère. Si nécessaire, ces objets dépendants doivent d'abord être supprimés.

18. Ce cas de figure est très fréquent. Remplaçons par exemple A par CLIENT, B par COMMANDE, C par FACTURE et D par EXPEDITION. Il est évident que les expéditions facturées à un client sont celles que ce dernier a commandées.

- *Suppression d'une table.* La table à supprimer ne peut être référencée par une clé étrangère ou apparaître dans une vue. Si nécessaire, ces objets dépendants doivent d'abord être supprimés.
- *Ajout d'un identifiant.* Si la table n'est pas vide, les lignes doivent respecter la contrainte d'unicité.
- *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.
- *Ajout d'une clé étrangère.* Si la table n'est pas vide, les lignes doivent respecter la contrainte référentielle.
- *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.

Lors de la suppression d'un objet par une instruction **drop** table/column/view/constraint/index, il est possible de préciser le comportement du SGBD vis-à-vis des objets dépendants. En ajoutant la clause `no action` ou `cascade`, on impose le mode *blocage* ou *propagation* similaire à celui des clés étrangères, mais au niveau des objets du schéma. L'opération ci-dessus supprimera la table COMMANDE, mais aussi la clé étrangère de DETAIL vers COMMANDE et toutes les vues impliquant la table COMMANDE :

```
drop table COMMANDE cascade;
```

8.15 QUE RETENIR ?

Le langage DML (*Data Manipulation Language*) comprend deux classes de requêtes : extraction de données et modification de données.

La forme la plus simple de l'**extraction de données** obéit à la structure `select-from-where`, ou SFW, qui spécifie la table ou les tables dont les données sont extraites (clause `from`), la condition, plus ou moins complexe, à laquelle les lignes doivent satisfaire (clause `where`) et la liste des données à extraire de ces lignes (clause `select`). L'extraction de plusieurs tables s'appuie sur une **jointure**, définie par une *condition de jointure*, spécifiant la règle selon laquelle les lignes des tables sont assemblées.

SQL fournit un registre très vaste de conditions de sélection de lignes (clause `where`). On signalera en particulier les **sous-requêtes**, définissant des associations entre les lignes à sélectionner et d'autres ensembles de lignes. Le résultat d'une requête d'extraction étant une table, les **opérateurs ensemblistes** classiques peuvent être utilisés : *union*, *intersection* et *différence*. L'opérateur de **groupement** permet de constituer des groupes de lignes (clause `group by`), qui peuvent faire l'objet d'une sélection (clause `having`) et d'une extraction de valeurs agrégées.

Les **requêtes de modification** de données opèrent sur les ensembles de lignes : **insertion** de lignes (`insert`), **suppression** de lignes sélectionnées (`delete`) et **modification** de valeurs de colonnes de lignes sélectionnées (`update`). L'exécution de ces

requêtes garantit la préservation de l'**intégrité** des données, exprimées par les contraintes d'intégrité. Le comportement **non déterministe** de certains SGBD doit inciter à la prudence, notamment dans le cas de structures de données cycliques.

8.16 POUR EN SAVOIR PLUS

SQL est un langage extrêmement riche, puissant et évolutif dont les chapitres précédents ont brossé un panorama relativement large. Le chapitre 9 apportera au lecteur des compléments d'information sur le langage SQL, notamment dans sa version SQL3. Le web regorge de tutoriels, blogs et forums dédiés à SQL. Citons par exemple stackoverflow.com, www.developpez.net, sqlpro.developpez.com, www.commentcamarche.net. Pour le reste, le lecteur sera invité à se tourner vers la littérature spécialisée fournie par les éditeurs de SGBD.

8.17 EXERCICES

Exprimer les requêtes suivantes en SQL.

- 8.1 Calculer le montant de chaque détail de commande du client 'C400'.
- 8.2 Calculer le montant commandé des produits en sapin.
- 8.3 Afficher le total et la moyenne des comptes des clients, ainsi que le nombre de clients, selon chacune des classifications suivantes : (1) par catégorie, (2) par localité, (3) par catégorie dans chaque localité.
- 8.4 Combien y a-t-il de commandes spécifiant un (ou plusieurs) produit(s) en acier (on proposera une solution différente de celle du chapitre 7) ?
- 8.5 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.
- 8.6 Annuler les comptes négatifs des clients de catégorie C1.
- 8.7 Compléter le fragment suivant de manière à former une requête valide

```
select CAT, NPRO, sum(QCOM*PRIX) from ...
```
- 8.8 En considérant le schéma 8.5, 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.

- 8.9 Calculer le montant dû par chaque client. Dans ce calcul, on ne prend en compte que le montant des commandes.
- 8.10 Afficher pour chaque localité, les libellés des produits qui y sont commandés.
- 8.11 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. Éviter les valeurs null.
- 8.12 Considérant le schéma 8.5, 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.
- 8.13 En considérant le schéma de la figure 8.8, calculer pour chaque ville, le prix moyen de chaque produit.
- 8.14 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 :* vue ou comptage dans le from.
- 8.15 Donner pour chaque produit la liste des localités dans lesquelles ce produit est commandé en plus de 100 unités (= au total pour la localité).

Exercices corrigés

• Exercice 8.1

```
select D.NCOM,P.NPRO,QCOM*PRIX as MONTANT
from   COMMANDE M, DETAIL D, PRODUIT P
where  D.NCOM = M.NCOM and D.NPRO = P.NPRO
and    NCLI = 'C400';
```

• Exercice 8.2

```
select sum(QCOM*PRIX) as MONTANT from DETAIL D,PRODUIT P
where  D.NPRO = P.NPRO and P.LIBELLE like '%SAPIN%';
```

• Exercice 8.3 (3)

```
select LOCALITE,CAT,sum(COMPTE),avg(COMPTE),count(*)
from   CLIENT group by LOCALITE, CAT;
```

• Exercice 8.4

```
select count(distinct M.NCOM)
from   COMMANDE M,DETAIL D,PRODUIT P
where  M.NCOM = D.NCOM and D.NPRO = P.NPRO
and    LIBELLE like '%ACIER%';
```

- **Exercice 8.5**

```
create table DETAIL_COM(NCOM ...,DATECOM ...,MONTANT ...);
insert into DETAIL_COM(NCOM,DATECOM,QCOM,NPRO,PRIX,MONTANT)
select M.NCOM,DATECOM,QCOM,P.NPRO,PRIX,QCOM*PRIX
from COMMANDE M,DETAIL D,PRODUIT P
where M.NCOM = D.NCOM and D.NPRO = P.NPRO;
```

- **Exercice 8.6**

```
update CLIENT set COMPTE = 0
where COMPTE < 0 and CAT = 'C1';
```

- **Exercice 8.7**

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

- **Exercice 8.8**

```
select NPRO
from PRODUIT
where NPRO not in (select COMPOSE from COMPOSITION);

select NPRO
from PRODUIT
where NPRO not in (select COMPOSANT from COMPOSITION);

select NPRO
from PRODUIT
where NPRO in (select COMPOSE from COMPOSITION)
and NPRO in (select COMPOSANT from COMPOSITION);

select PH.NPRO,sum(QTE*PB.PRIX_U),sum(QTE*PB.POIDS_U)
from PRODUIT PH,COMPOSITION C,PRODUIT PB
where PH.NPRO = C.COMPOSE and C.COMPOSANT = PB.NPRO
and not exists (select * from COMPOSITION CC,PRODUIT BB
where CC.COMPOSANT = BB.NPRO and CC.COMPOSE = PH.NPRO
and (BB.PRIX_U is null or BB.POIDS_U is null))
group by PH.NPRO;
```

- **Exercice 8.9**

```
select NCLI,sum(QCOM*PRIX)
from COMMANDE M,DETAIL D,PRODUIT P
where M.NCOM = D.NCOM and D.NPRO = P.NPRO group by NCLI
union
select NCLI,0 from CLIENT C
where not exists (select * from COMMANDE
where NCLI = C.NCLI);
```

- **Exercice 8.10**

```
select LOCALITE,LIBELLE
from CLIENT C,COMMANDE M,DETAIL D,PRODUIT P
where C.NCLI = M.NCLI and M.NCOM = D.NCOM
```

```
and      D.NPRO = P.NPRO
group by LOCALITE,LIBELLE order by LOCALITE,LIBELLE;
```

- **Exercice 8.11**

```
select distinct L.LOCALITE,C.CAT from CLIENT L,CLIENT C
where C.CAT is not null and not exists
      (select * from CLIENT where LOCALITE = L.LOCALITE
       and CAT = C.CAT);
```

- **Exercice 8.12**

```
update PRODUIT PH
set  PRIX_U = (select sum(QTE*PB.PRIX_U)
              from  COMPOSITION C,PRODUIT PB
              where  PH.NPRO = C.COMPOSE
              and    C.COMPOSANT = PB.NPRO)
where PRIX_U is null
and   not exists (select * from COMPOSITION CC,PRODUIT BB
                 where  CC.COMPOSANT = BB.NPRO
                 and    CC.COMPOSE = PH.NPRO
                 and    BB.PRIX_U is null);
```

- **Exercice 8.13**

```
select VILLE,PRODUIT,avg(PRIX) from VENTE V,IMPLANTAION I
where V.CHAINE = I.CHAINE group by VILLE,PRODUIT;
```

- **Exercice 8.14**

```
create view NOMBRE(NCOM,NBRE) as
  select NCOM,count(*) from DETAIL group by NCOM;
select avg(NBRE) from NOMBRE;
```

ou, sous une forme plus concise (section 8.10.2) :

```
select avg(NBRE)
from   (select count(*) as NBRE from DETAIL group by NCOM);
```

- **Exercice 8.15**

```
select D.NPRO,LOCALITE,sum(QCOM)
from   CLIENT C,COMMANDE M,DETAIL D
where  C.NCLI = M.NCLI and M.NCOM = D.NCOM
group by D.NPRO, LOCALITE having sum(QCOM) > 100;
```

Des exercices et solutions additionnels sont disponibles à l'annexe A8. On y trouvera également des compléments à ce chapitre, en particulier des scripts SQLfast.

Chapitre 9

Le langage SQL avancé

Ce chapitre est destiné au lecteur désireux d'aller plus loin dans la maîtrise des fonctions des bases de données et du langage SQL, et plus particulièrement du standard SQL3. On y décrit successivement les fonctions de contrôle d'accès et les vues, les requêtes récursives, les extensions objet, les mécanismes avancés tels que les prédicats, les procédures SQL et les déclencheurs, le concept de catalogue et l'interface SQL avec les programmes d'application. La question de l'information incomplète est abordée dans le cadre du langage SQL.

9.1 INTRODUCTION

Les chapitres 6, 7 et 8 présentent les fondements du langage SQL. Ceux-ci seront largement suffisants dans la plupart des situations. SQL offre cependant bien d'autres fonctions et services dont nous présenterons et discuterons certains parmi les plus utiles. Nous examinerons en particulier le *contrôle d'accès*, les *vues* SQL, le *catalogue* et les interfaces entre les programmes et les bases de données. Nous étudierons en outre les requêtes *récursives*, les *extensions objet*, les *prédicats*, les *procédures SQL* et les *déclencheurs*. Nous dirons enfin quelques mots sur un problème complexe, celui de l'*information incomplète*.

9.2 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 *privilèges*. SQL adopte les modèles de contrôle d'accès *discretionnaire* et à base de rôles décrit dans la section 5.8.

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 de 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. 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 autorise 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;
```

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 A9.1 à l'annexe A9.

c) Privilèges sur un programme

Ce privilège permet à un utilisateur d'exécuter un programme d'application ou une procédure SQL qui manipule le contenu d'une base de données. Deux exemples :

```
grant run
on    SUP_DETAIL
to    public;
```

```
grant run
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
from  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, ayant reçu un `grant option`, avait déjà transmis ce privilège ? Le comportement est défini par les clauses `cascade` (les privilèges transmis sont aussi retirés) et `restrict` (le privilège d'origine n'est pas retiré), tout comme pour l'instruction `drop`. On trouvera dans [Melton, 2002] une présentation intuitive des principales règles de propagation des privilèges et de leur retrait.

f) Les rôles

La gestion des privilèges peut s'avérer très complexe pour une large communauté d'utilisateurs. On utilisera alors le concept intermédiaire de *rôle*. Un rôle représente une classe générale d'utilisateurs. On attribue et retire des privilèges à un rôle tout comme on le fait pour un utilisateur. On peut ensuite assigner à un utilisateur un ou plusieurs rôles. L'utilisateur hérite des privilèges de tous les rôles qui lui ont été assignés. Dans les exemples ci-après, on crée le rôle `CONSULTANT`, auquel on attribue un ensemble de privilèges, puis on l'assigne à l'utilisateur `P_MERCIER`.

```
create role CONSULTANT;
grant select on CLIENT to CONSULTANT;
```

```
grant update (ADRESSE, LOCALITE) on CLIENT to CONSULTANT;  
grant CONSULTANT to P_MERCIER;
```

Les opérations de réduction et de retrait suivent la syntaxe habituelle :

```
revoke select (LOCALITE) from CONSULTANT;  
revoke CONSULTANT from P_MERCIER;  
drop role CONSULTANT;
```

9.3 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.

9.3.1 Principes et objectifs 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 de base (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.

9.3.2 Définition et utilisation d'une vue

Une vue est définie par une requête SQL qui spécifie son nom, (facultativement) ceux de ses colonnes et la requête SFW qui calcule ses lignes :

```
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 trois requêtes suivantes sont dès lors tout à fait valides :

```
select NOMCLI, NCOM, DATECOM  
from COM_COMPLETE  
where LOC = 'Toulouse';  
  
select distinct 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'exécute². 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    LOCALITE = 'Toulouse';
```

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. On peut ajouter à cette instruction les clauses `cascade` ou `restrict` comme pour les autres formes de `drop`.

9.3.3 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 suivante pourrait convenir à un analyste d'un département de marketing qui étudie la répartition géographique des commandes de produits :

```
create view HABITUDE_ACHAT (LOCALITE, NPRO, MONTANT)
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 CLI_TOULOUSE (section 8.13.1) avec une vue de même définition.

9.3.4 Les vues comme mécanisme de contrôle d'accès

Les vues, comme toutes les tables, peuvent être soumises à un contrôle d'accès. Étant donné une classe d'utilisateurs, on détermine à quelles données ceux-ci ont

2. La réalité est un peu plus complexe, mais ce modèle est suffisant pour nos besoins.

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. Attention cependant aux possibilités d'*inférence statistique* décrite à la section 5.8. 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;
grant select on ANALYSE to MARKETING;
revoke select on CLIENT from MARKETING;
```

9.3.5 Les vues comme mécanisme d'évolution de la base de données

Lorsque le schéma d'une base de données évolue, certaines requêtes ne sont plus valides, notamment dans les programmes d'application. Les vues permettent de protéger les utilisateurs et les programmes contre l'effet de certaines de ces modifications. Une vue permet en effet de restaurer la structure précédente des données de sorte que ses utilisateurs ne sont pas affectés par les modifications des structures.

Imaginons que la table CLIENT soit désormais décomposée en deux nouvelles tables, l'une qui reprend les données signalétiques CLIENT_SIG(NCLI, NOM, ADRESSE, LOCALITE) et la seconde partie qui ne reprend que les données commerciales CLIENT_COM(NCLI, CAT, COMPTE), la colonne NCLI étant commune. Une vue définie comme la jointure de ces deux tables sur NCLI reconstitue la table CLIENT d'origine, certes désormais virtuelle, mais tout à fait opérationnelle.

9.3.6 Les vues comme aide à l'expression de requêtes complexes

Les vues permettent également l'expression de requêtes complexes délicates ou impossibles à rédiger directement sous la forme SFW en SQL. Par exemple, le calcul de la *valeur globale des stocks, déduction faite des quantités commandées*, peut s'obtenir en deux étapes comme suit (on ignore les produits non commandés) :

```
create view VAL_STOCK_ACTU (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;3

select sum(VALEUR) from VAL_STOCK_ACTU;
```

3. On remarque la présence de QSTOCK et PRIX dans le critère de groupement.

9.3.7 Mise à jour des données via une vue

On sait qu'une vue est traitée 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 particulier, la vue doit reprendre les données qui permettent d'identifier sans ambiguïté les lignes à modifier. Que signifierait par exemple la modification de LOCALITE d'une ligne de la vue HABITUDE_ACHAT construite ci-dessus ? On conçoit aisément qu'on ne puisse supprimer une ligne d'une table dont l'identifiant primaire ne serait pas repris dans la vue, ou insérer une ligne via une vue qui ne reprendrait pas certaines colonnes obligatoires⁴.

Les contraintes qui rendent une vue *modifiable*, c'est-à-dire pouvant faire l'objet d'insert, delete, update, sont d'ordre logique, mais s'y ajoutent le plus souvent des restrictions non nécessaires imposées, par (excès de) prudence, par les SGBD. Les conditions de modifiabilité sont généralement les suivantes :

- 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 des tables de base susceptibles d'être modifiée;
- si la vue est définie sur une autre vue, celle-ci doit être elle-même modifiable.
- pas d'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 (section 8.8.4/b) peuvent faire l'objet de modifications. 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 modifiable grâce à la clause additionnelle `with check option`.⁵ 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 NCLI, NOM, ADRESSE, LOCALITE, CAT, COMPTE
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 :

4. À moins que celles-ci ne disposent d'une valeur par défaut, ou qu'un *trigger before* (section 9.9) ne garnisse ces colonnes.

5. Disponible dans la plupart des SGBD. PostgreSQL et SQLite sont des exceptions.

```
insert into CLI
values ('B313', 'DURAND', 'place Monge', 'Mons', 'D7', 0);
```

Concernant la mise à jour d'une vue, signalons encore que les déclencheurs *instead of*, qui seront étudiés plus loin dans ce chapitre, permettront de contourner les limitations des vues modifiables décrites ci-avant.

9.4 LES COMMON TABLE EXPRESSIONS (CTE)

Nous avons vu à la section 9.3.6 qu'une vue pouvait être utilisée pour simplifier l'expression d'une requête complexe en la décomposant en fragments plus simples à rédiger et à lire. Si une vue n'a que cette seule utilité, il est conseillé d'utiliser un autre mécanisme, plus léger, appelé *common table expression*, ou CTE. Il s'agit d'une vue temporaire, qui n'existe que pour la requête *select* courante⁶. La requête suivante est une reformulation de la séquence 9.3.6. La CTE `VAL_STOCK_ACTU [1]` est une vue temporaire exploitée par la requête principale [2].

```
with VAL_STOCK_ACTU (STOCK, VALEUR)                                [1]
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_ACTU;                             [2]
```

La forme générale d'une CTE est la suivante :

```
with T1 as (E1), T2 as (E2), ... R;
```

R est une requête *select* de forme quelconque utilisant les vues temporaires (CTE) **T1**, **T2**, etc., préalablement définies par les requêtes *select* **E1**, **E2**, etc.

9.5 LES REQUÊTES RÉCURSIVES

Le mécanisme des CTE serait de peu d'intérêt s'il n'en existait une extension particulièrement intéressante, la **CTE récursive**. Sa forme la plus simple est la suivante :

```
with recursive T as (E0 union all Er) R;
```

où **T** est une CTE récursive définie par `(E0 union all Er)`, **E0** est la requête d'initialisation de **T**, **Er** la requête récursive proprement dite de **T** et **R** la requête principale. Cette forme étant un peu plus délicate que les requêtes classiques, nous la présenterons d'abord sur un exemple très simple.

On se propose d'afficher la liste des 5 premiers nombres entiers accompagnés chacun de leur carré. La requête ci-après définit cette liste sous forme de la CTE **CARRES**, dotée des colonnes **N** et **Carre** [1], et construite par l'union de deux tables.

6. Une forme similaire existe pour les trois requêtes de modification.

La première est une table statique [2] qui initialise CARRES en y introduisant le premier couple (1,1). La deuxième table est produite par une requête `select` récursive [3] qui décrit le calcul des données suivantes de CARRES. Cette requête va être exécutée de manière itérative jusqu'à ce qu'il n'y ait plus de lignes à ajouter.

```

with recursive CARRES(N,Carre)           [1]
as (values (1,1))                        [2]
    union all
    select N+1, (N+1) * (N+1)            [3]
    from   CARRES where N < 5)           [3]
select * from CARRES order by N;         [4]

```

Le résultat est, comme attendu, le suivant :

N	Carre
1	1
2	4
3	9
4	16
5	25

Examinons plus en détail la manière dont ce résultat est obtenu.⁷

- Pour exécuter la requête principale [4], il faut au préalable avoir évalué la CTE CARRES. On examine donc sa mécanique d'exécution.
- Après exécution de la requête d'initialisation [2], la table CARRE se trouve dans l'état de la figure 9.1 (gauche).
- Ensuite, la requête récursive [3] est exécutée. À partir de la ligne (1,1) qui vient d'être introduite et qui satisfait la condition $N < 5$, elle construit la ligne $(N+1, (N+1) * (N+1))$, soit (2,4), qui est ajoutée à CARRES (figure 9.1, centre).
- La requête récursive est à nouveau exécutée, mais de manière particulière : elle n'examine dans la table CARRES que les lignes *qui n'ont pas encore été exploitées*. La ligne (1,1) est donc ignorée tandis que la ligne (2,4), qui n'a pas encore été exploitée et qui satisfait la condition $N < 5$, est extraite et traitée. Elle permet de produire la ligne (3,9) qui est ajoutée à CARRES (figure 9.1, droite).
- Lors de la troisième exécution de la requête récursive, les lignes (1,1) et (2,4) sont ignorées et seule la dernière ligne (3,9) est lue. La ligne (4,16) est ajoutée.
- Les itérations se terminent lorsqu'il n'y a plus de ligne candidate qui satisfait la condition $N < 5$, qui constitue ici la condition d'arrêt.

7. La page explicative de la requête `with recursive` de SQLite [http://www.sqlite.org/lang_with.html] propose une description très intuitive de la procédure d'exécution : 1. *Run the initial-select and add the results to a queue.* 2. *While the queue is not empty:* (a) *Extract a single row from the queue.* (b) *Insert that single row into the recursive table.* (c) *Pretend that the single row just extracted is the only row in the recursive table and run the recursive-select, adding all results to the queue.*

- La requête principale peut enfin extraire les lignes de CARRE, qui seront traitées de manière appropriée (voir section 9.11 par exemple).

N	Carre	N	Carre	N	Carre
1	1	1	1	1	1
		2	4	2	4
				3	9

Figure 9.1 - Les trois premières étapes du remplissage de CARRES

Dans cet exemple, les lignes générées successivement sont distinctes par construction, de sorte que l'opérateur *union all* ne produit pas de doublons. Dans le cas contraire et si les doublons sont jugés indésirables, on utilisera l'opérateur *union* si le SGBD l'autorise (mais il faut savoir qu'il est coûteux) ou on éliminera les lignes en double par le modifieur *distinct* dans la requête principale.

Nous pouvons à présent aborder un problème plus intéressant, par exemple celui que nous avons abandonné à la section 8.7 : extraire les données de *tous les subordonnés, directs et indirects, d'une personne*, selon la table de la figure 8.2.

Par analogie avec le calcul des carrés, la requête, appliquée par exemple à la personne **p4**, devrait ressembler à ceci :

```
with recursive ORGA (NIVEAU, NPERS)
as (données de la personne 'p4'
    union all
    données des subordonnés des dernières personnes introduites
)
select NIVEAU, P.NPERS, NOM, RESPONSABLE
from ORGA O, PERSONNE P where O.NPERS = P.NPERS;
```

La CTE ORGA constitue une liste de couples (NIVEAU, NPERS), chacun d'eux désignant soit **p4**, soit un subordonné de **p4**, accompagné de son niveau dans la hiérarchie : 1 pour **p4**, 2 pour les subordonnés directs de **p4**, 3 pour les subordonnés directs des subordonnés directs de **p4**, et ainsi de suite. La jointure de la requête principale ajoute le nom et le responsable de chaque personne extraite. Le résultat attendu est celui-ci :

NIVEAU	NPERS	NOM	RESPONSABLE
1	p4	Dupont	p1
2	p5	Verger	p4
2	p6	Dupont	p4
3	p7	Dermiez	p6

Poursuivons l'analyse de la CTE, dont le corps est constituée de l'union de deux requêtes. La requête d'initialisation range dans ORGA les données de **p4** :

```
values (1, 'p4')
```

La requête récursive extrait les nouveaux subordonnés directs, c'est-à-dire les personnes (alias P) dont le responsable (alias O) a été ajouté à ORGA mais n'a pas encore été exploité :

```
select O.NIVEAU + 1, P.NPERS
from   ORGA O, PERSONNE P
where  O.NPERS = P.RESPONSABLE
```

En assemblant ces composants, on construit la requête récursive complète :

```
with recursive ORGA(NIVEAU, NPERS)
as (values (1, 'p4')
     union all
     select O.NIVEAU + 1, P.NPERS
     from   ORGA O, PERSONNE P
     where  O.NPERS = P.RESPONSABLE)
select NIVEAU, O.NPERS, NOM, RESPONSABLE
from   ORGA O, PERSONNE P where O.NPERS = P.NPERS;
```

Remarques

- La requête E0 peut être de complexité quelconque mais ne peut citer T. La requête Er doit citer T dans sa clause from. La requête R doit aussi citer la table T. Pour le reste, R peut inclure des jointures avec d'autres tables, des fonctions agrégatives, des conditions de sélection, des groupements et une clause order by.
- La forme de la requête Er et les données de base (PERSONNE) peuvent conduire à un processus sans fin. Tel serait le cas, par exemple, si les données de la table PERSONNE constituaient un circuit et non des arbres. Lorsque la chose est possible, on inclura dans la clause where de Er une condition prévenant ce type d'accident. Certains SGBD prévoient des protections contrôlant par exemple le nombre d'appels en cascade dans les évaluations de Er ou excluent les lignes déjà visitées. Dans la requête ci-dessus, une simple condition limitant les valeurs de Niveau au nombre de personnes permettrait de sécuriser le processus.
- Nous verrons à la section 9.9 que certains problèmes tels que celui des subordonnés traité ci-dessus peuvent aussi être résolus à l'aide de déclencheurs récursifs.

9.6 LES EXTENSIONS OBJET DE SQL3

L'apport le plus spectaculaire de SQL3 est certainement celui des constructions dites *orientées objet*, ce qui vaut au modèle ainsi décrit l'appellation *relationnel objet*. On dira par exemple que Oracle 18 ou DB2 v12 sont, parmi bien d'autres, des *SGBD relationnels objet*. Nous en ferons une description sélective permettant au lecteur de comprendre et d'apprécier cet apport.

Il faut reconnaître que les extensions enrichissent le modèle et le langage SQL mais rendent ceux-ci plus complexes, d'autant plus qu'elles ne sont pas toujours

traduites conformément au standard par tous les éditeurs de SGBD. Au lecteur de juger si leur adoption lui est utile.

Nous examinerons successivement les types de données complexes, les types définis par l'utilisateur, les tables typées, les hiérarchies de types et de tables, et les références entre tables.

9.6.1 Types de données complexes (**row** et **array**)

En SQL2, les valeurs d'une colonne sont dites élémentaires puisqu'elles ne peuvent être ni composées ni multiples. Tel est le cas des types de base décrits au chapitre 6. Ce langage accepte que la valeur d'une colonne d'une ligne soit un nombre, une date, une chaîne de caractères, mais guère plus. Les constructeurs **row** et **array** permettent de définir des types plus complexes.

*Le constructeur **row***

Le constructeur **row**⁸ définit un type de données composé de plusieurs types de données élémentaires ou composées, ce qui nous permet d'écrire :

```
create table CLIENT (
    NCLI      char(10) not null primary key,
    NOM       char(32) not null,
    ADRESSE   row(RUE char(30), LOCALITE char(60)),
    CAT       char(2));
```

Une ligne de cette table se présenterait comme suit :

```
values ('B332', 'MONTI', row('r. Neuve', 'Genève'), 'B2')
```

Les composants, qu'on appelle *attributs* pour la circonstance, sont désignés dans les requêtes selon la notation pointée classique :

```
select NCLI, NOM, ADRESSE.RUE
from   CLIENT
where  ADRESSE.LOCALITE = 'Poitiers';
```

*Le constructeur **array***

Le constructeur **array** définit un type de données constitué d'un tableau d'éléments à une ou plusieurs dimensions dont on peut spécifier le nombre maximum. La représentation des prénoms des clients (on admet qu'un client possède de 0 à 4 prénoms) et de leurs multiples adresses s'exprimera comme suit :

```
create table CLIENT2 (
    NCLI      char(10) not null primary key,
    NOM       char(32) not null,
    PRENOM    char(15) array(4),
    ADRESSES  row(RUE char(30), LOCALITE char(60)) array(2),
    CAT       char(2));
```

8. Une valeur de ce type est une ligne (*row* en anglais) analogue à un élément d'une table.

Une ligne de CLIENT2 se présenterait ainsi :

```
values ('B332', 'MONTI', array['Alain', 'Jean'], ...)
```

La désignation d'éléments d'un tableau suit la notation indiquée habituelle :

```
select NCLI, NOM, PRENOM[1], PRENOM[2], ADRESSES[1].RUE
from    CLIENT2
where   ADRESSES[1].LOCALITE = 'Poitiers';

update CLIENT2 set PRENOM[1] = 'Anne' where NCLI = 'W702';
```

Les tableaux constituent des constructions complexes à manipuler qu'on utilisera avec prudence. On précise ci-après quelques règles de manipulation.

Un élément non affecté d'un tableau contient null. Une partie (*slice*) d'un tableau est définie par son intervalle d'indices : PRENOM[1:2].

On peut définir un tableau par concaténation : `array['Jean', 'Jules'] || array['Jim']` ou encore `array['Jean', 'Jules'] || 'Jim'`. Des fonctions spécifiques permettent de supprimer des éléments, de rechercher le premier élément non null et de trier les éléments. Pour tester si un élément est présent dans un tableau, on écrira : `'Jules' in PRENOM` ou `'Jules' = any(PRENOM)`.

Des fonctions de conversion permettent de transformer un tableau à partir d'une table (`nest`, `array_agg`) et inversement (`unnest`). Voir section 3.9.

Il faut malheureusement observer que la manière dont les tableaux sont traités varie considérablement d'un SGBD à l'autre.

La norme SQL 2003 introduit le constructeur **multiset**, qui correspond à un multi-ensemble, soit une collection de valeurs non nécessairement distinctes (annexe A24). La taille d'un **multiset** n'est pas limitée. On notera qu'Oracle propose le concept de *nested table*, un type de colonne constitué d'un *ensemble* de lignes.

La fonction `group_concat`

La plupart des versions récentes des SGBD offrent, sous des syntaxes variées,⁹ une fonction agrégative non standard mais particulièrement utile, qui transforme une table en une liste. La requête ci-dessous, rédigée dans la syntaxe de MySQL,

```
select NCOM, count(*) as Nombre,
       group_concat(NPRO order by NPRO separator ',') as Pro
from    DETAIL
group by NCOM;
```

donne, pour chaque valeur de NCOM extraite de la table DETAIL, le nombre de lignes de DETAIL ainsi que la liste des numéros de produits NPRO, ordonnés par valeurs croissantes et séparés par une virgule. Elle produit le résultat suivant :

9. `listagg()` d'Oracle et DB2, `string_agg()` de PostgreSQL et SQL Server, `list()` de Firebird, `group_concat` de MySQL, SPARQL (pour RDF) et SQLite.

NCOM	Nombre	Pro
30178	1	CS464
30179	2	CS262, PA60
30182	1	PA60
30184	2	CS464, PA45
30185	3	CS464, PA60, PS222
30186	1	PA45
30188	4	CS464, PA45, PA60, PH222

La syntaxe complète admet la clause `distinct` qui éliminer les doublons dans la liste résultante.¹⁰ Cette fonction est introduite dans le dernier standard, SQL:2016, sous le nom `list_agg`.

9.6.2 Type défini par l'utilisateur (TDU)

L'instruction `create domain`, plutôt discrète en SQL2, est devenue une des plus importantes en SQL3 sous la forme `create type`. Sa portée a été étendue aux types de données complexes. La table CLIENT précédente peut être définie comme suit :

```
create type Chaine as varchar(60) default '?';
create type Contact as (RUE Chaine, LOCALITE Chaine);

create table CLIENT(
    NCLI      char(10) not null primary key,
    NOM       Chaine not null,
    ADRESSE   Contact not null
    CAT       char(2));
```

Chaine et Contact sont ce qu'on appelle des *types définis par l'utilisateur* ou TDU, par opposition aux types de base prédéfinis tels que `character`, `numeric` ou `date`. Un type peut être supprimé s'il n'est plus utilisé (`drop type`).

9.6.3 Table typée

La portée d'un TDU dépasse celle de la simple définition de colonnes puisqu'il permet également de définir des tables. L'exemple suivant crée un type, puis deux tables de ce type :

```
create type TINDIVIDU as (
    NPERS      char(10),
    NOM        Chaine,
    ADRESSE    Contact)
```

10. Certains SGBD n'offrent qu'une version incomplète de la fonction. Tel est le cas de SQLite (v22), qui permet de spécifier l'unicité (`distinct`) ou le séparateur, mais pas les deux. La clause `order by` est également manquante. Il est à noter que SQLfast fournit la fonction `group_concat2`, équivalente à la version de MySQL. Par ailleurs, cette fonction peut être simulée par une CTE récursive (voir script `group_concat.sql` du chapitre 9).

```
create table EMPLOYE of TINDIVIDU;

create table FOURNISSEUR of TINDIVIDU;
```

EMPLOYE et FOURNISSEUR sont des tables dites *typées* dont chaque colonne dérive d'un attribut du TDU, lui empruntant son nom et son type. À ce stade, ces définitions sont encore incomplètes.

9.6.4 Hiérarchie de types

Les TDU et les tables typées nous font entrer dans un nouveau monde, celui des *objets*. Une ligne d'une table typée est plus qu'une simple ligne, c'est (plus ou moins) un objet au sens de la programmation et de la modélisation orientées objet¹¹. Cette section n'a pas pour ambition de constituer une introduction aux approches objet, mais plutôt d'établir un lien entre ces approches et les bases de données. Nous nous limiterons donc à l'essentiel en supposant que le lecteur est déjà familiarisé avec le concept d'objet¹².

Conformément à ce concept, un TDU peut être défini comme un *sous-type* d'un autre TDU, qui est de ce fait son *surtype*. Par cette relation, le sous-type hérite de tous les attributs de son surtype, mais il est possible de lui ajouter des attributs propres. Un TDU ne peut avoir qu'un seul surtype¹³, de sorte que cette relation définit ce qu'on appelle une hiérarchie simple, qui prend la forme d'arbres.

La figure 9.2 décrit une hiérarchie de cinq types. Intuitivement, le graphe exprime, notamment, que le TDU TCLIENT hérite du TDU TPERSONNE et qu'à ce titre il est formé des attributs hérités NPERS, NOM, ADRESSE, LOCALITE et des attributs propres CAT et COMPTE.

La clause REF est relative à la colonne, encore inconnue à ce stade, qui, dans toute table définie sur ce type, jouera le rôle d'identifiant d'objet. Elle spécifie le mode d'assignation de valeur lors d'une insertion dans la table, ici *automatique*, le système se chargeant de choisir une valeur non encore attribuée (REF is system generated). On peut aussi déclarer que l'identifiant d'objet sera constitué d'une colonne gérée par l'utilisateur et dont on donne le type (REF using char(20)) ou qu'il sera tout simplement égal à l'identifiant primaire (REF is derived). Un surtype peut être déclaré *not instantiable*, ce qui interdit d'insérer directement des lignes dans une table de ce type. On devra procéder via l'insertion de lignes dans une sous-table *instantiable*. On observe qu'un type ne comporte aucune contrainte d'intégrité, qu'il s'agisse d'attributs obligatoires, de contrainte d'unicité ou référentielle ou de prédicat check. Les contraintes sont en fait associées aux tables comme nous allons le voir.

11. Plus précisément, une instance d'un TDU est une valeur et non un objet. Cependant, une ligne d'une table définie sur un TDU va recevoir d'autres composants qui vont lui donner un statut proche des objets.

12. Le concept de hiérarchie de types sera réétudié dans un contexte voisin, celui de la modélisation conceptuelle, au chapitre 16.

13. PostgreSQL constitue une exception : un type peut hériter de plusieurs surtypes.

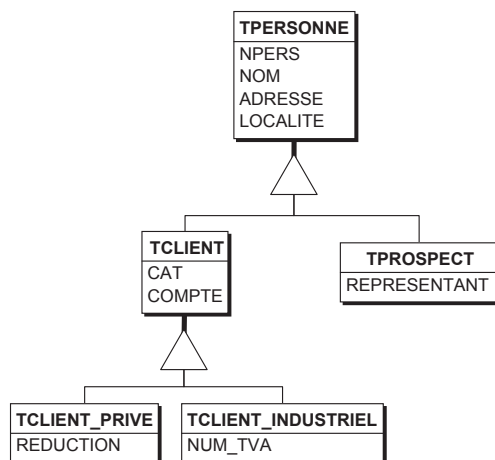


Figure 9.2 - Une hiérarchie de TDU

On exprimera cette hiérarchie comme suit :

```

create type TPERSONNE as (
  NPERS    char(10),
  NOM      Chaîne,
  ADRESSE  Chaîne,
  LOCALITE Chaîne)
REF is system generated;

create type TCLIENT under TPERSONNE as (
  CAT      char(2),
  COMPTE   decimal(9,2));
  
```

9.6.5 Hiérarchie de tables typées

Une table typée est soumise à des exigences particulières. D'une part, outre les colonnes qui dérivent des attributs de son type, elle doit comporter une colonne supplémentaire de type REF (dénomination malheureuse !) qui joue le rôle d'identifiant d'objet. D'autre part, il faut préciser les contraintes auxquelles la table et les colonnes sont soumises, contraintes qui ne peuvent être déclarées dans le type. Illustrons ces principes par la construction de deux tables basées sur les types TPERSONNE et TCLIENT. Leur schéma est représenté à la figure 9.3 (schéma de gauche) ; il est conforme à la hiérarchie de types 9.2. On observe l'apparition d'un identifiant d'objet¹⁴ (IdP), d'un identifiant primaire (NPERS) et de colonnes obligatoires.

14. Curieusement dénommée *self-referencing column*, comme si cette colonne constituait une clé étrangère vers sa propre table, ce qui est mathématiquement exact mais plutôt anecdotique !

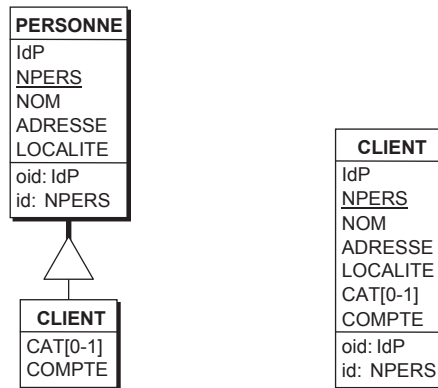


Figure 9.3 - Hiérarchie de tables (gauche)
et composition effective de la table CLIENT (droite)

Le mécanisme d'héritage s'applique également aux nouvelles caractéristiques des tables. C'est ainsi que la table CLIENT hérite d'un identifiant primaire formé de NPERS, que toutes ses colonnes héritées sont obligatoires, et que la colonne propre COMPTE est obligatoire (figure 9.3, droite). Ces tables sont déclarées comme suit :

```

create table PERSONNE of TPERSONNE
(REF is IdP system generated,
 NPERS      with options not null,
 NOM        with options not null,
 ADRESSE    with options not null,
 LOCALITE   with options not null,
 primary key (NPERS));

create table CLIENT of TCLIENT under PERSONNE (
 COMPTE with options not null default 0 check(COMPTE >= 0));
  
```

La clause **REF** introduit la colonne jouant le rôle d'identifiant d'objet. Elle spécifie son nom (IdP) et rappelle le mode d'assignation de valeur spécifié dans le type de la table : automatique (*system generated*), par l'utilisateur (*user generated*) ou copie de l'identifiant primaire (*derived*). La clause *with options* permet d'ajouter des propriétés à une colonne, telle qu'une contrainte ou une valeur par défaut.

Une requête s'adressant à la table CLIENT peut utiliser les colonnes propres et les colonnes héritées. Une requête portant sur une table qui possède des sous-tables est plus délicate. Ainsi, la requête

```

select NPERS, NOM, ADRESSE
from   PERSONNE
where  LOCALITE = 'Poitiers';
  
```

nous donnera bien sûr toutes les lignes de la table PERSONNE qui satisfont la condition de sélection. Parmi ces lignes, nous en trouverons très certainement qui appartiennent à la table CLIENT. En effet, comme nous en discuterons au chapitre 16, il est

entendu que les lignes de CLIENT sont, *dans un certain sens*, assimilées à des lignes de PERSONNE (*tout client est une personne*), enrichies des colonnes CAT et COMPTE. Si nous ne désirons que les lignes de PERSONNE à l'exclusion de celles de ses sous-tables, nous devons utiliser le modifieur `only` :

```
from only (PERSONNE)
```

Par rapport aux relations *is-a* classiques, que nous étudierons à la section 16.9, les structures de tables SQL correspondent aux hiérarchies simples (un seul surtype), disjointes (un objet est d'un seul des sous-types de son surtype) et totale (`not instantiable`) ou non totale (`instantiable`).

9.6.6 Références entre tables

Les clés étrangères constituent en SQL2 le moyen privilégié d'établir des relations entre des lignes de tables. Elles existent également en SQL3, mais s'y ajoute un nouveau mécanisme emprunté des approches objet, la *colonne de référence*.

Une colonne de référence est du type REF et ses valeurs sont empruntées à l'identifiant d'objet d'une ligne d'une autre table typée (ou de la même table). Imaginons une table COMMANDE définie comme suit :

```
create table COMMANDE (  
    NCOM char(10) not null primary key,  
    DATECOM date not null,  
    REFCLI REF(TCLIENT) scope CLIENT not null);
```

La colonne REFCLI de chaque ligne de COMMANDE est une référence vers une ligne de type TCLIENT et plus particulièrement une ligne de la table CLIENT. La clause `scope` peut être omise, auquel cas la valeur de REFCLI peut référencer une ligne de toute table de type TCLIENT.

Une colonne REF peut être soumise à une contrainte référentielle similaire à celle des clés étrangères. On distingue les cas suivants, d'application lors de la création ou de la modification d'une ligne de COMMANDE :

1. REF(TCLIENT) : intégrité référentielle non vérifiée ;
2. REF(TCLIENT) references are not checked : intégrité référentielle non vérifiée ;
3. REF(TCLIENT) references are checked : intégrité référentielle vérifiée ;
4. REF(TCLIENT) scope CLIENT references are not checked : intégrité référentielle non vérifiée ;
5. REF(TCLIENT) scope CLIENT references are checked : intégrité référentielle vérifiée ;
6. REF(TCLIENT) scope CLIENT : intégrité référentielle vérifiée.

Lorsque la vérification de l'intégrité est demandée, on pourra compléter la spécification par l'indication du comportement à adopter lors d'une suppression. Celui-ci est du même type que le *delete mode* des clés étrangères (`no action`, `cascade`, `set`

null, set default). Il n'y a pas d'*update mode* explicite, celui-ci étant inclus dans le comportement lors de l'insertion.

L'accès aux données via une colonne de référence se fait par un mécanisme différent de la jointure propre aux clés étrangères.

```
select NCOM, DATECOM, REFCLI->NPERS, REFCLI->NOM
from   COMMANDE
where  REFCLI->LOCALITE = 'Poitiers';
```

L'expression "REFCLI->" représente la ligne référencée par REFCLI tandis que "REFCLI->NPERS" désigne la valeur de NPERS de la ligne référencée par REFCLI.

9.6.7 Remarques finales

Nous n'avons abordé dans cette section que les principes généraux relatifs aux structures d'objets tels que perçus par SQL. Parmi les concepts additionnels mentionnons en particulier :

1. Les *types distincts* (distinct type) qui sont des TDU simples (analogues aux domaines de SQL2) soumis à des contraintes strictes quant à la comparabilité de leurs valeurs : les valeurs du TDU MONTANT_DOLLAR sont comparables entre elles mais pas avec celles de MONTANT_EURO.
2. Les caractéristiques *not final / final* (accepte ou non des sous-types) ou *encore instantiable / not instantiable* (peut-on ou non créer des instances) des TDU.
3. Conformément à l'approche objet, on peut associer à un TDU des *méthodes* qui permettent, en particulier, de manipuler ses instances de manière appropriée.
4. Les concepts d'encapsulation¹⁵, de polymorphisme et de surcharge sont d'application parmi les TDU d'une même branche d'une hiérarchie.

La manière dont SQL perçoit les objets est qualifiée d'hybride; Elle ne satisfait ni les puristes de l'approche relationnelle ni ceux de l'approche orientée objet. Il est clair que SQL3 résulte d'un compromis pragmatique entre des tendances opposées.

9.7 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 modifica-

15. Laquelle est relativement contradictoire avec la logique du langage SQL, puisque ce dernier est basé sur l'accès direct aux composants des lignes. Par exemple, l'accès à la valeur de LOCALITE de la ligne C de CLIENT peut être demandé de deux manières : classiquement par C.LOCALITE et selon l'approche objet par l'accessor par défaut C.LOCALITE(). La réconciliation est réalisée comme suit : on peut utiliser la forme C.LOCALITE comme raccourci pour C.LOCALITE(). Deux philosophies opposées pour le prix d'une !

tion. 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 le prédicat de table :

```
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 contrainte¹⁶ :

```
alter table CLIENT
add constraint CHK_CAT
check (CAT is null or CAT in ('B1','B2','C1','C2'));
```

Le prédicat suivant garantit que les dates de commande (lorsque l'ensemble des commandes est non vide, d'où le prédicat `is not false`) sont correctement attribuées selon l'ordre chronologique :

```
alter table COMMANDE
add check ((DATECOM >= (select max(DATECOM)
                        from    COMMANDE)
            and DATECOM <= CURRENT_DATE) is not false);
```

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 prédicats précédent et suivant sont donc interdits. Ils sont en revanche autorisés en InterBase et PostgreSQL.

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

16. Aussi valable dans la déclaration de la table : `constraint CHK_CAT check (...)`

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, et constitue ainsi un prédicat de colonne :

```
CAT char(2) check(CAT is null or
                  CAT in ('B1','B2','C1','C2'))
```

Signalons enfin qu'il est possible d'associer une clause check à un domaine :

```
create domain MONTANT integer check(value >= 0);
```

9.8 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.

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.

À 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 supprimée¹⁷ :

```
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
    endif;
end;
```

17. 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.

On pourra l'invoquer par une commande telle que la suivante :

```
call SUP_DETAIL('30182','PA60');
```

9.9 LES DÉCLENCHEURS (*triggers*)

Un déclencheur (*trigger*) est un mécanisme constitué d'une section de code (appelée *action* ou *corps*) 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 généralement 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.

L'exemple suivant est celui d'un déclencheur qui, après que la catégorie d'un client en négatif passe à null, efface toutes ses commandes.

```
create trigger SUP_COM
after update of CAT on CLIENT
for each row
when (new.COMPTE < 0)
begin
  if (old.CAT is not null and new.CAT is null)
  then delete from COMMANDE where NCLI = old.NCLI;
  endif;
end;
```

Ce *trigger*, qui porte le nom de SUP_COM, est défini comme suit :

1. l'action doit être exécutée après l'événement (*after*) ;
2. l'événement déclencheur est la modification de la colonne CAT de la table CLIENT (*update of CAT on CLIENT*) ;
3. la condition et l'action seront appliquées pour chaque ligne ainsi modifiée (*for each row*) ;
4. on exécute l'action si la valeur de COMPTE est inférieure à 0 (*when new.COMPTE < 0*) ;
5. l'action consiste à supprimer les lignes de COMMANDE attachées à la ligne de CLIENT qui est modifiée.

Examinons plus en détail les différents aspects d'un déclencheur.

Le moment de l'exécution du déclencheur

L'action A du déclencheur est exécutée **avant** l'exécution de la modification (before) ou **après** cette exécution (after) ou encore sous certaines conditions **à la place de** cette exécution (instead of). Cette dernière variante est offerte par certains SGBD en compensation des limitations de mise à jour des données via une vue. Il est ainsi possible de préciser ce qu'on entend par insert, update et delete des données d'une vue lorsque ces opérations sont interdites par le SGBD. Certains SGBD autorisent les triggers instead of sur des tables de base.

L'événement déclencheur

L'événement est en principe toute demande de modification des données via une instruction insert, update ou delete. Les modifications de structures sont parfois admises (create, drop, alter) et même une requête select (Informix). À ces événements internes à la base de données pourraient s'ajouter des événements externes tels que l'horloge (dès qu'il est 18h), le calendrier (le lundi à 0h), le démarrage ou l'arrêt d'un programme, la connexion ou la déconnexion d'un utilisateur, la clôture ou l'annulation de la transaction courante, la réception d'un message électronique, etc. Malgré l'intérêt de ces événements, ceux-ci ne sont en général pas pris en compte directement par les SGBD¹⁸. Il est cependant possible de les simuler par des procédures externes à la base de données qui traduisent ces événements sous forme d'insertion d'une ligne dans une table *ad hoc*. Par exemple, lors de la connexion d'un utilisateur, le script de démarrage de sa session inclut l'insertion dans cette table d'une ligne reprenant l'identification de cet utilisateur, la date et l'heure, conduisant au déclenchement d'un trigger after insert.

La portée du déclencheur

Certaines instructions entraînent la modification de plusieurs lignes. Il en est ainsi par exemple des instructions update et delete dont la clause where décrit une collection de plusieurs lignes. On peut assigner au déclencheur relatif à cette instruction deux comportements différents : il doit être activé une seule fois par exécution de l'instruction, quel que soit le nombre de lignes affectées, ou il doit être activé pour chaque ligne modifiée par l'instruction. Dans le premier cas on précisera for each statement et dans le second cas for each row. Un déclencheur for each statement aura un comportement plutôt généraliste (compter les exécutions, vérifier des conditions globales, tenir un journal des instructions) tandis qu'un déclencheur for each row travaillera sur les données modifiées. D'ailleurs, les alias old et new ne sont pas utilisables dans le premier type de déclencheur; en DB2 en revanche, ce sont les états avant et après de la table entière qui sont disponibles sous les noms OLD_TABLE et NEW_TABLE.

18. Oracle admet notamment les événements suivants, qualifiés de *database-level event* : {before | after} {servererror | startup | shutdown | logon | logoff} on database.

La condition de déclenchement

La clause `when` présente deux avantages : elle permet de clarifier la structure du déclencheur et elle permet au SGBD d'identifier plus précisément les déclencheurs à exécuter lors de chaque événement. Certains SGBD n'admettent pas la clause `when C`. On utilisera dans ce cas la forme simplifiée :

```
before/after E
begin
    if C then A endif
end;
```

Les états de la ligne en cours de modification

Dans les diverses clauses d'un déclencheur, `old` et `new` désignent les états de la ligne en cours de modification respectivement *avant* et *après* la modification. Pour un `update`, ces deux états existent effectivement. 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.CAT` désigne donc la nouvelle valeur de `CAT` de la ligne courante tandis que `old.CAT` désigne son ancienne valeur. Dans un déclencheur `before`, il est possible de modifier la valeur des colonnes de la ligne courante avant enregistrement dans sa table. Dans le déclencheur `SUP_COM`, l'instruction `new.COMPTE := 0`; aurait permis de remettre à zéro le compte du client dont on supprime les commandes. Ceci n'est évidemment possible que dans un déclencheur `before`. Dans un déclencheur `after`, les états `old` et `new` sont également disponibles, mais ce dernier n'est plus modifiable puisque l'opération de modification a déjà été effectuée dans la base de données. Il est possible de modifier la ligne courante dans un déclencheur `after`, mais on est alors forcé d'utiliser une instruction `update` dont la clause `where` identifie cette ligne.

L'action d'un trigger

Le corps `A` d'un déclencheur est de complexité quelconque. Dans sa forme la plus simple, le déclencheur se contente d'annuler les opérations par une instruction spécifique, comme illustré dans l'exemple ci-après qui vérifie que la contrainte d'intégrité `C` (sous la forme d'une condition SQL appliquée à l'état `new` de la ligne courante) est respectée en cas d'insertion dans la table `T` :

```
before insert on T
for each row
when not C
begin
    abort();
end;
```

La fonction `abort()`, dont la syntaxe exacte est fonction du SGBD¹⁹, a pour résultat d'annuler les effets de l'opération de modification, ceux du déclencheur courant et de ceux de tous les déclencheurs qui auraient déjà été exécutés suite à cet événement.

19. Une forme courante est `raise exception (paramètres)`.

Le corps d'un déclencheur peut aussi être constitué d'une procédure longue et complexe utilisant des variables locales, exécutant des instructions SQL et interagissant avec l'environnement de la base de données. Certains SGBD imposent cependant des contraintes assez strictes sur les modifications autorisées dans le corps d'un déclencheur afin d'éviter les opérations *non déterministes* (dont l'effet, dans des conditions initiales identiques, peut être différent d'une exécution à l'autre). Par exemple, Oracle soumet le développement des déclencheurs à des règles qui empêchent un déclencheur d'accéder à une table dite *en mutation*, c'est-à-dire dont le contenu est en cours de modification, ou susceptible d'être indirectement modifié.

Le langage dans lequel l'action est rédigée peut être spécifique au SGBD (ex. PL/SQL d'Oracle, Transact-SQL de SQL Server et Sybase) ou au contraire être un langage général (Java, C, COBOL pour DB2 d'IBM). Certains SGBD offrent les deux possibilités : PL/SQL et Java chez Oracle.

Procédure d'exécution en présence de déclencheurs

Soit une instruction de modification **M** relative à la table T transmise par un programme au SGBD et susceptible d'agir sur plusieurs lignes de T. Lorsque des déclencheurs ont été définis sur T pour le type d'opération de **M**, le SGBD effectue les actions suivantes :

1. Le SGBD recherche dans les tables du catalogue (voir section 9.10) tous les déclencheurs de T relatifs au type d'opération de **M**.
2. Chacun des déclencheurs *before, for each statement* est exécuté.
3. Pour chacune des lignes sur lesquelles **M** est exécutée :
 - 3.1 Chacun des déclencheurs *before, for each row* est exécuté pour la ligne courante.
 - 3.2 L'opération **M** est exécutée sur la ligne courante.
 - 3.3 Chacun des déclencheurs *after, for each row* est exécuté pour la ligne courante.
4. Chacun des déclencheurs *after, for each statement* est exécuté.

Si l'opération **M** ou un des déclencheurs échoue (y compris par l'exécution d'une instruction `abort()`), toutes les actions effectuées lors des étapes 1 à 4 sont annulées. Dans ce cas, les données se retrouvent dans l'état où elles étaient avant l'étape 1 et le programme est averti de l'échec de **M**. Ce comportement est celui d'une *transaction*.

Remarques

- Un déclencheur est attaché à une table. Il peut cependant exécuter des modifications dans d'autres tables, et ainsi déclencher d'autres déclencheurs.
- Un déclencheur peut provoquer directement ou indirectement son propre redéclenchement. Ce cas est discuté ci-après.

- Plusieurs déclencheurs peuvent être attachés à une même table; dans ce cas, on précise l'ordre dans lequel ils sont déclenchés. Pour certains SGBD, cet ordre est explicitement défini lors de la création du déclencheur (clause `position` d'InterBase et Firebird, clause `follows` d'Oracle et MySQL). Pour d'autres, cet ordre est lié à celui de leur création (Oracle 8, DB2) ou de leur nom (PostgreSQL).
- Certains SGBD ne proposent que certains types de déclencheurs. Le tableau ci-dessous reprend les types proposés par les principaux SGBD.

SGBD	before statement	after statement	before row	after row	instead of
Oracle	X	X	X	X	X
Informix	X	X		X	X
DB2		X	X	X	X
SQL Server		X			X
InterBase/Firebird			X	X	X
PostgreSQL	X	X	X	X	X
MySQL			X	X	
SQLite			X	X	X

- Le corps d'un trigger peut demander l'exécution de procédures SQL.

Les déclencheurs récursifs

Un déclencheur dont l'action a pour effet de le réactiver est qualifié de **récursif**. Son usage (prudent) permet de résoudre de manière concise et élégante certains problèmes difficiles. Le script SQL ci-dessous stocke dans la table ORGA l'organigramme que produisait la vue récursive ORGA de la section 9.5.

```

create table ORGA (NIVEAU .., NPERS .., NOM .., RESPONSABLE ..;
create trigger ORG_INS                                [1]
after insert on ORGA
for each row
begin
    insert into ORGA
    select new.NIVEAU + 1, NPERS, NOM, RESPONSABLE
    from   PERSONNE where RESPONSABLE = new.NPERS;
end;
insert into ORGA                                      [2]
select 1, NPERS, NOM, RESPONSABLE
from   PERSONNE where NPERS = 'p4';

```

La requête finale [2] insère dans ORGA une première ligne, celle de la personne **p4**, de niveau 1. Le déclencheur [1] réagit en insérant dans ORGA les lignes des sous-données directs de cette personne (`PERSONNE where RESPONSABLE = new.NPERS`), avec un niveau incrémenté de 1. Ces insertions elles-mêmes activent ce déclencheur

pour insérer les subordonnés directs de ces derniers, et ainsi de suite. Au final, le contenu de ORGA est exactement celui de la vue récursive ORGA.

Pour éviter des chaînes infinies de déclenchements on s'assurera qu'une ligne dont la modification a activé un déclencheur, ne le déclenchera plus dans la suite de l'exécution. Une discussion plus approfondie des vues et des déclencheurs récursifs est proposée à l'annexe A9, ainsi que diverses applications, notamment dans le cadre des bases de données actives et des bases de données temporelles.

9.10 LE CATALOGUE

Une base 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*. Ces données décrivant les données de l'utilisateur, on les appellent *métadonnées*.

a) Principes d'un dictionnaire de données

Les tables du catalogue documentent tous les objets qui composent le schéma de la base de données, tels que les tables de base, les vues, les colonnes, les contraintes, les index, les déclencheurs, les utilisateurs et les privilèges. Ces informations étant rangées dans des tables SQL ordinaires, les utilisateurs et l'administrateur²⁰ de la base de données peuvent en consulter leur contenu. Elles sont également utilisées par le SGBD lui-même, qui peut ainsi vérifier la validité des requêtes qui lui sont soumises, les traduire en algorithmes efficaces, les exécuter en fonction des privilèges en cours (section 5.3).

b) Les tables du catalogue

Le dictionnaire contient plusieurs dizaines de tables, dont le nombre, le nom, la composition et l'interprétation varient 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 9.4). Nous présenterons brièvement la composition de ces deux tables.

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

20. Un administrateur d'une base de données destinée à desservir une communauté d'utilisateurs est un 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.

caractère obligatoire (NULLS = N) ou facultatif (NULLS = Y). Ces tables comportent d'autres colonnes qui ne présentent pas d'intérêt ici.

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 SYS_COLUMN elles-mêmes ainsi que leurs colonnes), dont le créateur est par convention SYSTEM.

SYS_TABLE		
TNAME	CREATOR	TTYPE
SYS_TABLE	SYSTEM	R
SYS_COLUMN	SYSTEM	R
SYS_KEY	SYSTEM	R
SYS_KEY_COMP	SYSTEM	R
....		.
CLIENT	AGF	R
COMMANDE	AGF	R
DETAIL	AGF	R
PRODUIT	PDE	R
VAL_STOCK	AFB	V
....		.

SYS_COLUMN					
TNAME	CNAME	CTYPE	LEN1	LEN2	NULLS
SYS_TABLE	TNAME	varchar	18		N
SYS_TABLE	CREATOR	char	8		N
....		..	..	..	.
CLIENT	ADRESSE	char	60		N
CLIENT	CAT	char	2		Y
CLIENT	COMPTE	decimal	9	2	N
CLIENT	LOCALITE	char	30		N
CLIENT	NCLI	char	10		N
CLIENT	NOM	char	32		N
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 9.4 - 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 complexe²¹. 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

21. 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 A9.5 de l'annexe A9, aborde cette question.

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.

SQLfast crée automatiquement un catalogue comportant les quatre tables décrites ci-avant, quel que soit le SGBD sous-jacent. Il est décrit à l'annexe A9.

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 9.5 - Deux tables du catalogue représentant les identifiants primaires, les clés étrangères, leurs composants et leurs relations

c) Utilisation du catalogue

Le contenu des tables du catalogue ne peut évidemment 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. Le contenu des tables du catalogue ne peut être modifié que par le SGBD lui-même lorsqu'il exécute une requête SQL-DDL create, drop, alter, 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.

- 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

```

- Quels 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

```

d) Information Schema, le standard ANSI

En réalité, jusqu'il y a peu, la plupart des SGBD ne mettaient à disposition les données du catalogue que via des commandes et des formats propriétaires et non sous la forme des tables décrites ci-avant. C'est le cas notamment de MySQL, SQLite et PostgreSQL. L'ANSI a défini, dans la norme SQL:2003, un ensemble de vues unifiant la présentation des métadonnées. Ces vues forment l'*Information Schema* de la base de données. Aujourd'hui, pratiquement tous les SGBD offrent cette fonctionnalité.

e) Le catalogue d'une base de données dans SQLfast

Pour toute base de données ouverte dans un script SQLfast, il est possible de demander la construction dynamique des quatre tables du catalogue (instruction `createDictionary`). Son utilisation est illustrée à l'annexe A9.

9.11 LES INTERFACES SQL DES PROGRAMMES D'APPLICATION

Toute instruction SQL peut être transmise au SGBD par un programme d'application en vue de son exécution immédiate. L'instruction est incorporée au programme selon des règles que nous allons décrire. Étant donné la nature procédurale et/ou orientée objet des langages de programmation classiques (COBOL, C/C++, Java,

Basic, C#, PHP, etc.), un protocole de communication spécifique doit être établi entre le programme d'application et le SGBD. Il faut en effet faire collaborer un programme qui fonctionne de manière itérative et un SGBD qui renvoie ses résultats sous la forme d'un ensemble de lignes²². Ce protocole concerne d'une part les échanges de données entre le programme et le SGBD et d'autre part le mode de livraison des lignes du résultat que le SGBD fournit en réponse à une requête.

Examinons l'exemple d'une requête SFW élémentaire rédigée comme nous l'avons fait jusqu'ici :

```
select NOM, LOCALITE
from   CLIENT
where  NCLI = 'C400';
```

Cette formulation n'est guère pratique pour une requête insérée dans un programme. En effet, il est probable que la valeur 'C400' n'est qu'un exemple, et que d'autres valeurs seront utilisées au fil des exécutions. D'autre part, on ne voit pas comment les deux valeurs de NOM et de LOCALITE composant la ligne du résultat seront transmises au programme pour y être traitées. Les trois composants 'C400', NOM, LOCALITE constituent des *paramètres* de la requête. Le premier doit être transmis à la requête avant son exécution (paramètre d'entrée) tandis que les valeurs des deux autres seront transmises au programme après l'exécution (paramètres de sortie). Il existe plusieurs modèles d'exécution d'une instruction SQL dans un programme, qui se classent selon deux dimensions :

1. SQL **statique** (*static SQL*) et SQL **dynamique** (*dynamic SQL*), selon le moment où l'instruction SQL est construite,
2. SQL **intégré** (*embedded SQL* ou *ESQL*) et SQL **appelé** (*call-level interface* ou *CLI-SQL*), selon la manière dont l'instruction est transmise au SGBD.

En *SQL statique*, les instructions SQL (DDL et DML) sont construites (rédigées) lors de l'écriture du programme, sont traduites lors de la phase de compilation du programme et exécutées lors de l'exécution du programme. En *SQL dynamique*, ces instructions SQL sont construites, compilées et exécutées lors de l'exécution du programme. Le langage SQL inclut des instructions spéciales destinées à demander la compilation et l'exécution des instructions DDL et DML.

En *SQL intégré* (ESQL), les instructions SQL apparaissent comme des instructions normales du programme au même titre que les assignations, les appels de procédure ou les instructions de contrôle. En *SQL appelé* (CLI-SQL), les instructions SQL sont transmises au SGBD via des appels de procédures. Le modèle *SQL appelé* implique nécessairement le modèle *SQL dynamique*. Nous examinerons successivement les interfaces ESQL statique et dynamique dites classiques, propres aux langages tels que C, Pascal ou COBOL ainsi que leur spécialisation au langage Java : JSQL (ESQL statique) et JDBC (CLI-SQL dynamique).

22. On pourrait ranger les lignes du résultat dans un tableau du programme, mais ce serait irréaliste pour un résultat de plusieurs millions de lignes !

La présentation se limitera à la description et à la justification des concepts fondamentaux mais n'a pas la prétention de constituer un manuel destiné au programmeur. Ce dernier trouvera dans la littérature technique de nombreuses références sur ces matières.

9.11.1 L'interface SQL statique classique

On distingue le *programme hôte*, rédigé dans un *langage hôte* (C, Java, COBOL, etc.), et les requêtes SQL qui sont insérées dans ce programme. Reprenons la requête précédente et montrons comment il conviendrait de l'écrire afin qu'elle puisse s'interfacer correctement avec le programme hôte. La forme suivante identifie les trois paramètres de la requête :

```
select NOM, LOCALITE into :varNOM, :varLOC
from   CLIENT
where  NCLI = :varNCLI;
```

Dans cette expression, **varNCLI**, **varNOM** et **varLOC** sont les noms de trois variables du programme hôte. Ces noms sont préfixés par ":" pour éviter toute confusion avec des noms de colonnes. Avant l'exécution de la requête, la variable **varNCLI** doit avoir reçu une valeur, et après l'exécution, les variables **varNOM** et **varLOC** auront chacune reçu une valeur. Ce canevas est convaincant pour une requête qui renvoie une seule ligne mais est inapproprié pour une requête qui renverrait plusieurs lignes. Dans ce cas, nous devons distinguer l'élaboration de l'ensemble des lignes du résultat et l'accès itératif à ces lignes. Nous serons donc amenés à étendre ce canevas. Après avoir précisé les conventions syntaxiques de l'interface, nous examinerons successivement les requêtes d'extraction renvoyant une seule ligne, les requêtes d'extraction renvoyant plusieurs lignes et les autres instructions SQL, telles que la modification des données et des structures de données.

a) Conventions syntaxiques

Un programme qui comporte des instructions SQL statiques est traité par un préprocesseur SQL qui traduit chaque ordre SQL en une procédure et remplace cet ordre dans le programme par un appel à celle-ci. Le programme est alors compilé et assemblé d'une manière traditionnelle. En outre, ce préprocesseur vérifie la conformité des ordres par rapport aux définitions qui sont enregistrées dans les tables du catalogue (en particulier les privilèges). Il met également celles-ci à jour²³.

23. Plus précisément, chaque instruction SQL est traduite en un module exprimé dans un code procédural propre au SGBD, dénommé **plan d'accès** dans la section 5.3. Ce module est stocké dans le catalogue. Lorsque le programme se connecte à la base de données, le SGBD vérifie que le schéma n'a pas été modifié depuis la compilation des instructions. Si le schéma a été modifié, alors le SGBD procède à une recompilation des instructions SQL avant de laisser le programme poursuivre son exécution. Selon cette procédure, les programmes ne doivent être ni modifiés ni même recompilés lorsque les structures de données sont modifiées (suite à l'ajout/modification/suppression d'une table, d'une colonne, d'une contrainte, d'un index, etc.)

Nous précisons les conventions qui sont d'application pour les langages tels que COBOL, C ou PASCAL. Le cas de Java sera étudié spécifiquement à la section suivante.

1. Toute instruction SQL doit être précédée de "exec SQL" et clôturée par "end-exec" en COBOL et ";" en C. Ceci permet son repérage par le préprocesseur.
2. Les variables qui interviennent dans une instruction SQL comme paramètres d'entrée et de sortie doivent être déclarées dans un bloc délimité par les instructions :

```
exec SQL begin declare section;
et
exec SQL end declare section;
```

3. Les communications entre le programme hôte et le SGBD SQL utilisent certaines variables système (dont SQLCODE) qu'il faut déclarer, ou qu'on peut faire introduire par le préprocesseur par la commande :

```
exec SQL include SQLCA;
```

b) Requêtes d'extraction produisant une seule ligne

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-après, 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.

```
exec SQL begin declare section;
varNOM char(32);
varLOC char(30);
varNCLI char(10);
exec SQL end declare section;
exec SQL include SQLCA;
read varNCLI;
exec SQL
select NOM, LOCALITE into :varNOM, :varLOC
from CLIENT
where NCLI = :varNCLI;
write varNOM, varLOC;
```

c) Requêtes d'extraction produisant plusieurs lignes

Un résultat comportant plusieurs lignes nécessite une procédure plus complexe. 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 le prochain enregistrement
```

```
tant que la lecture a réussi faire
    traiter l'enregistrement
    lire le prochain 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 virtuel qu'il faut ouvrir, qu'on peut lire ligne par ligne, puis qu'il faut fermer lorsque son traitement est terminé. Le curseur retient également l'endroit dans ce résultat où se fera la prochaine lecture (d'où le terme de *curseur*).

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 fictif. 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 unique dans le programme, 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`) range les valeurs des colonnes de cette ligne dans des variables du programme; le curseur est alors 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. À 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*). Cet indicateur fait partie d'un bloc d'indicateurs dénommé `SQLCA` (*SQL Control Area*) permettant au programme et au SGBD d'échanger des informations.
5. La fermeture du fichier résultat (`close`) entraîne son abandon. Toute ouverture ultérieure provoquera à nouveau l'évaluation de la requête. Entre une fermeture et une ouverture, on peut donc modifier des valeurs des paramètres d'entrée de la requête.

L'exemple suivant extrait et traite les informations relatives aux clients de Toulouse. La déclaration des variables partagées et de `SQLCA` est ignorée.

```
exec SQL declare CURCLI cursor for
select NCLI,NOM
from CLIENT
where LOCALITE = :varLOC;
```

```

varLOC := "Toulouse";
exec SQL open CURCLI;
exec SQL fetch CURCLI into :varNCLI,:varNOM;
while SQLCODE = 0 do
    <traiter les valeurs de varNCLI et varNOM>
    exec SQL fetch CURCLI into :varNCLI,:varNOM;
endwhile;
exec SQL close CURCLI;

```

d) Modification des données

Les commandes insert, delete, et update peuvent, elles-aussi, apparaître dans un programme, selon les conventions syntaxiques spécifiées pour les requêtes SFW produisant une seule ligne. 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 NCLI,NOM from CLIENT where LOCALITE = :varLOC;
varLOC := "Toulouse";
exec SQL open CURCLI;
exec SQL fetch CURCLI into :varNCLI,:varNOM;
while SQLCODE = 0 do
    if STAT > 0 then
        exec SQL update CLIENT
            set      CAT = 'A1'
            where current of CURCLI;
    endif
    exec SQL fetch CURCLI into :varNCLI,:varNOM;
endwhile;
exec SQL close CURCLI;

```

e) Définition et modification des structures de données

Les commandes SQL DDL create, alter, drop, etc. peuvent être exécutées par un programme. Elles apparaissent, tout comme les instructions SFW, dans le texte de celui-ci entre "exec SQL" et ";" ou "end-exec".

9.11.2 SQLJ : une interface statique pour Java

SQLJ (SQL-Java) est une extension du langage Java permettant au programme d'interagir avec une base de données SQL via une interface ESQL statique. Le précompilateur traduit les ordres SQLJ en appels à l'interface JDBC que nous étudierons plus loin. Avant l'exécution d'instructions SQL, le programme se connecte à une base de données en créant un objet de la classe Db et en lui demandant de créer une connexion, appelée *contexte*. Il a été proposé par Oracle après

l'introduction de JDBC afin d'améliorer la lisibilité des programmes et la productivité des programmeurs.

a) Requêtes d'extraction produisant une seule ligne

La requête spécifie les paramètres d'entrée et de sortie sous forme de variables du programme hôte.

```
String varNCLI, varNOM, varLOC;

varNCLI = "C400";
#sql {select NOM, LOCALITE into :varNOM, :varLOC
      from CLIENT where NCLI = :varNCLI};

<traiter les valeurs des variables varNOM et varLOC>
```

b) Requêtes d'extraction produisant plusieurs lignes

Le rôle du curseur est joué en SQLJ par un objet *itérateur* (*iterator*). Cet objet est créé en deux temps : création d'une classe d'itérateurs propre à la requête [1] puis création d'un itérateur de cette classe [2]. SQLJ propose deux types d'itérateurs de même fonctionnalité : nommés (*named iterator*) et positionnels (*positional iterators*).

Itérateur nommé

La définition d'une classe d'*itérateurs nommés* spécifie le nom et le type des paramètres de sortie [1]. L'exécution de la requête [3] assigne l'ensemble des lignes du résultat à l'itérateur. L'accès aux lignes successives s'obtient par la méthode `next()` [4], qui renvoie `null` en cas d'absence de ligne. Les valeurs des paramètres de sortie sont directement extraites de l'itérateur par des méthodes dont le nom est celui de ces paramètres [5]. Après usage, l'itérateur est fermé [7].

```
#sql iterator ItérateurClient(String NCLI, String Nom);      [1]
ItérateurClient CurCli;                                     [2]
String varLOC = "Poitiers";
#sql CurCli = {select NCLI, NOM from CLIENT
               where LOCALITE = :varLOC};                   [3]
while (CurCli.next()) {                                    [4]
    varNCLI = CurCli.NCLI();                                [5]
    varNOM  = CurCli.Nom();                                  [5]
    <traiter les valeurs de varNCLI et varNOM>
}
CurCli.close();                                           [7]
```

Itérateur positionnel

La définition d'une classe d'*itérateurs positionnels* spécifie uniquement le type des paramètres de sortie [1]. L'accès aux lignes successives du résultat s'obtient par la commande SQL classique `fetch into`, qui range dans les variables ad hoc les

valeurs des paramètres de sortie [6]. La méthode `endFetch()` de l'itérateur indique si la dernière instruction `fetch` a trouvé une ligne.

```
#sql iterator IterateurClient(String, String); [1]
IterateurClient CurCli; [2]
String varLOC = "Poitiers";
#sql CurCli = {select NCLI,NOM from CLIENT
               where LOCALITE = :varLOC}; [3]
#sql {fetch :CurCli into :varNCLI, :varNOM}; [6]
while (!CurCli.endFetch()) {
    <traiter les valeurs de varNCLI et varNOM>
    #sql {fetch :CurCli into :varNCLI, :varNOM}; [6]
}
CurCli.close(); [7]
```

c) Modification des données et des structures

Les instructions `insert`, `update`, `delete`, `create`, `drop` et `alter` suivent le canevas des requêtes renvoyant une seule ligne :

```
varCAT = "C2"; varNCLI = "C400";
#sql {update CLIENT set CAT = :varCAT
      where NCLI = :varNCLI};
```

9.11.3 L'interface SQL dynamique classique

SQL dynamique désigne un mode d'interaction avec le SGBD selon lequel les instructions SQL ne sont pas rédigées en même temps que le programme hôte, mais sont construites dynamiquement au moment de son exécution. Leur traitement se fait en trois phases qui se déroulent pendant l'exécution du programme :

1. construction de la requête sous la forme d'une chaîne de caractères,
2. préparation (ou compilation) de la requête,
3. exécution de la requête ainsi préparée.

L'interface classique correspond au modèle SQL intégré (*ESQL*) mais des variantes CLI existent. Le langage SQL comporte des instructions spécifiques contrôlant la compilation et l'exécution de requêtes, notamment `prepare`, `execute` et `execute immediate`.

a) Requêtes d'extraction produisant une seule ligne

Nous illustrerons les principes généraux de cette interface par un exemple exprimé en pseudo-code, dans lequel une requête SFW renvoyant une seule ligne est exécutée dynamiquement. On y distingue les instructions de création [C], de préparation [P] et d'exécution de la requête [E].²⁴

```
varNCLI char(10);
exec SQL include SQLCA;
```

```

exec SQL begin declare section;
    varNOM    char(32);
    varLOC    char(30);
    Requete   char(250);
exec SQL end declare section;

varNCLI := "C400"; [C]
Requete := "select NOM,LOCALITE from CLIENT "
          + "where NCLI = '" + varNCLI + "'"; [C]

exec SQL prepare Q from :Requete; [P]
exec SQL execute Q into :varNOM, :varLOC; [E]

```

Dans certains cas simples, les phases de préparation et d'exécution pourront être fusionnées²⁵ :

```

exec SQL execute immediate from :Requete
    into :varNOM, varLOC; [PE]

```

Les paramètres de sortie de la requête (ici varNOM et varLOC) sont spécifiés dans la clause `into` de la commande `execute`. Le rôle du paramètre d'entrée est joué par la variable varNCLI dont la valeur a été injectée sous forme d'une constante dans la chaîne de la requête au moment de sa construction. Il existe une méthode de spécification des paramètres d'entrée plus élégante et plus sûre. Elle suppose que la chaîne de la requête indique par un symbole spécial (en l'occurrence "?") la position de chaque constante et non plus sa valeur. La section de code précédente devient, si on se limite aux instructions utiles :

```

varNCLI := "C400"; [C]
Requete := "select NOM,LOCALITE from CLIENT "
          + "where NCLI = ?"; [C]

exec SQL prepare Q from :Requete; [P]
exec SQL execute Q using :varNCLI into :varNOM, :varLOC; [E]

```

On observe que l'instruction `execute` mentionne les paramètres d'entrée (`using`) et les paramètres de sortie (`into`).

Dans les exemples présentés, les parties variables de la requête sont les constantes apparaissant dans la clause `where`. En réalité, tous les composants d'une requête peuvent être variables, comme le montre l'exemple suivant :

```

var1 := "select ";
var2 := "NOM, ADRESSE";
var3 := "CLIENT";
var4 := "LOCALITE = ?";
Requete := var1 + var2 + " from " + var3 + " where " + var4;

```

24. Rappelons au lecteur un peu perdu que le délimiteur de chaînes de caractères est `'` en SQL et `"` dans le langage hôte. De même, l'assignation se note `=` en Java et `:=` dans notre pseudo-code représentant le langage hôte.

25. Mais dans cette variante, la compilation est effectuée pour chaque exécution de la requête.

Les valeurs des fragments peuvent provenir de sources quelconques, telles que l'utilisateur, les tables du catalogue ou la base de données elle-même. On conçoit aisément qu'une telle souplesse entraîne une certaine complexité dans le traitement de ces requêtes. Par exemple, le nombre et la nature des paramètres de sortie (le contenu de `var2`) peuvent être inconnus au moment de la compilation du programme. L'interface classique (*ESQL*) est relativement complexe sur ce point, de sorte qu'on se contentera d'illustrer ce type de requête pour le modèle *CLI* à l'occasion de l'étude de JDBC.

b) Requêtes d'extraction produisant plusieurs lignes

La logique d'exécution s'inspire de celle de la variante statique, basée sur les curseurs.

```
Requete := "select NCLI,NOM from CLIENT "
         + "where LOCALITE = ?";

exec SQL declare CURCLI cursor for inst;
exec SQL prepare inst from :Requete;
varLOC := "Poitiers";
exec SQL open CURCLI using :varLOC;
exec SQL fetch CURCLI into :varNCLI,:varNOM;
while SQLCODE = 0 do
    <traiter les valeurs de varNCLI et varNOM>
    exec SQL fetch CURCLI into :varNCLI,:varNOM;
endwhile;
exec SQL close CURCLI;
```

On observe que, lors de sa déclaration, un curseur n'est pas associé à une instruction SQL spécifique mais à une variable requête libre (`inst`). C'est dans l'étape de préparation que cette variable est associée à une instruction SQL réelle. Pour le reste, les instructions `open`, `fetch` et `close` suivent le modèle classique.

c) Modification des données et des structures

La demande d'exécution des requêtes de modification suit le même scénario que les requêtes produisant une seule ligne. Par exemple, une instruction d'insertion sera construite comme suit :

```
Requete := "insert into DETAIL values (?, ?, ?)";
exec SQL prepare Q from :Requete;
exec SQL execute Q using :valNCOM, :valNPRO, :varQCOM;
```

ou :

```
Requete = "insert into DETAIL values ("
         + valNCOM + "',' + valNPRO + "',' + valQCOM + ")";
exec SQL execute immediate from :Requete;
```

ou encore :

```
exec SQL execute immediate from
    "insert into DETAIL values ('"
    + valNCOM + "','" + valNPRO + "','" + valQCOM + ")";
```

9.11.4 JDBC : une interface CLI-SQL dynamique pour Java

JDBC (Java Database Connectivity) est une interface développée initialement par Sun permettant à un programme Java d'interagir avec une ou plusieurs bases de données relationnelles. Les instructions sont (syntaxiquement du moins) indépendantes des particularités du SGBD de sorte qu'un programme Java/JDBC peut (en principe) accéder à une base de données Oracle, DB2, SQL Server ou MySQL, pour autant que les fournisseurs de ces derniers aient livré les pilotes propres à leur moteur SQL. La plupart des instructions SQL peuvent être transmises au SGBD via l'interface JDBC. Nous illustrerons cette dernière pour quelques instructions importantes. Dans les exemples ci-après, nous supposerons que le pilote choisi a déjà été chargé par le programme.

a) Requêtes d'extraction produisant une seule ligne

L'interaction avec une base de données utilise principalement trois objets Java que le programme doit créer : une connexion avec une base de données (`conn`, de la classe `Connection`), une instruction (`inst`, de la classe `Statement` ou `PreparedStatement`) qu'on exécutera dans cadre de cette connexion et le résultat de cette exécution (`res`, de la classe `ResultSet`). Après création de la connexion [1],

1. la requête SQL est construite dans la variable `requete`; la position de chaque paramètre d'entrée est indiquée par le symbole "?" [2] ;
2. la requête est compilée (préparée) dans l'objet `inst` par la méthode `prepareStatement` de `conn` [3] ;
3. la valeur du paramètre d'entrée est spécifiée par la méthode `setString` (ou `setInt`, `setFloat`, etc) de `inst` [4] ;
4. l'instruction est exécutée par la méthode `executeQuery` de `inst` et renvoie les données extraites sous la forme de l'objet `res`; cet objet joue le rôle de curseur [5] ;
5. l'objet `res` est positionné, par la méthode `next()`, sur la prochaine ligne renvoyée (ici la première) [6]; cette méthode renvoie `true` si une ligne a été trouvée et `false` sinon ;
6. la valeur de chaque paramètre de sortie est extraite par la méthode `getString` (ou `getInt`, `getFloat`, etc) de `res` [7] ;
7. l'objet `res` est fermé [8] ;
8. les étapes 4 à 8 peuvent être effectuées plusieurs fois.


```

String locBD, login, motDePasse;
String requete, varNCLI, varNOM, varLOC;
Connection conn;
PreparedStatement inst;
ResultSet res;

urlBD = <adresse de la BD>; login = "j1h"; motDePasse = ..; [1]
conn = DriverManager.getConnection(urlBD, login, motDePasse); [1]
requete = "select NOM, LOCALITE from CLIENT "
        + "where NCLI = ?"; [2]
inst = conn.prepareStatement(requete); [3]
varNCLI = "C400"; [4]
inst.setString(1, varNCLI); [4]
res = inst.executeQuery(); [5]

if (res.next()) { [6]
    varNOM = res.getString(1)26; [7]
    varLOC = res.getString(2);
    <traiter les valeurs de varNOM et varLOC>
};

res.close(); [8]

```

Il existe une variante simplifiée, correspondant à un *execute immediate*, dans laquelle l'instruction `inst` n'est pas préparée mais créée sans contenu [3] puis exécutée [5]. Ici encore, les étapes 4 à 8 peuvent être effectuées plusieurs fois.

```

Statement inst;

inst = conn.createStatement(); [3]
varNCLI = "C400"; [4]
requete = "select NOM, LOCALITE FROM CLIENT where NCLI = '"
        + varNCLI + "'"; [4]
res = inst.executeQuery(requete); [5]

if (res.next()) { [6]
    varNOM = res.getString(1); [7]
    varLOC = res.getString(2);
    <traiter les valeurs de varNOM et varLOC>
};

res.close(); [8]

```

b) Requêtes d'extraction produisant plusieurs lignes

L'exécution de requêtes à résultat multilignes s'extrapole sans difficulté à partir du canevas précédent :

26. On peut aussi écrire `res.getString("NOM")`

```

requete = "select NCLI, NOM from CLIENT "
        + "where LOCALITE = ?"; [2]

inst = conn.prepareStatement(requete); [3]

varLOC = "Poitiers"; [4]
inst.setString(1, varLOC); [4]

res = inst.executeQuery(); [5]

while (res.next()) { [6]
    varNCLI = res.getString(1); [7]
    varNOM = res.getString(2);
    <traiter les valeurs de varNCLI et varNOM>
};

res.close(); [8]

```

c) Modification des données et des structures

Le canevas est similaire à celui d'une requête qui ne renvoie qu'une seule ligne. On utilise la méthode `executeUpdate`.

```

requete = "update CLIENT set CAT = ? where NCLI = ?";

inst = con.prepareStatement(requete);

varCAT = "C2"; varNCLI = "C400";
inst.setString(1, varCAT);
inst.setString(2, varNCLI);

inst.executeUpdate();

```

d) L'interface DB-API de Python

On trouvera à l'annexe A9 un bref descriptif de l'interface SQL dynamique comprise dans la distribution de Python.

9.11.5 Comparaison des modèles d'interaction

SQL intégré et SQL appelé

Le modèle SQL *intégré* (*ESQL*) conduit à une écriture claire et lisible des instructions SQL, puisque chacune d'elles apparaît comme une instruction du programme. Les programmes sont concis et lisibles. Du côté négatif, ce modèle exige un préprocesseur spécifiquement développé pour le langage hôte. Ce composant est complexe pour SQL statique (tout le langage est à traduire) alors qu'il est fortement simplifié en SQL dynamique (`prepare`, `execute`, etc. mais pas les instructions classiques)

Le modèle *appelé* (*CLI-SQL*) est plus facile à ajouter à un langage hôte quelconque pourvu qu'il admette l'appel de procédures avec passage d'arguments. En revanche, le code des programmes est moins lisible, et donc plus difficile à maintenir puisque les instructions SQL n'apparaissent que comme valeurs de variables ou comme expressions calculées. La compilation des instructions SQL est effectuée

au moment de l'exécution du programme. ODBC et JDBC sont les *interfaces appelées* les plus populaires. La première a été développée par Microsoft et la seconde résulte d'une initiative de Sun Microsystems.

Rappelons que le modèle *intégré* donne accès aux variantes statique et dynamique de SQL tandis que le modèle *appelé* implique obligatoirement SQL dynamique.

SQL statique et SQL dynamique

Les instructions en ***SQL statique*** produisent un code concis, clair et lisible. En effet, chaque instruction SQL correspond à une instruction du programme hôte et cette instruction exprime de manière explicite ses paramètres d'entrée et de sortie. La traduction et l'optimisation des requêtes sont réalisées lors de la précompilation, c'est-à-dire avant l'exécution des programmes. En revanche, les seuls composants qu'il est possible de faire varier d'une exécution à l'autre d'une instruction sont les constantes qui apparaissent dans la clause *where*. Le précompilateur doit prendre en charge la totalité du langage SQL.

En ***SQL dynamique***, toutes les parties d'une instruction SQL peuvent varier d'une exécution à l'autre. Il est ainsi possible de construire des programmes adaptatifs très compacts. En revanche, de tels programmes sont très peu lisibles et difficiles à tester et à maintenir. La nécessité de compiler les requêtes au moment de l'exécution entraîne un coût d'exécution supérieur, même dans un scénario où les instructions sont *préparées*. Dans le modèle intégré, les instructions SQL à précompiler sont très peu nombreuses, tout l'effort étant reporté au moment de l'exécution.

Signalons encore que l'usage du modèle ***SQL dynamique appelé*** crée des problèmes de sécurité que nous présenterons succinctement ci-après.

En résumé

On donnera la préférence autant que possible une interface statique intégrée. Elle donne en effet les meilleures performances et conduit à des programmes concis, lisibles, faciles à écrire et à maintenir. Elle exige en revanche un précompilateur spécifique, qui n'existe pas pour tous les SGBD ni tous les langages hôtes, et convient à des applications stables et peu dynamiques.

9.11.6 Un problème de sécurité : l'injection de code SQL

L'utilisation négligente²⁷ de code SQL dynamique construit à partir de fragments fournis par un utilisateur crée une grave vulnérabilité de sécurité à une attaque dénommée *injection de code SQL* [Ben Natan, 2005], [Clarke, 2009]. Nous l'illustrerons par l'exemple classique de la connexion à un système de gestion bancaire sécurisé (figure 9.6).

27. Selon la plupart des sites de sécurité, l'injection de code SQL constituerait la première technique d'attaque d'un site web jusqu'en 2016. Consulter par exemple <https://codecurmudgeon.com/wp/sql-injection-hall-of-shame/> et <https://blog.checkpoint.com/2015/05/07/latest-sql-injection-trends/>

Figure 9.6 - Une boîte de dialogue de connexion à un système bancaire. À gauche, réponses *loyales* (mot de passe en clair). À droite, tentative d'injection de code SQL.

L'utilisateur, normalement un agent de la banque, doit introduire son code et son mot de passe (figure 9.6, gauche), lesquels sont saisis par le gestionnaire de formulaires par la procédure `getForm` et rangés dans les deux variables `varLOGIN` et `varPW`.

Le contenu de celles-ci est injecté par concaténation dans une chaîne de caractères de manière à former dans la variable `requete` une requête SQL vérifiant s'il existe dans la table `SYS_USER`(`ID_USER`, `PW`, ...) une ligne correspondant à ces deux valeurs. Si aucune ligne n'est trouvée ($N = 0$), l'accès est refusé, sinon, il est autorisé, décision qui est traduite par la valeur de la variable booléenne `accepte`. Le code du programme de contrôle d'accès qui assure la saisie des données, constitue la requête, l'exécute et évalue l'autorisation pourrait ressembler à ceci :

```
getForm(formID,varLogin,varPW);
requete = "select count(1) from SYS_USERS "
        + " where ID_USER = '" + varLogin + "'"
        + " and PW = '" + varPW + "' ";
exec SQL execute immediate from :requete into :N;
if (N = 0) then accepte = False else accepte = True;
```

Dans l'hypothèse où l'agent joue honnêtement le jeu, et introduit son code et son mot de passe, la requête exécutée pourrait être la suivante :

```
select count(*) into :N
from SYS_USERS
where ID_USER = 'Albert-Durant'
and PW = 'A7cfg990';
```

Supposons à présent qu'un attaquant se faisant passer pour un agent tente d'accéder au logiciel de gestion bancaire. Comme il ne dispose ni d'un code ni d'un mot de passe valides, la procédure normale lui interdira en principe l'accès. La chose n'est pas si sûre ! Supposons en effet qu'il introduise dans le formulaire d'accès les données suivantes (guillemets extérieurs non compris) : `" ' or ' ' = ' "` (figure 9.6, droite). Dans ce cas, la requête que construira le programme de contrôle d'accès aura la forme suivante :

```
select count(*) into :N
from SYS_USERS
where ID_USER = 'X'
or 'X' = 'X'
and PW = ' '
or 'X' = 'X';
```

Compte tenu de la priorité de l'opérateur `and` sur le `or`, la condition de la clause `where` aura toujours la valeur `true`, de sorte que l'attaquant se verra accorder illégalement l'accès au logiciel de gestion ! Cet exemple est implémenté en SQLfast et discuté plus en détail dans l'annexe A9. Cet exemple a valeur purement pédagogique. Dans la réalité, ce n'est pas le mot de passe qui est stocké dans la table mais sa signature (voir section 4.16.2).

Contre-mesures

L'analyse du mécanisme d'injection permet de comprendre que la vulnérabilité réside dans un usage inapproprié du modèle *SQL dynamique appelé*. On observe en effet que ce type de contrefaçon devient impossible si :

- la requête est exprimée en SQL statique,
- ou si elle est exécutée en mode dynamique mais avec préparation (`prepare Statement` en JDBC) et liaison des variables d'entrée (`setString` en JDBC) en évitant soigneusement la construction par concaténation de valeurs.

Si ces approches ne sont pas possibles, une analyse et un traitement des valeurs transmises par l'utilisateur peuvent être envisagés. En particulier, une apostrophe ne peut normalement apparaître que comme un caractère de la valeur transmise. Dans ce cas, on peut dédoubler chaque apostrophe pour respecter la syntaxe des constantes SQL (section 7.2.2). Les apostrophes pirates à la base de l'attaque provoquent des erreurs de syntaxe et deviennent inopérantes.

9.12 SQL ET L'INFORMATION INCOMPLÈTE

9.12.1 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 ne peut avoir de catégorie ;
- *information inconnue* : le client a une catégorie, mais celle-ci n'est pas connue.

9.12.2 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.

9.12.3 La logique ternaire de SQL

On rappelle que le langage SQL, pour certains de ses aspects, est basé sur une logique ternaire, incluant la valeur de vérité *unknown*. L'extension des opérateurs logiques et leurs tables de vérité sont présentées dans la section A24.4.8 de l'annexe A24. La valeur logique *unknown* est l'un des moyens dont dispose SQL pour exprimer l'information incomplète. Rappelons encore l'existence de six prédicats SQL qui permettent de réduire la logique ternaire à la logique binaire :

- (CAT = 'B1') is true (CAT = 'B1') is not true
- (CAT = 'B1') is false (CAT = 'B1') is not false
- (CAT = 'B1') is unknown (CAT = 'B1') is not unknown

9.12.4 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 est *null* ;
- ADRESSE || ' - ' || CAT vaut *null* si CAT est *null* ;
- COMPTE - COMPTE vaut *null* si COMPTE est *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.

9.12.5 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 préciserons ensuite les principales règles d'évaluation des prédicats, qui s'ajoutent à la table de vérité décrite précédemment.

a) Les expressions de comparaison

Une comparaison entre deux valeurs dont une au moins est `null` renvoie `unknown`.²⁸ 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 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 = 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 `unknown`²⁹. Le prédicat `not in` s'étudie de manière similaire.

28. Sauf, rappelons-le, pour Access, qui interprète malheureusement `CAT = null` comme `CAT is null`.

29. 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.

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 op any (E)`" et "`A 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].

9.12.6 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, `null` 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

Il n'en va pas de même de la requête suivante 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 7.6.4, et comme nous venons de le rappeler, la raison en est que la comparaison de deux expressions dont la valeur est `null` renvoie `unknown`. 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` est `null` ont *même valeur* de `CAT`, et donc qu'ici `null = null`. On retrouvera cette interprétation dans la clause `distinct`, qui éliminera les doublons, y compris ceux qui incluent `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'agrément 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` :

```
NUM_AGREMENT 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 identique³⁰. Conclusion, ici encore, `null = null`.

30. À ces interprétations divergentes, certains SGBD ont assimilé une chaîne de caractères vide à l'absence de valeur (`null`). Cette erreur a été corrigée dans les versions ultérieures. La différence semble subtile mais est importante. 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. Concaténer une chaîne `S` avec une chaîne vide produit `S` alors que la concaténation avec `null` produit `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)³¹ :

```
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 est 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 < 'C2') or (CAT = 'C2') or (CAT > 'C2')
```

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 les nombres 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 9.7, où null est représenté par une cellule vide).

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 9.7 - Table d'activités

On désire calculer la charge de chaque titulaire. Deux expressions sont possibles :

31. SQL permet d'imposer la position de null comme suit : order by CAT nulls last

```
select TITULAIRE,
       sum(H_COURS) + sum(H_TP)
       as CHARGE
from   ACTIVITE
group by TITULAIRE;
```

```
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	<null>	NHA	<null>
PHE	90	PHE	45
VEN	<null>	VEN	<null>

Il est inutile de soupçonner une quelconque erreur de calcul, SQL applique fidèlement les règles d'évaluation décrites précédemment.

De même, pour un prédicat P quelconque, les deux expressions ci-après ne donnent pas nécessairement le même résultat, même si P relève de la logique 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 8.13.3.

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 !

9.12.7 Deux recommandations

En raison des multiples difficultés qu'entraîne l'usage de la valeur `null`, certains auteurs recommandent de les éviter purement et simplement³². Citons deux techniques qui permettent de les contourner si on l'estime nécessaire :

a) *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);
```

b) *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);
```

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'empêche malheureusement pas 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.

32. Citons par exemple les références [Codd, 1989], [Date, 1992], et [Date, 1997], auxquelles nous avons emprunté certains exemples de cette section.

Face à ces critiques, les auteurs du langage SQL avancent des arguments de bon sens qu'on retrouvera par exemple dans [Chamberlin, 1996].

9.13 QUE RETENIR ?

SQL comporte un sous-langage permettant de contrôler l'accès aux données. Tout propriétaire d'un objet, tel qu'une table ou une procédure, peut accorder à un autre utilisateur de droit d'exécuter certaines actions sur cet objet (`grant`). Ce droit constitue un **privilège**. À un privilège peut être associé un droit secondaire, celui de transmettre le privilège reçu à d'autres utilisateurs. Un privilège peut être retiré par celui qui l'a accordé (`revoke`).

Étant donné une base de données, il est possible de définir un nombre quelconque de tables virtuelles ou **vues** (`create view`). Une vue est spécifiée par la requête SFW qui en calcule le contenu chaque fois qu'on y fait référence. Une vue peut faire l'objet de requêtes SFW au même titre que les tables de base. En revanche, les possibilités de modifier les données d'une vue sont limitées. SQL propose une variante locale des vues, les *Common Table Expressions*. La définition d'une CTE peut mentionner celle-ci, ce qui permet de construire des **requêtes récursives**.

L'apport principal de SQL3 est celui des **structures d'objets**, qui transforment le modèle *relationnel* en *relationnel objet*. Ces structures incluent en particulier les types complexes `row` et `array`, les types définis par l'utilisateur (TDU), l'organisation des types en une hiérarchie d'héritage (`surtype/sous-type`), les colonnes et tables définies sur un TDU, les identifiants d'objet et les hiérarchies de tables.

SQL inclut trois constructions particulièrement puissantes permettant de développer ce qu'on appelle des *bases de données actives*. Il s'agit des **prédicats** (`check`), qui permettent d'imposer des contraintes que les données doivent respecter, des **procédures SQL** (`stored procedure`) qui peuvent être invoquées par tout utilisateur et des **déclencheurs** (`trigger`), procédures exécutées automatiquement lors de la survenance de certains événements affectant les données.

Toute base de données comporte un ensemble de tables, dit **catalogue**, qui contiennent la description de tous les objets du schéma. Parmi les plus importantes, on mentionnera la table des tables, la table des colonnes, la table des clés et celle des composants de clés. Le contenu des tables du catalogue constitue des *métadonnées* (c'est-à-dire des *données décrivant des données*).

L'accès aux données à partir des **programmes d'application** se fait au travers d'interfaces techniques appropriées. On distingue *l'interface statique*, selon laquelle les ordres SQL apparaissent comme des instructions normales dans le programme, et *l'interface dynamique*, par laquelle les instructions SQL sont construites dynamiquement lors de l'exécution du programme. L'interface dynamique doit être utilisée avec prudence pour éviter des problèmes de sécurité.

L'information incomplète est partiellement prise en charge en SQL par la valeur `null` et la valeur logique `unknown`. La manipulation de ces valeurs doit faire l'objet d'une attention toute particulière. En effet, leur comportement et leur interprétation sont parfois non intuitifs, voire incohérents.

9.14 POUR EN SAVOIR PLUS

Les librairies et les bibliothèques regorgent d'ouvrages techniques abordant les concepts abordés dans ce chapitre dans une perspective de programmation. Les références décrivant et discutant les principes à la base de ces techniques sont plus rares. Bien que consacré à Oracle exclusivement, [Owens, 2004] constitue une excellente référence dans le domaine des déclencheurs et des procédures SQL. Le lecteur trouvera des compléments d'information sur SQL3 dans [Melton, 2002] et [Melton, 2003], qui lui sont entièrement consacrés. L'approche relationnelle objet est également traitée dans des ouvrages plus généraux tels que [Garcia-Molina, 2008], [Gardarin, 2003], [Elmasri, 2010] et [Date, 2004]. On trouvera un exposé d'applications plus spécifiques dans ces ouvrages mais aussi dans [Connolly, 2009].

Le web est riche en ressources concernant directement le langage SQL et son utilisation. Quelques points d'entrée à titre d'exemple : <http://sqlpro.developpez.com>, <http://dba.stackexchange.com>, <http://stackoverflow.com>,

La sécurité des bases de données, et en particulier la protection contre les intrusions, est développée de manière approfondie dans [Ben Natan, 2005] et [Clark, 2009].

L'annexe A9 propose un jeu d'exercices, dont certains sont accompagnés d'une solution, sur les concepts de ce chapitre. On y trouvera aussi des compléments d'information et des applications avancées basées sur ces concepts, ainsi que de nombreux scripts SQLfast complets.

Chapitre 10

Les bases de données non relationnelles

Ce chapitre décrit les objectifs et les caractéristiques des systèmes de gestion de données non relationnelles, en particulier ceux de la catégorie dite NoSQL. On étudie leurs modèles de données et les défis posés par les données massivement distribuées. Une présentation plus détaillée est proposée pour deux SGBD représentatifs : Cassandra et MongoDB. Ces technologies sont comparées à celle des SGBD SQL. On abordera également un type particulier de base de données massivement répliquée, la chaîne de blocs.

10.1 INTRODUCTION

Si la majorité des bases de données utilisent un SGBD SQL, il existe, dans l'offre technologique actuelle, une grande variété de SGBD qui s'écartent plus ou moins radicalement de ce standard. Outre le désir légitime d'inventer et d'explorer d'autres voies, ces technologies cherchent à répondre à des défis que ne rencontre pas, selon leur promoteurs, la technologie relationnelle.

a) Les nouveaux besoins

L'argumentaire en faveur de ces nouveaux SGBD énumère des critiques dont on reprendra les principales.

- **Représentativité des structures de données.** Le modèle de données relationnel ne représente pas naturellement les entités du domaine d'application. En raison de

leur normalisation, les données sont distribuées dans plusieurs tables, ce qui entraîne de nombreuses et coûteuses jointures pour rassembler des données de ces entités. En outre, les SGBD relationnels sont conçus pour gérer des données de structure tabulaire mais sont peu adaptés aux données semi-structurées (documents), non structurées (binaires, multimedia) ou encore de structure non tabulaire (graphes, sémantiques, géographiques).

- **Rigidité des structures de données.** Modifier le schéma d'une base de données relationnelle pour l'adapter à de nouveaux besoins est une opération lourde et coûteuse, exigeant souvent la mise hors ligne de la base de données.
- **Lourdeur du langage SQL.** Ce langage, par sa richesse et son indépendance physique, est jugé trop lourd et trop coûteux pour des besoins d'accès simple : insérer, lire, modifier, supprimer un enregistrement.
- **Vulnérabilité des systèmes centralisés.** SGBD et bases de données relationnels sont vulnérables vis-à-vis des diverses sources de défaillances. Une panne d'un composant entraîne généralement l'arrêt du système (*single point failure*).
- **Extensibilité¹ limitée.** L'augmentation du volume des données et du nombre des utilisateurs exige des composants plus puissants, donc plus coûteux (processeurs plus puissants et plus nombreux, disques de plus grande capacité et plus rapides, plus de RAM). L'extensibilité est coûteuse et atteint rapidement des limites techniques.
- **Inadaptation aux très gros volumes de données distribuées.** La gestion efficace de plusieurs centaines de To, voire de Po², n'est pas à la portée des SGBD relationnels classiques. En outre, si ces SGBD peuvent s'accommoder d'un certain degré de distribution des données, ils ne sont pas conçus pour gérer plusieurs milliers de sites de données.
- **Inadaptation du modèle de transaction ACID.** Ce modèle, qui garantit les propriétés ACID, n'est plus adapté aux données massivement répliquées et fragmentées, et utilisées simultanément par plusieurs millions d'utilisateurs. Des modèles plus tolérants sont nécessaires afin d'assurer un maximum de fluidité.
- **Coût de la transparence.** Les SGBD relationnels assurent la *transparence* des applications vis-à-vis des caractéristiques techniques des données et de leur gestion. L'optimisation des requêtes (utilisation des index par exemple), la localisation des données, la fragmentation, la réplication, la gestion des transactions, la gestion de l'intégrité sont assurées de manière automatique par le SGBD. Tout ceci a un coût important en ressources machine et en manque de souplesse.

1. Traduction du terme anglais *scalability*. On distingue l'*extensibilité verticale*, réputée limitée et coûteuse (cf. les machines de bases de données, section 4.4.5), qui consiste à ajouter des composants hardware plus puissants à la configuration centralisée courante, de l'*extensibilité horizontale*, basée sur l'ajout de serveurs peu coûteux supportant chacun un fragment supplémentaire des données. Cette dernière organisation requiert une gestion optimisée de la répartition des données et des requêtes. C'est le rôle essentiel des SGBD NoSQL.

2. 1 Po = 1 péta-octets, soit 10¹⁵ octets ou encore mille téra-octets.

Les griefs envers les bases de données relationnelles ne sont pas nouveaux. Ils ont, au cours des décennies précédentes, donné naissance à des propositions souvent originales telles que les bases d'objets complexes (tables en N1FN), les bases de données déductives, distribuées, géographiques, documentaires, temporelles, réactives, orientées objet, relationnelles-objet, XML, d'entrepôts de données, multi-media, de graphes, de capteurs, mobiles, et aujourd'hui **NoSQL**. La plupart de ces propositions ont été (ou vont être), selon leur degré de succès, intégrées en tout ou en partie à la technologie SQL.

Si certaines des critiques évoquées ci-dessus sont parfois discutables, il faut reconnaître que le contexte actuel présente des caractéristiques tout à fait nouvelles qui justifient les autres points. Nous en citerons deux :

- Le nécessité commerciale de traiter de très gros volumes de données variées (plusieurs Po) massivement distribuées (plusieurs milliers de serveurs), c'est-à-dire fragmentées et répliquée à très grande échelle, en croissance très rapide (plusieurs To par jour) et accessibles par de très nombreux utilisateurs (plusieurs millions en parallèle).³
- Le développement de SGBD alternatifs adaptés à ce contexte, non pas par des équipes universitaires ou entreprises informatiques, mais par de très grandes entreprises internationales disposant d'équipes et de budgets de R&D colossaux. Ces entreprises peuvent se permettre de développer leur propre solution sans se préoccuper de standardisation ou de recherche d'une clientèle. Les SGBD ont souvent au départ un seul client comme Google, Amazon et Facebook.

C'est ainsi qu'on a vu se développer dans les années 2000 les premiers prototypes de ces nouveaux SGBD, tels que *Bigtable* chez Google et *Dynamo* chez Amazon. Leurs caractéristiques et leurs performances ont suscité une vague de développement de ces gestionnaires de données dits NoSQL.

Dans la suite de ce chapitre, nous étudierons succinctement les principales caractéristiques des modèles de données et de distribution des SGBD NoSQL ainsi que quelques détails de deux d'entre eux, Cassandra et MongoDB.

10.2 LES BASES DE DONNÉES NoSQL

On désigne par ce terme un ensemble de SGBD de facture récente (moins de 10 ans pour la plupart) d'une très grande variété, le plus souvent Open Source, et ayant comme trait commun l'abandon du modèle relationnel des données. Ils ont pour objectif la gestion et l'exploitation efficace de bases de données très volumineuses massivement distribuées (en fragmentation et réplication). Ils supportent un passage à l'échelle linéaire sans perte de performances, par ajout, généralement à chaud, de nouveaux fragments de la base de données. Leurs modèles de données sont en général très simples, voire primitifs, se limitant à des agrégats de données de structure plus ou moins complexe mais indépendants les uns des autres. Les bases de données sont dites sans schéma (*schemaless* ou *schema-free*). Le langage de mani-

3. Les 3V : Volume, Vitesse, Variété.

pulation sont également très simples et assez proches des structures physiques. Ils abandonnent le modèle de transaction strict ACID au profit d'une gestion plus souple tolérant certaines incohérences de lecture et d'écriture.

Si les SGBD NoSQL peuvent supporter certaines opérations de gestion opérationnelle, leur domaine d'utilisation privilégié est celui de l'analyse de vastes collections de données distribuées dans la perspective d'*aide à la décision*, domaine aussi appelé **BigData**⁴.

La structure d'une base de données NoSQL est conçue pour favoriser certaines requêtes au détriment d'autres. Elle est par conséquent dédiée à une application. Il s'agit d'une différence importante avec les bases de données relationnelles normalisées qui, par construction, doivent satisfaire les besoins en données de toutes les applications d'une entreprise ou d'un de ses départements.

La littérature classe les SGBD NoSQL en quatre catégories selon leur modèle de données : *clé-valeur*, *orienté documents*, *orienté colonnes* et *graphe*. Nous décrirons brièvement les trois premiers, le dernier relevant plutôt du domaine des bases de données spécialisées, au même titre que les bases de données spatiales ou textuelles.

10.2.1 Modèles clé-valeur

Les modèles *clé-valeur* constituent de simples mécanismes de stockage d'enregistrements indépendants non structurés, identifiés par une valeur de **clé** de nature quelconque. Traduit en SQL, le schéma d'une base de données *clé-valeur* quelconque se présenterait ainsi⁵ :

```
create table RECORD(Cle varchar(2048) not null primary key,  
                    Valeur blob);
```

L'ensemble des enregistrements peut parfois être partitionné selon les intervalles des valeurs de clé. La clé correspond à un index primaire généralement d'organisation calculée. L'interprétation du contenu du champ **Valeur** est de la responsabilité du programme d'application : texte libre, liste de valeurs, objet sérialisé en XML, YAML ou JSON. Les primitives de manipulation sont réduites : *get*, *put* et *delete*. Elles s'appliquent à l'enregistrement dont on spécifie la valeur de **Cle**. Certains systèmes admettent l'accès séquentiel aux enregistrements (*scan*). Parmi des SGBD apparentés à ce modèle, citons *Redis*, *Memcached*, *DynamoDB*, *Riak*, *Voldemort* et *Berkeley DB*, dont une variante est à la base d'*Oracle NoSQL*. Le site db-engines.com⁶ répertorie **53** SGBD du type *clé-valeur*.

4. Traduction française usuelle mais approximative de l'anglais *BigData*.

5. Les définitions SQL de cette section ont uniquement pour but d'illustrer dans un formalisme commun les constructions admises par les modèles NoSQL. Comme nous le verrons plus loin, chaque SGBD possède ses propres concepts et son propre langage de définition.

6. Le site <http://db-engines.com/en/ranking> répertorie les SGBD qui font l'objet d'échanges dans des forums ou sur les réseaux sociaux. Ces SGBD sont classés par popularité. Les chiffres cités ici datent de février 2018. Ce site répertorie **123** SGBD relationnels.

10.2.2 Modèles orientés documents

Ces modèles dérivent du modèle *clé-valeur* dans lequel le composant *valeur* serait la représentation **explicite**, en JSON ou en XML par exemple, d'un document structuré de complexité quelconque. Un document est constitué d'un agrégat de champs, où chaque champ est représenté par son nom et sa valeur. Une valeur peut être une constante élémentaire (chaîne de caractères ou nombre par exemple), un agrégat de champs, une liste de constantes ou une liste d'agrégats de champs. La description d'un document décrivant des commandes et leurs détails pourrait être la suivante, si elle devait être exprimée en SQL :

```
create table COMMANDE(
    NCOM      varchar(12) not null primary key,
    CLIENT    row(NCLI      char(10),
                  NOM        varchar(32),
                  ADRESSE     varchar(60),
                  LOCALITE    varchar(30)),
    DETAILS   row(QCOM      decimal(8),
                  PRODUIT    row(NPRO      char(15),
                                LIBELLE    varchar(60),
                                PRIX       decimal(6))) array);
```

Cependant, le nom et la structure des champs sont définis **dans chaque document**, ce qui conduit à deux caractéristiques importantes :

- il n'existe pas de structure de référence commune aux documents, donc pas de schéma au sens traditionnel du terme (la représentation SQL ci-dessus est donc illustrative mais pas équivalente),
- deux documents peuvent avoir des structures différentes.

Le fait que la structure du document soit explicite permet au SGBD d'offrir un langage de requête relativement précis exploitant cette structure. En revanche, le programme d'application est chargé d'analyser et exploiter la structure de chaque document en particulier. Il peut pour ce faire utiliser les bibliothèques de classes et de fonctions associées à JSON et XML ainsi que les langages d'analyse et de transformation tels que XQuery, XSLT ou JSONiq.

Parmi des SGBD orientés documents, citons *MongoDB*, *CouchDB*, *RavenDB*, *Azure* (Microsoft), *AsterixDB* (sic !), *Oracle NoSQL* et *Datastore* (Google). Le site db-engines.com répertorie **39** SGBD de ce type.

10.2.3 Modèles orientés colonnes

Le modèle *orienté colonnes* consiste en un ensemble dynamique de colonnes. Une base de données comprend un ensemble de lignes, composées chacune d'une valeur de **clé** et d'un ensemble de couples (**nom,valeur**), où **nom** est le nom d'une colonne et **valeur** la valeur assignée à cette colonne dans cette ligne. Lors de l'insertion d'une ligne, s'il n'existe pas encore de colonne de ce nom, elle s'ajoute à l'ensemble courant. Il n'existe pas en principe de schéma prédéfini pour les lignes, chacune définissant sa propre structure. Il est parfois possible de partitionner l'ensemble des lignes en sous-ensembles nommés. La base de données peut être perçue comme un

tableau à deux dimensions (colonne,ligne) de très grande taille dans lequel, à l'intersection d'une ligne et d'une colonne est inscrite la valeur de la colonne pour cette ligne. Certains SGBD admettent qu'une colonne soit composée de plusieurs valeurs (colonne composite). La description des lignes de la base de données pourrait être la suivante, si elle devait être exprimée en SQL :

```
create table LIGNES (
    CLE      integer not null primary key,
    ELEMENT row(NOM varchar(256) , VALEUR varchar(256)) array
);
```

Parmi des SGBD orientés colonnes, citons *Cassandra*, *Bigtable* (Google), *HBase*, *Accumulo*, *SimpleDB* (Amazon) et *Hypertable*. Le site db-engines.com répertorie 11 SGBD de ce type. Comme on le verra lors de l'analyse de *Cassandra*, ces SGBD ont tendance à se rallier au modèle SQL.

10.3 LES AUTRES MODÈLES NON RELATIONNELS

Les SGBD non relationnels ne se limitent pas aux familles NoSQL. Nous mentionnerons quelques modèles spécifiques pour lesquels il existe une offre diversifiée.

- **Les bases de données de graphes.** Ces SGBD permettent de représenter de manière naturelle des domaines qui se décrivent comme des graphes, c'est-à-dire un ensemble de noeuds connectés par un ensemble d'arcs. On cite comme exemple les relations entre personnes, les données des réseaux sociaux, les réseaux biochimiques, les données géographiques, les ontologies. On peut associer des attributs aux noeuds et aux arcs. Les langages associés sont de type navigationnel (à la manière de XPath) et possèdent souvent des fonctions récursives de parcours de graphe. Quelques représentants : *Neo4j*, *Titan*, *Sparksee*, *Giraph*, *OrientDB*. Les SGBD à modèle de graphes sont souvent classés dans la catégorie NoSQL, bien qu'ils supportent moins bien la fragmentation en raison de la forte connectivité des données.
- **Les bases de données RDF ou Triple store.** RDF est un langage de représentation de graphes conceptuels (le web sémantique et les ontologies par exemple) selon lequel toute information s'exprime sous la forme de classes d'objets, de relations d'inclusion entre classes et de triplets *objet-attribut-valeur* (123 et 234 désignent des instances de PERSONNE; 123 a pour nom 'Dupont'; 123 a pour conjoint 234, etc.). RDF est accompagné d'un système d'inférence permettant de dériver des informations de manière déclarative, à la manière du langage Prolog. Le W3C identifie 47 SGBD *Triple store* en avril 2015 (http://www.w3.org/2001/sw/wiki/Category:Triple_Store). Quelques représentants : *Triple-Graph Store* d'IBM, *Apache Jena*, *Sesame*, *Oracle Spatial*, *AllegroGraph*, *Owlim*, *RedStore*.
- **Les bases de données XML.** Le modèle de données est du type document XML, soit proche de celui des SGBD orientés document. Le langage de requêtes intègre les langages XPath de désignation d'éléments et de sous-structures, XQuery et

XSLT (langage de transformation), propre à la norme XML. Quelques représentants : *Sedna*, *BaseX*, *Tamino* (le premier véritablement commercialisé en 2000 par Software AG), *eXist-db*. La plupart des SGBD relationnels ont à présent intégré un module XML, permettant de gérer et manipuler des données XML à l'aide d'extensions du langage SQL.

- **Les bases de données orientés objet.** Conçus dans les années quatre-vingt dix, ces SGBD ont pour objectif de réduire la distance entre le modèle des langages de programmation et les modèles de bases de données. Les programmes d'application manipulent des objets, dont certains, stockés dans une base de données, sont persistants. Les langages de requêtes ont été développés, généralement comme extension de SQL (OQL par exemple) Quelques représentants : *Caché*, *Db4o*, *Versant Object Database*, *ObjectStore*, *Objectivity/DB*, *GemStone/S*. Les concepts issus de ces SGBD ont été intégrés dans la norme SQL3 et sont disponibles dans les principaux SGBD relationnels.

10.4 LES ARCHITECTURES DE DISTRIBUTION

Nous avons étudié dans la section 5.7.4 le protocole de `commit` en deux phases qui étend celui des transactions centralisées aux données distribuées en plusieurs fragments. On envisage à présent la question des **données répliquées**, dont nous avons vu qu'elles répondaient à des besoins de vitesse d'accès, de disponibilité et de résistance aux pannes (section 5.7.2). Précisons d'abord que l'ensemble des données se présente en toute généralité sous la forme d'un ou plusieurs **clusters**, chaque cluster étant constitué d'un ensemble de nœuds. Ceux-ci peuvent être localisés dans un même centre de données mais peuvent aussi être répartis géographiquement. Chaque nœud est constitué d'une machine et d'une instance du SGBD. Une machine est formée principalement d'un processeur (ou d'un coeur de processeur) et d'un groupe de disques. Les répliques et les fragments⁷ dont il est question dans ce chapitre sont localisés sur des nœuds distincts.

10.4.1 La cohérence finale

Un nouveau problème va se poser dans les configurations *non hiérarchiques asynchrones*. Lorsqu'une donnée est mise à jour sur un nœud, tous ses répliques doivent aussi être modifiés de manière que toutes les applications qui lisent cette donnée, quel qu'en soit le réplique, en reçoivent la dernière version. Cette propriété s'appelle la **cohérence (forte) de lecture**.⁸ Elle est, hélas, inaccessible à coût raisonnable dans la plupart des cas,⁹ et ce, pour deux raisons.

7. Dans le jargon des bases de données NoSQL, un fragment est appelé **shard**. Fragmenter une base de données ou lui ajouter un fragment s'appelle **sharding**.

8. Le terme peut prêter à confusion avec la propriété **C** de ACID. Il s'agit d'un tout autre concept.

9. Une des rares exceptions : VoltDB, qui garantit les propriétés ACID des transactions distribuées.

1. La propagation de la modification prend un certain temps, durant lequel, en principe, nul ne peut lire la donnée avant que tous les réplicas aient été mis à jour. Si tous les réplicas se trouvent sur des disques attachés à un même processeur, ce temps est négligeable, mais si ces réplicas sont nombreux et répartis géographiquement, ce temps peut être de plusieurs secondes voire de plusieurs minutes. Pendant ce temps, la donnée est indisponible, ce qui n'est pas acceptable.
2. Si, suite à une panne, un réplica est devenu inaccessible, il ne peut être mis à jour. La donnée est ici encore globalement indisponible durant un temps indéterminé.

Or, dans les applications visées par les SGBD NoSQL, la précision absolue n'est pas toujours exigée. Ne pas disposer des informations sur les dernières ventes d'un article n'est pas critique dans un traitement statistique. De même, dans un site de vente en ligne, afficher une quantité disponible d'un article qui date d'une minute ne constitue pas une catastrophe : *mieux vaut une donnée périmée* (ventes potentiellement problématiques) *que pas de données du tout* (ventes ratées). Si un problème survient avec un client, on le réglera matériellement, à l'ancienne.

Dans ce contexte, on introduit une variante faible de la cohérence associée à une modification d'une donnée : la **cohérence finale**. On dit que la donnée est **finale-ment cohérente** (*eventually consistent*) si on peut garantir l'existence d'un instant fini (le plus rapproché possible bien sûr) auquel tous les réplicas seront synchronisés. Avant cet instant, période qu'on appelle la **fenêtre d'incohérence**, il est possible que certains réplicas ne soient pas encore mis à jour.

10.4.2 Les protocoles à quorum

Puisque, pratiquement, on ne peut attendre cet instant, on doit accepter un compromis pendant la fenêtre d'incohérence. On adopte dès lors la politique de gestion suivante¹⁰, caractérisée par deux paramètres :

- **R**, qui spécifie le nombre minimum de réplicas à lire pour considérer qu'on connaît la vraie valeur d'une donnée ; en cas de divergence, on choisit la version dont la date de modification est la plus récente ;
- **W**, qui spécifie le nombre minimum de réplicas mis à jour pour admettre que la donnée a été mise à jour, et donc pour autoriser les lectures.

Soit **N** le nombre de réplicas d'une donnée **D**. Examinons quelques combinaisons de ces paramètres.

- $\mathbf{W + R > N}$: cohérence forte car au moins une version mise à jour de **D** sera lue.
- $\mathbf{W + R \leq N}$: cohérence faible ou finale; lecture de la dernière version de **D** non garantie.
- $\mathbf{W = N}$ et $\mathbf{R = 1}$: cohérence forte et vitesse de lecture maximale.
- $\mathbf{W = 1}$ et $\mathbf{R = N}$: cohérence forte et vitesse d'écriture maximale.

10. Sa description est détaillée dans de nombreuses références, dont la suivante : http://www.allthingsdistributed.com/2008/12/eventually_consistent.html, dont cet exposé s'inspire.

Le choix de ces paramètres dépend des performances exigées et du risque admis lors des lectures. Leurs valeurs peuvent être fixées statiquement pour la base de données, pour un fichier ou une table, ou dynamiquement par le programme d'application.

10.4.3 Le théorème CAP

Du fonctionnement d'une base de données distribuée, on peut attendre trois qualités de base :

- **Cohérence (C)** : pour une donnée **D**, tous les noeuds accessibles fournissent la même valeur de **D**.
- **Disponibilité (A, Availability)** : un noeud en fonctionnement renverra, dans un délai *raisonnable*, une réponse pertinente à tout message qui lui est envoyé.
- **Tolérance au partitionnement (P)** : en cas de partitionnement du réseau, chaque noeud doit continuer à fonctionner.

Le théorème dit CAP affirme qu'il n'est pas possible de garantir plus de deux de ces qualités pour une base de données distribuée. Par sa stratégie de gestion de la distribution, une base de données distribuée fonctionnera selon le type **CA**, **CP** ou **AP**. Considérant que la tolérance au partitionnement est généralement indispensable, il reste deux types de profils :

- **CP** : si une demande de lecture reçoit une réponse positive, alors la donnée qu'elle contient est la plus récente.
- **AP** : une demande de lecture reçoit toujours une réponse, même si la donnée qu'elle contient n'est pas la plus récente.

10.5 UN SGBD ORIENTÉ COLONNES : CASSANDRA

Cassandra est un des membres principaux de la famille *orientée colonnes*. Initialement développé par Facebook, il a été repris comme projet Open Source par la fondation Apache dès 2008. Cassandra gère des bases de données massivement distribuées dans plus de 1 500 entreprises de taille internationale, dont Apple, eBay, Instagram, Netflix¹¹ et ... Call of Duty, où il gère l'activité de 100 millions de joueurs.

Le modèle de données d'origine utilise un jeu de dénominations assez déroutant, très éloigné du vocabulaire habituel des bases de données. Par exemple, une base de données est appelée *keyspace*, une table *column family*, une valeur d'une colonne d'une ligne *column*, une colonne composite (de type *row*, selon SQL3) *super column* et un identifiant primaire (*primary key*) *rowkey*.

La manipulation de données se fait principalement via une API Java (d'autres langages sont admis) de bas niveau (l'interface Thrift). Les requêtes s'expriment de manière assez laborieuse en une suite de fonctions Java élémentaires. On a assisté depuis peu à une évolution importante de l'interface d'accès aux données qui

11. <http://cassandra.apache.org>. Selon ce site, Apple possède la plus grande infrastructure Cassandra, avec 75 000 noeuds et un volume de 10 Po.

s'oriente vers une variante du langage SQL dénommée **CQL** (*Cassandra Query Language*). Nous baserons ce qui suit sur la version 3 de CQL.¹² CQL s'appuie sur un modèle de données qui diffère substantiellement du modèle historique. Nous montrerons par quelques exemples certaines constructions spécifiques des modèles de données et du langage CQL.

a) Le modèle de données historique

La version actuelle de Cassandra abandonne le modèle historique et l'interface Thrift au profit d'une variante du modèle et du langage SQL. Nous n'avons donc plus repris dans cette édition la description du modèle historique.

b) Le modèle de données CQL

Le concept de base de données est appelé *keyspace*. Il contient des **tables** formées de **colonnes**. Une table possède un **identifiant primaire** (*primary key*) et au moins une autre colonne. La contrainte `not null` n'existe pas. Les types de base des colonnes sont similaires à ceux de SQL-92. En outre des types personnalisés peuvent être définis par l'utilisateur comme une suite de types.

La valeur d'une colonne du type **tuple** est une séquence de constantes. Chaque constante est d'un type de base qui lui est propre (cf. type `row` en SQL3).

La valeur d'une colonne du type *collection* est une collection de valeurs. CQL propose trois types de collections : `map`, `list` et `set`. La valeur d'une colonne **map** est un ensemble de couples *clé-valeur*, où *clé* est une chaîne de caractères et *valeur* une constante. Une colonne **map** se manipule syntaxiquement comme un dictionnaire Python et non comme une colonne composite. La valeur d'une colonne **list** est une liste de constantes et celle d'une colonne **set** est un ensemble de constantes.

La première colonne (ou les premières colonnes) de l'identifiant primaire constitue la **clé de partition** (*partition key*). La partition est une séquence de lignes stockées dans un même fichier dans l'ordre des valeurs des autres colonnes de l'identifiant primaire (les **clustering columns**). L'intégrité référentielle n'est pas gérée. Si une colonne est déclarée **statique**, sa valeur est identique pour toutes les lignes d'une partition.

Des **index secondaires** peuvent être définis chacun sur une seule colonne¹³.

c) Le langage CQL

Nous énumérerons ci-après les principales commandes CQL, accompagnées de leurs caractéristiques spécifiques et d'un exemple d'application.

– **create keyspace, alter keyspace, drop keyspace**

Créer, modifier et supprimer un espace de clés (une base de données). On précise notamment la stratégie de réplication.

```
create keyspace CLICOM;
```

12. <https://cassandra.apache.org/doc/latest/> pour la version 3.4 (février 2018)

13. Implémentés sous la forme d'index bitmap.

– **create type, alter type, drop type**

Créer, modifier et supprimer un type de colonne.

```
create type CONTACT (ADRESSE text, VILLE text);
```

– **use**

Rendre courant un espace de clés. Les commandes suivantes sont adressées par défaut à celui-ci.

```
use CLICOM;
```

– **create table, alter table, drop table**

Créer, modifier et supprimer une table. On spécifie ses colonnes, leur type ainsi que l'identifiant primaire (il n'y a pas de clés étrangères). On précise aussi le mode de stockage des lignes, leur ordre de rangement, ainsi que diverses autres propriétés de gestion. Il est possible d'ajouter et retirer des colonnes à tout instant.

```
create table DETAIL (NCOM text, NPRO text, QCOM decimal,
                    EXPEDITIONS map<timestamp, int>,
                    PAIEMENTS list<decimal>,
                    primary key ((NCOM), NPRO));
```

– **create index, drop index**

Créer et supprimer un index secondaire. Cet index peut être défini sur une colonne simple ou, pour une colonne *map*, sur le composant *clé* ou le composant *valeur*.

```
create index on COMMANDE (DATECOM);
```

– **truncate**

Supprimer les lignes d'une table.

```
truncate DETAIL;
```

– **create trigger, drop trigger**

Créer et supprimer un déclencheur. Celui-ci est de type *before* et réagit à toute modification de lignes de sa table. Le corps du déclencheur est une procédure Java.

```
create trigger TRG_PRO
on PRODUIT
using 'triggers.TRG_PRO';
```

– **select**

Variante simplifiée de la requête *select* de SQL2. Elle ne comprend pas de jointures ni de sous-requêtes. Les conditions de la clause *where* comparent chacune la valeur d'une colonne à une constante ou évaluent son appartenance à une liste de constantes. Ces conditions sont combinées à l'aide du seul connecteur *and*. Par défaut, chaque condition doit être évaluable via un index existant (préfixe de l'identifiant primaire ou index secondaire). Pour autoriser le filtrage sur une colonne non indexée, il faut spécifier l'option *allow filtering*.

```
select NCOM,NCLI
from   COMMANDE
where  DATECOM = '2015-12-23'
order by NCOM;
```

- insert

Variante simplifiée de la requête `insert` de SQL2. Si une ligne de même valeur de l'identifiant primaire existe déjà, cette commande se comporte comme un `update`, à moins que la clause `if not exists` ne soit spécifiée.

```
insert into DETAIL (NCOM,NPRO,QCOM,EXPEDITIONS,PAIEMENTS)
values ('30179','PA60',20,
       {'2016-01-18':16,'2016-01-18':4},
       [1520,380]);
```

- update

Variante simplifiée de la requête `update` de SQL2. Modifie la valeur de certaines colonnes de la ligne spécifiée par la valeur de son identifiant primaire. Si cette ligne n'existe pas, cette commande se comporte comme un `insert`. Il est possible d'ajouter et de retirer un élément à une colonne du type `collection`.

```
update DETAIL
set   QCOM = 25,
      EXPEDITIONS['2016-01-18'] = 15,
      EXPEDITION = EXPEDITIONS + {'2016-01-25':6},
      PAIEMENTS = PAIEMENTS + [450]
where NCOM = '30179' and NPRO = 'PA60';
```

- delete

Variante de la requête `delete` de SQL2. Supprime certaines colonnes des lignes spécifiées par une condition similaire à celle du `select`. Si aucun nom de colonne n'est spécifié, supprime les lignes sélectionnées.

```
delete EXPEDITIONS,PAIEMENTS from DETAIL
where  NCOM = '30179' and NPRO = 'PA60';
```

- begin batch ... apply batch

Définit une séquence de requêtes de modification des données traitées comme une transaction. L'atomicité est garantie mais l'isolation ne l'est qu'au sein d'une partition.

```
begin batch;
  delete from DETAIL   where NCOM = '30179';
  delete from COMMANDE where NCOM = '30179';
apply batch;
```

L'introduction de la commande CQL-DDL `create table`, qui spécifie les colonnes d'une table, a pour conséquence de réintroduire le concept de schéma, dont l'absence assumée constituait une des caractéristiques remarquables de la version historique de Cassandra. On doit sans doute y voir un signe de maturité. CQL est accessible notamment à partir de Java, Python et C++. La documentation technique sur Cassandra est assez pauvre et incomplète. Principales références : le site <https://cassandra.apache.org> et [Hewitt, 2016].

d) Le modèle de distribution

Le modèle de distribution est *non hiérarchique* (pair-à-pair). Il admet la fragmentation et la réplication. Le facteur de réplication (nombre de répliques) est fixé pour l'espace de clés. Cassandra implémente un protocole à quorum pour les lectures et les écritures.

10.6 UN SGBD ORIENTÉ DOCUMENTS : MONGODB

Considéré comme le SGBD NoSQL le plus populaire selon le site db-engines.com/en/ranking¹⁴, MongoDB est le premier représentant des SGBD orientés documents. Il a été développé par la société 10gen et est diffusé depuis 2009.

a) Le modèle de données MongoDB

L'unité de données est le **document**. Les documents similaires sont regroupés dans une **collection** et un ensemble de collections forme une **base de données** MongoDB. Un serveur MongoDB peut gérer plusieurs bases de données.

Un document est décrit dans une notation s'inspirant de JSON¹⁵ (ou plutôt BSON, sa variante binaire). Il est constitué de champs définis chacun par la forme **nom: valeur**, où **nom** est le nom du champ et **valeur** la valeur assignée à ce champ dans le document. Il n'existe pas de répertoire des noms des champs d'une collection. Chaque document définit sa propre structure, ce qui caractérise les *bases de données sans schéma*. Deux documents d'une même collection peuvent avoir des structures différentes.

Un champ de nom **_id**, obligatoire, est l'identifiant technique du document dans sa collection. Si sa valeur (un entier de 12 octets) n'est pas fournie lors de l'enregistrement du document, MongoDB lui en assigne une automatiquement. Par souci de simplicité, nous ne le mentionnerons pas dans les exemples suivants.

Un **document simple** est défini par une liste entre accolades de champs dont les valeurs sont des nombres, des chaînes de caractères, des dates, etc. :

```
{_id:948537,
  NCLI: 'C400',
  NOM: 'FERARD',
  ADRESSE: '65,rue du Tertre',
  LOCALITE: 'Poitiers',
  COMPTE: 350
}
```

14. Par ordre décroissant en février 2018 : Oracle (1303), MySQL (1252), SQL Server (1122), PostgreSQL (388), MongoDB (336), DB2 (190), Access (130), Redis (127), Cassandra (123), SQLite (117). Ces chiffres représentent le trafic des échanges au sujet de ces SGBD et non le nombre d'installations. Sur ce dernier critère, c'est SQLite qui est en première position.

15. *JavaScript Object Notation*. Représentation textuelle parenthésée de données structurées de manière arborescente. Format spécifique au langage JavaScript mais largement utilisé dans d'autres contextes. Concurrent d'XML mais moins verbeux. Voir description dans l'annexe A10.

La valeur d'un champ peut être un **tableau de valeurs** :

```
{NOM: 'Mercier',
  PRENOMS: ['Anne', 'Claire', 'Cécile']}
}
```

ou encore un (sous-)document :

```
{NCOM: 30179,
  CLIENT: {NCLI: 'C400',
            NOM: 'FERARD',
            LOCALITE: 'Poitiers'
          },
  DATECOM: '2015-12-22'
}
```

voire même un **tableau de documents** :

```
{NCOM: 30179,
  NCLI: 'C400',
  DATECOM: '2015-12-22',
  DETAILS: [{NPRO: 'CS262', QCOM: 60},
             {NPRO: 'PA60', QCOM: 20}]
}
```

Cette définition étant récursive, on peut construire des documents de complexité quelconque. Lors de l'enregistrement d'un document, le SGBD renvoie la valeur de son identifiant `_id`. Celle-ci peut être assignée à un champ d'un autre document, de manière à créer un lien entre documents.

Il est possible de créer des index sur les documents d'une collection. Les composants d'un index sont des champs simples, des champs de sous-documents ou des éléments de tableaux. Un index peut être déclaré identifiant, ce qui permet de déclarer indirectement des identifiants dans une collection. Un index peut être partiel. Il existe deux types d'index spéciaux : *textuel*, pour accélérer la recherche dans les textes et *spatial*, pour rechercher des données localisées spatialement.

b) Le langage MongoDB

Le langage hôte natif de MongoDB est JavaScript, auquel il emprunte la notation de description des documents. Des pilotes spécifiques permettent cependant d'accéder aux bases de données à partir de la plupart des autres langages de programmation. Une opération MongoDB est décrite par un document JSON.¹⁶ Les exemples ci-dessous sont exprimés en Python, via le module pilote PyMongo. Les noms des objets de la base de données sont représentés par des constantes *string* Python entou-

16. Ce document fournit une sorte d'arbre syntaxique qui résulterait de la compilation d'une requête SQL équivalente. Il n'est pas interdit de rêver à un *MQL* (MongoDB Query Language), variante de SQL, qui permettrait d'abandonner progressivement le langage, disons, *ésotérique* actuel, à la manière de NIQL du concurrent CouchBase.

rées d'apostrophes ou de guillemets. Les arguments des fonctions sont le plus souvent des dictionnaires {} et des listes [].

Examinons via des exemples la forme des principales instructions MongoDB. On suppose qu'une connexion a été effectuée avec le serveur et que la base de données CLICOM a été ouverte. Celle-ci est représentée par l'objet **db**.

- **client = db.createCollection("CLIENT")**
Crée la collection CLIENT dans la base de données **db**. Cette collection est représentée par l'objet **client**.
- **client = db["CLIENT"]**
Accède à la collection existante CLIENT dans la base de données **db**. Cette collection est représentée par l'objet **client**.
- **id = client.insert({"NCOM": "C400", ... })**
Insère un document dans la collection CLIENT. Le document est représenté par sa structure JSON. La méthode renvoie la valeur de l'**oid** du document créé.
- **client.remove({"CAT": "B2"})**
Supprime les documents qui satisfont la condition CAT = 'B2'.
- **client.update({"NCLI": "C400"}, {"\$set": {"CAT": "B2"}})**
Assigne la valeur 'B2' au champ CAT dans le document dont la valeur de NCLI est 'C400'.
- **cli = client.find_one({"NCLI": "C400"})**
Extraire un document dont on spécifie un champ identifiant. Contrairement à CQL, MongoDB n'exige pas que ce champ soit indexé.
- **cli = client.find_one({"NCLI": "C400"}, {"_id": 0, "NOM": 1, "LOCALITE": 1})**
Extraire certains champs d'un document. La première clause spécifie le critère de sélection et la seconde indique les champs à extraire (1) et ceux à ignorer (0).
- **clients = client.find({"LOCALITE": "Poitiers", "CAT": "C1"})**
for cli in clients:
print cli
Forme générale d'une extraction multiple. L'objet **clients** est un *curseur* désignant les documents de la collection CLIENT tels que LOCALITE = 'Poitiers' and CAT = 'C1'.
- **client.find({"\$or": [{"LOCALITE": "Poitiers"}, {"CAT": "B1"}]})**
Cette expression désigne les documents de la collection CLIENT tels que LOCALITE = 'Poitiers' or CAT = 'C1'. La condition **or** s'exprime par un champ prédéfini **\$or** dont la valeur est un tableau dont les éléments sont des conditions. Les patterns **\$and** et **\$or** peuvent être combinés.
- **client.find({"COMPTE": {"\$lt": 100}})**
Désigne les documents de la collection CLIENT dont la valeur du champ COMPTE est inférieure à 0. On a observé que la condition élémentaire **nom: valeur** s'interprète comme **nom = valeur**. Pour les autres opérateurs de comparaison,

on utilisera un document élémentaire dont le champ prédéfini `$lt` (*less than* ou $<$) a pour valeur le second terme de la comparaison. Autres opérateurs : `$lte` ($\leq$), `$gt` ($>$), `$gte` ($\geq$), `$ne` ($\neq$).

– `client.find({"CAT": "B1"}).count()`

Donne le nombre de documents de la collection CLIENT dont `CAT = 'B1'`. La méthode `count` agit comme un opérateur qui est appliqué au résultat de la méthode `find`. Cette structure est généralisable à d'autres opérateurs : `sort("LOCALITE")`, qui trie l'ensemble obtenu sur les valeurs de LOCALITE, `skip(2)`, qui ignore les 2 premiers documents, `limit(5)` qui renvoie les 5 premiers documents. On peut donc écrire :

```
client.find({"CAT": "B1"}).sort("LOCALITE").skip(2).limit(5)
```

MongoDB offre plusieurs méthodes d'agrégation similaires aux fonctions de SQL, depuis les fonctions simples telles que la méthode `count()` jusqu'à une implémentation de l'algorithme MapReduce.

c) Le modèle de distribution

Le modèle de **réplication** peut être qualifié de *hiérarchique asynchrone adaptatif*. Dans un cluster (mais aussi dans des clusters différents), tous les noeuds qui possèdent les mêmes données forment un *groupe de réplicas*. Parmi les noeuds de ce groupe, l'un est déclaré *primaire*, les autres étant *secondaires*. C'est au noeud primaire que s'adressent les demandes de modification des données. La modification est alors propagée automatiquement vers les noeuds secondaires. En revanche, les demandes de lecture peuvent être adressées à tout noeud du groupe. Lorsque le noeud primaire est défaillant (ne répond plus après 10 secondes), les noeuds secondaires élisent un nouveau noeud primaire. Les lectures adressées au noeud primaire jouissent d'une *cohérence forte* alors que les lectures adressées aux noeuds secondaires n'offrent qu'une *cohérence finale*.

La **distribution** est réalisée via une clé de fragmentation (*shard key*), qui est un champ commun à tous les documents de la collection à fragmenter. MongoDB assigne les sous-ensembles de documents aux différents noeuds selon des intervalles de valeurs de cette clé ou selon le résultat d'une fonction de hashing.

10.7 LES SGBD NewSQL : L'EXEMPLE DE VoltDB

Le terme *NewSQL* désigne une tendance récente (2011) d'architecture de SGBD cherchant à concilier les avantages *exclusifs* des systèmes NoSQL avec ceux des SGBD relationnels [Pavlo, 2016]. VoltDB, qui en est l'exemple le plus représentatif à l'heure actuelle, a été conçu par l'équipe de M. Stonebraker¹⁷.

17. Chercheur et entrepreneur renommé dans le domaine des SGBD. Créateur et architecte de plusieurs SGBD tels que Ingres et Postgres (ancêtre de PostgreSQL). A reçu en 2014 le *Turing Award*, une des plus hautes distinctions scientifiques en informatique.

VoltDB gère des bases de données *in-memory* (section 5.9) fragmentées et répliquées au travers d'un ou plusieurs *clusters*, comme dans les bases de données NoSQL. L'interaction avec les données se fait dans un sous-ensemble de SQL-92 via une interface du type JDBC. La base de données peut être dotée d'identifiants et d'index mais pas de clés étrangères, de prédicats *check* ni de déclencheurs. Les colonnes du type *varchar* peuvent être décodées comme des structures JSON.

Profitant de la vitesse d'accès aux données en mémoire centrale et du faible coût des processeurs, les concepteurs ont développé une architecture très dépouillée. Par exemple, chaque nœud d'un cluster est un cœur de processeur, doté d'une mémoire centrale dédiée, exécutant une instance de VoltDB, prenant en charge une seule transaction à la fois et manipulant son propre fragment de données (architecture *share nothing*). On évite ainsi les processus de gestion de transactions concurrentes et la gestion des tampons, très consommateurs de ressources. Les propriétés ACID des transactions sont garanties comme dans les SGBD relationnels, y compris pour les transactions distribuées. La documentation commerciale de VoltDB avance, à configuration équivalente, des performances transactionnelles près de 50 fois supérieures à celles des SGBD relationnels classiques.

La résistance aux pannes est assurée par la réplication des fragments, les sauvegardes régulières sur disque et les journaux de commandes (beaucoup plus compacts que les journaux d'images). En cas d'incident le fragment est soit recopié d'un nœud du cluster soit reconstruit à partir de la sauvegarde et du journal local.

10.8 CONCLUSIONS SUR LES SGBD NOSQL

Les *nouveaux SGBD* ont été conçus pour répondre aux critiques adressées aux SGBD SQL dans des domaines d'application caractérisés par de gros volumes de données massivement distribuées, en croissance rapide, nécessitant une grande vitesse de traitement et présentant une grande variété de types de données (les 3 V).

Ils jouent sur différents facteurs : politiques efficaces de fragmentation et de réplication, simplicité et souplesse des modèles de données, simplicité des langages de manipulation de données, assouplissement des mécanismes de transaction. Il est intéressant de comparer les apports de ces SGBD non relationnels aux fonctions des SGBD SQL.

- *Modèles de données plus souples.* Les approches centrées sur les documents sont assurément plus représentatives du domaine d'application que celle des données relationnelles normalisées. Cependant, la comparaison est un peu courte. D'une part, les SGBD SQL offrent également cette possibilité, soit via les structures d'objets complexes (*row* et *array*) de SQL3, soit via les méthodes de tables typées, soit encore via le mécanisme de vues qui reconstituent dynamiquement ou statiquement (vues matérialisées) des agrégats non normalisés. D'autre part, la plupart des SGBD SQL intègrent actuellement des modules de gestion de données du type XML et JSON. Enfin, deux applications NoSQL différentes réclameront la livraison des mêmes données sous la forme de deux documents différents, ce qui n'est possible qu'en acceptant des redondances.

L'absence de schéma permet certainement une grande souplesse dans les structures de données. Le prix à payer est non négligeable : les métadonnées étant dispersées dans les données (voir JSON et XML par exemple), leur interprétation est à charge des programmes d'application. Par ailleurs, l'absence de schéma impliquant la faiblesse des contraintes d'intégrité, dont la gestion est également de la responsabilité des programmes d'application. On observe d'ailleurs une évolution progressive vers plus de rigueur, comme en témoigne par exemple CQL, ainsi que les SGBD récents, qui se réclament à nouveau de la norme SQL.

- *Langage de manipulation plus simples.* Ici encore, l'argument peut être discuté. Les langages (souvent d'ailleurs les API) propriétaires offrent peu d'avantage mais réclament chacun un apprentissage spécifique et produisent des programmes non transférables vers d'autres SGBD, plus délicats à maintenir. La comparaison a peu d'intérêt pour des requêtes simples. En revanche, on voit mal l'avantage que pourrait présenter un programme complexe dans un langage propriétaire par rapport à une requête, certes complexe elle aussi, mais rédigée dans un langage standard, déclaratif, concis, expressif et beaucoup plus puissant. Est-il préférable de programmer de manière procédurale des jointures dans un langage incomplet plutôt que les exprimer simplement et explicitement dans une requête SQL ? S'il est vrai qu'un SGBD SQL consomme plus de ressources pour interpréter une requête qu'un SGBD NoSQL, il n'est pas prouvé que cet aspect joue de façon significative sur les performances d'exécution de cette requête, lesquelles dépendent surtout du parallélisme d'exécution et de l'accès aux données en mémoire centrale (larges tampons et bases de données *in-memory*).
- *Gestion décentralisée de gros volumes de données.* La souplesse des méthodes d'extension des données constitue le point fort des nouveaux SGBD. L'ajout dynamique d'un noeud ou d'un cluster et la redistribution dynamique des données permet une croissance en souplesse de la base de données qui n'est pas pour l'instant à la portée des SGBD relationnels, qui appliquent une gestion fortement centralisée. Les données massivement distribuées se caractérisent par un haut taux de fragmentation et de réplication, notamment sur une base géographique, qui réclame une gestion très spécifique des opérations d'écriture et de lecture. Cette gestion entraîne, dans certaines configurations, l'assouplissement des contraintes des transactions ACID propres aux données centralisées.
- *Résistance aux défaillances.* Le problème des défaillances devient critique dès lors que la base de données est distribuée sur des milliers de noeuds. Les pannes ne sont plus des accidents redoutés mais relèvent du comportement normal du système. Ici encore, les SGBD SQL ne sont pas idéalement armés pour gérer en souplesse la protection de telles données.

Les SGBD SQL et les SGBD NoSQL ont des domaines d'application différents. Les premiers sont généralistes, adaptés à la résolution d'une large gamme de problèmes et offrant un contrôle strict de l'intégrité des données. Ils sont particulièrement adaptés aux applications transactionnelles (dites OLTP¹⁸). Une base de données SQL doit généralement satisfaire les besoins en données de nombreux programmes d'application, d'où la nécessité d'un schéma de données neutre. SQL permet une

écriture de programmes indépendante des structures techniques des données et des chemins d'accès optimaux. Une base de données NoSQL sera en général taillée sur mesure pour une application particulière peu complexe ayant des exigences fortes en matière de performances, de volume et de distribution de données. Ces applications seront souvent destinées à écrire et extraire des agrégats de données ou à extraire des statistiques de données volumineuse et distribuées.¹⁹

On observe cependant une convergence des deux côtés. D'une part, certains SGBD NoSQL intègrent progressivement une variante du langage SQL²⁰ et offrent, dans certaines configuration de données, une gestion de transactions ACID. D'autre part, les principaux éditeurs de SGBD relationnels incluent dans leur offre des fonctionnalités de gestion de données distribuées. On citera par exemple Oracle RAC (plus ancien), Oracle NoSQL (BerkleyDB surmonté d'une couche JSON), MySQL Cluster, DB2-MongoDB, AzureDB de Microsoft.

10.9 LES CHAÎNES DE BLOCS

Dit simplement, une *chaîne de blocs* (*blockchain*) est une base de données historique massivement répliquée et sécurisée dans laquelle sont enregistrées chronologiquement les descriptions d'opérations intéressant une communauté d'utilisateurs.

L'application privilégiée, d'ailleurs celle pour laquelle la technologie a été spécifiée en 2008, est celle d'un *registre* de transactions monétaires, exécutées par les membres de la communauté. Cependant, bien d'autres applications sont déjà en fonctionnement ou sont envisagées à court terme.

Les objectifs sont, notamment, ceux que réalisent les systèmes centralisés tels que les banques ou les administrations, mais *sans faire appels à ces tiers*. La technologie des chaînes de blocs doit satisfaire les conditions suivantes :

- Sécurité : les opérations décrites dans le registre sont définitivement effectives dès qu'elles ont subi un processus de validation.
- Infalsifiabilité : les données enregistrées ne peuvent plus être ultérieurement supprimées ni modifiées.
- Transparence : les données enregistrées sont publiques; tous les membres de la communauté peuvent en prendre connaissance et vérifier leur validité.
- Anonymat : l'identité des membres concernés par une opération est mentionnée via un *pseudonyme*.

18. On-Line Transaction Processing

19. Sous les dénominations d'*applications d'aide à la décision*, ou OLAP (On-Line Analytical Processing), ou encore, selon un vocabulaire plus actuel, BigData, Analytics ou Business Intelligence.

20. Les manuels de programmation de ces SGBD s'étendant longuement sur les règles de conversion des requêtes SQL dans leur langage propriétaire (et inversement), il serait évidemment plus raisonnable à tout point de vue de développer un traducteur de requête SQL. Plusieurs offres de traducteurs SQL-MongoDB sont actuellement proposées.

Les objectifs sont réalisés par une gestion collégiale du registre et des techniques cryptographiques telles que celles qui ont été décrites à la section 4.16.

La technologie des chaînes de blocs est relativement complexe. Nous focaliserons cette section sur les structures de données et les techniques de gestion qui garantissent les objectifs énoncés ci-dessus.

10.9.1 Principes généraux

Considérons l'exemple de transactions monétaires dont le réseau BitCoin est l'exemple le plus ancien et le plus emblématique. Classiquement, la gestion de comptes bancaires comprend des opérations de plusieurs types tels que l'ouverture d'un compte, le dépôt d'une somme d'argent sur un compte à partir d'une source externe et le transfert d'une somme d'un compte vers un autre. Par l'usage d'une chaîne de blocs on se propose de réaliser ces opérations en dehors du contexte bancaire, c'est à dire sans l'intervention de ce tiers de confiance qu'est la banque.

Appelons **BC** cette chaîne de blocs qui se substitue à l'organisme bancaire habituel. Les personnes intéressées par son usage vont donc ouvrir un compte, y déposer de temps en temps des sommes émises par des sources externes (salaire, versement de montants en Euros par exemple) et transférer une somme vers un autre compte notamment pour effectuer un achat.

Nous dirons que ces personnes sont les *membres du réseau BC*. Pour réaliser une de ces opérations, qu'on appelle *transactions*, un membre va ajouter dans **BC** un enregistrement décrivant celle-ci sans ambiguïté. Pour rendre les choses plus concrètes, fixons comme suit la syntaxe des trois transactions de base :

- **OPER**=inscription; **CLI**=Jean : Jean, un nouveau membre, s'inscrit et ouvre un compte.
- **OPER**=dépôt; **CLI**=Jean; **SOMME**=2000 : Jean dépose une somme de 2000 sur son compte.
- **OPER**=transfert; **CLI**=Jean; **DEST**=Marie; **SOMME**=150 : Jean transfère une somme de 150 sur le compte de Marie.

Selon le principe de transparence, chaque membre du réseau possède une copie de **BC**, de sorte que chacun peut *vérifier la validité* des transactions. En particulier pour le dernier exemple de transaction,

- Jean possède-t-il un compte ?
- Marie possède-t-elle un compte ?
- le solde du compte de Jean est-il d'au moins 150 ?

Cette dernière question peut être résolue par le parcours des transactions antérieures, qui nous permet de calculer le solde comme la différence entre la somme des entrées et la somme des sorties.

Pour des raisons que nous examinerons plus loin, les nouvelles transactions sont rassemblées sous la forme d'un **bloc** qui est ensuite diffusé auprès de tous les membres pour être ajouté à chaque copie de **BC**. Chaque bloc possède un identifiant dans **BC**. Il comprend aussi l'identifiant du dernier bloc précédemment ajouté. La structure de **BC** est donc celle d'une **chaîne de blocs**, chaque bloc contenant les

descriptions d’une série de transactions, comme illustré par la figure 10.1. Cette figure montre qu’un bloc est constitué de deux champs, **Précéd.**, l’identifiant du bloc précédent, et **Contenu**, la liste des transactions.

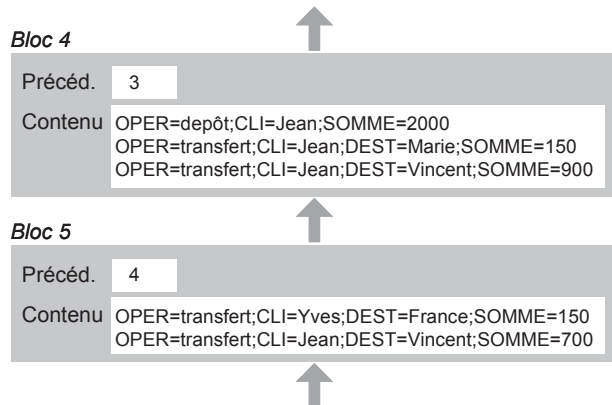


Figure 10.1 - Extrait d’une chaîne de blocs

a) Identification des blocs

Lorsqu’un bloc est créé, on lui assigne un identifiant. Cet identifiant pourrait être un simple numéro de série indiquant sa position dans l’ordre chronologique de création, comme le suggère la figure 10.1. Or, nous savons que la création d’identifiants globaux n’est pas un problème simple (section 4.16.3). On ne peut envisager sérieusement que les membres du réseau se concertent en permanence pour choisir un numéro non encore utilisé.

La solution adoptée par les protocoles de gestion les chaînes de blocs consiste à calculer la signature de l’ensemble du bloc par *hachage cryptographique*, typiquement SHA-256 (section 4.16.2). Cet identifiant, appelé le **hash** du bloc, apparaît comme un champ de celui-ci (figure 10.2).



Figure 10.2 - Identification des blocs par hachage cryptographique

Pour résumer, un bloc est à présent constitué de trois composants : le *hash* du bloc précédent, une liste de transactions et son identifiant (*hash*).²¹ Cet identifiant est calculé comme suit :

$\text{hash} = \text{SHA256}(\text{Précéd.} + \text{Contenu})$

Ainsi, l'identifiant du bloc 5, '6f1bfa...ef6da', est obtenu en calculant :

$\text{SHA256}('78919d...1e3e\text{OPER}=\text{transfert};\text{CLI}=\text{Yves}... \text{OPER}=\text{transfert};\text{CLI}=\text{Jean}...700')$

Cette technique d'identification induit une propriété très importante : elle rend **BC** difficilement falsifiable. Supposons qu'un membre désire modifier la description d'une transaction. Par exemple, Marie voudrait discrètement augmenter le montant du transfert que Jean a effectué sur son compte. Elle modifie donc le contenu du bloc 4 en changeant la somme 150 en 200 dans la 2^e transaction. Ce faisant, elle invalide le **hash** de ce bloc, qui doit être recalculé. À présent, le bloc 5 devient invalide à son tour, puisque le champ Précéd. devient incorrect. Marie doit donc modifier ce champ du bloc 5 puis recalculer son **hash**. Cette correction doit en réalité être appliquée à tous les blocs qui suivent le bloc falsifié, ce qui constitue un travail non négligeable et surtout fort peu discret.

b) Chiffrement des transactions

Il existe un niveau de sécurité supérieur qui interdit à Marie de falsifier les données d'un bloc : le **chiffrement** du texte de la transaction. Chaque membre du réseau dispose d'un couple de clés (**P**,**K**), nommées respectivement *clé privée* et *clé publique*. Ces clés sont des chaînes de bits générées à la demande par une fonction mathématique. **P** est générée aléatoirement et **K** est calculée à partir de **P** de telle manière qu'il est impossible de calculer **P** à partir de **K**. Il existe une fonction **ch**, dite de *chiffrement*, qui transforme tout message **M** selon une clé **P** quelconque en une suite de bits **S** d'apparence aléatoire :

$$S = \text{ch}(M, P)$$

Il existe une autre fonction **dech**, dite de *déchiffrement*, qui déchiffre toute chaîne **S** en produisant le message source **M** :

$$M = \text{dech}(S, K)$$

On a donc :

$$M = \text{dech}(\text{ch}(M, P), K)$$

La clé privée **P** est secrète et ne doit jamais être divulguée.²² En revanche, la clé publique **K** est ... publique, et connue de tous les membres du réseau. Le processus de génération des clés garantit que tant les clés privées que les clés publiques sont uniques dans le réseau. On peut donc utiliser la clé publique des membres au lieu de leur nom. Nous allons donc remplacer leur nom par leur clé publique, ce qui garantit leur anonymat.

21. Dans cet exemple et les suivants, les chaînes de bits sont visualisées en hexadécimal

22. Ni perdue, auquel cas les avoirs du membre sont irrémédiablement perdus.

Lorsqu'un membre désire créer une transaction **T**, il l'accompagne de son chiffre **S = ch(T,P)**, communément appelée *signature* du bloc.²³ Tout membre du réseau peut dès lors vérifier que la transaction en clair **T'** est bien authentique et que seul son émetteur a pu la créer et chiffrer. Si Marie trafique une donnée de **T**, cette fraude est immédiatement détectée puisque dans ce cas **T' ≠ dech(S,K)**. La figure 10.3 montre les deux transformations de la première transaction du bloc 5 : l'identification des deux membres se fait via leur clé publique et la transaction est complétée de sa signature calculée à l'aide de la clé privée d'Yves.

```
OPER=transfert;CLI=Yves;DEST=France;SOMME=150
```

```
OPER=transfert
CLI=66dda3d304c226d70462c89ef04bd4aff2c3ca5b2ec27d0cc9375b1a587590dc
DEST=ac51f7441c298d2dab97444eae0307ecc2d6b4483be354b082e07d426d8c19c1
SOMME=150
SIGN:c285c286c2a9c2b6c29ec2a7c396c294c29ec2a7c389c297c2a4c2aac2a7c28379716c6
8c387c29cc29ac298c38a6364c296c38766c293c29cc38a69c29367c299c293c298c29a6
bc38ac389626bc386c294c297c384c29fc29969c298c295c294c3826ac29a69c29ac29c6
2c29bc38766c299c387c29dc2946ac299c29561c295c2986a696bc29d67c294c2977a77
c2b6c28c76c386c3896565c388c29b68c295c297c38964c29c6bc387c293c299c383c29
4c29ec29a666bc298c295c384c3886966676cc387c294c38467c29c6dc2976d64c29cc29
6c298c29bc297c299c295c295c2946b62c299c29369c2966ac2966dc2946cc29963c29cc
29b6ac2b8c2b57dc281c2a7c2a165c296c296
```

Figure 10.3 - Traduction anonymisée et sécurisée d'une transaction

L'architecture que nous venons de définir permet

- de préserver l'anonymat des partenaires des transactions
- à chacun d'examiner toutes les transactions de la chaîne
- d'authentifier chaque transaction
- de garantir que le contenu d'un bloc n'a pas été falsifié
- de vérifier l'état du compte de chaque membre du réseau

Il reste cependant bien des questions à résoudre, dont les suivantes, que nous étudierons dans la section suivante :

- comment et par qui sont assemblées les transactions pour former les blocs ?
- comment et par qui la validité des transactions est-elle vérifiée ?
- comment calculer efficacement le solde d'un compte.

Remarque.

Le processus de chiffrement présente certaines analogies avec le hachage cryptographique de *M*. Il existe cependant une différence fondamentale entre ces deux techniques : le hachage de *M* est **destructif** alors que le chiffrement **préserve** son contenu.

23. À ne pas confondre avec le résultat d'un hachage cryptographique, que nous avons également appelé *signature*.

10.9.2 Le protocole BitCoin

Conçu en 2008 par Satoshi Nakamoto²⁴, le système de monnaie virtuelle, ou *crypto-monnaie*, *BitCoin* est la plus connue des applications des chaînes de blocs, au point que les deux concepts se confondent souvent dans la littérature. L'unité monétaire est le *bitcoin* (en minuscules).

a) Les adresses BitCoin

Dans le protocole BitCoin, l'identifiant d'un membre, ce qu'on appelle son *adresse BitCoin*, n'est pas exactement la clé publique, mais une chaîne de caractères dérivée de celle-ci, plus courte et plus facile à manipuler. Elle est obtenue par double hachage cryptographique de la clé publique, produisant une chaîne de 160 bits, suivi d'une transformation en 34 caractères ASCII. Cette adresse peut être convertie en QR code utilisable pour des paiements mobiles

b) Les transactions BitCoin

Le transfert d'une somme **S** à partir du compte (d'un membre) **A** s'effectue par une transaction **T** selon les principes suivants, plus élaborés que ceux que nous avons décrits ci-dessus :

- Une transaction de transfert distribue la somme **S** vers un ou plusieurs destinataires; chaque partie de **S** transmise à un destinataire est appelée une *sortie* (*output*) de la transaction.
- La somme **S** est constituée des sorties d'une ou plusieurs transactions dont **A** est le destinataire. Chacune de ces transactions constitue une *entrée* (*input*) de **T**. Soit **S'** le total des sorties de ces transactions. La transaction **T** est valide si $S \leq S'$.
- Si $S < S'$, la différence est retournée à **A**. Ainsi, la somme **S'** est entièrement distribuée par la transaction.
- La spécification dans **T** des *inputs* nécessaires à la constitution de **S** évite l'examen de toutes les transactions qui précèdent **T**. Il suffit d'identifier les *outputs* des transactions antérieures qui n'ont pas encore été consommées.

La figure 10.4 illustre le mécanisme d'une transaction **T13** par laquelle **Jean** verse une somme de **220** bitcoins à **Albert** (pour des raisons de clarté, on utilise à nouveau le nom des membres plutôt que leurs adresses BitCoin). Pour réunir cette somme, **Jean** va se servir d'*outputs* non encore consommés des transactions antérieures dont il est le destinataire, en l'occurrence **T3**, **T4** et **T6**. Parmi celles-ci, il choisit comme *inputs* à sa transaction **T3** et **T6**, qui lui apportent un montant de **250**. Comme le versement laissera un solde de **30** bitcoins, ceux-ci lui sont reversés. Lorsque cette transaction sera validée et définitivement inscrite dans la chaîne de blocs, l'avoir de **Jean** sera de **65** (de T4.out1) + **30** (de T13.out2), soit **95** bitcoins, disponibles pour ses futures transactions.

24. Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System* [<https://bitcoin.org/bitcoin.pdf>]. Le nom de l'auteur semble être le pseudonyme d'un individu ou d'un groupe dont l'identité réelle reste inconnue à ce jour.

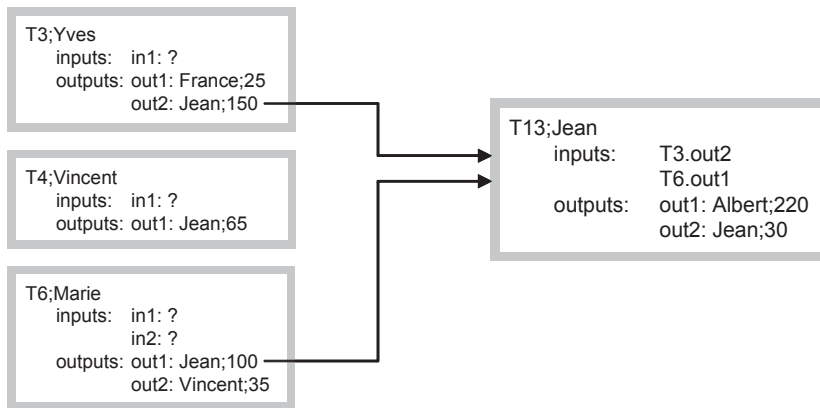


Figure 10.4 - Structure simplifiée d'une transaction BitCoin (syntaxe fictive)

c) Les mineurs

Dès qu'une transaction a été complétée par sa signature, l'émetteur l'envoie à tous les membres du réseau.

Parmi ceux-ci, certains se sont proposés pour évaluer la validité des transactions. On les appellent les *mineurs*, pour une raison que nous allons préciser. Un mineur joue plusieurs rôles :

- Il vérifie la validité de chaque transaction reçue : existence de l'émetteur et des destinataires (identifiés par leur adresse BitCoin), disponibilité des inputs nécessaires aux sommes distribuées, authenticité de la transaction au moyen de la signature et de la clé publique de l'émetteur.
- Lorsqu'il a reçu un certain nombre de transactions qu'il a validées, il forme un bloc qui référence le dernier de la chaîne puis il y inscrit le *hash* de ce bloc ainsi que la date courante.
- Il diffuse ce bloc dans le réseau pour le proposer aux autres membres comme nouveau bloc *officiel*.

Ces opérations sont assez simples et peuvent être exécutées en quelques millisecondes. Les mineurs vont donc injecter dans le réseau, quasiment en même temps, des centaines de blocs candidats. Il est impossible dans un tel cas de choisir celui qui sera le nouveau bloc officiel.²⁵ Pour répartir les arrivées des blocs candidats sur une période plus longue, le protocole BitCoin va demander aux mineurs d'exécuter un certain travail supplémentaire, d'une durée aléatoire, lors de la constitution du bloc. Il s'agit d'imposer au *hash* du bloc une contrainte de forme qui rende son calcul plus long. Le mineur doit trouver un complément (appelé *nonce* dans le vocabulaire BitCoin) à ajouter au bloc, tel que la valeur du *hash* calculée sur l'ensemble soit inférieure à une valeur v fixée pour tous. Cette contrainte revient à imposer que les k premiers bits du *hash* soient 0.²⁶

25. En général, deux mineurs construiront des blocs différents.

Le pseudocode ci-dessous recherche le premier nombre qui, ajouté à la chaîne de bits du bloc, produit un hash dont les 26 premiers bits sont 0.

```
nonce = 0
while True
    hash = SHA256(bloc + nonce)
    if hash.startswith(b'0000000000000000000000000000') exit
    nonce += 1
endwhile
```

Une autre technique consiste à générer des valeurs aléatoires de ce complément. Dès qu’une valeur acceptable du *nonce* a été trouvée, le mineur l’ajoute au bloc, au titre de *preuve de travail*, de même que la valeur du *hash* puis diffuse le bloc ainsi complété (figure 10.5).

La recherche d’un *hash* valide consomme un temps de calcul considérable²⁷, d’autant plus que la valeur de *k* augmente régulièrement et exige donc une puissance de calcul de plus en plus importante. Cette puissance est actuellement fournie par des machines spécialisées composées de processeurs graphiques²⁸, voire même de circuits spécialisés dans le calcul des *hashes* (*ASIC*).

Précéd.	268c47caa3aa22de42835f0d8e9d5cc2a033dbac989a9e15cff6bf6a84320a25
Contenu	<Transaction t35> <Transaction t36> <Transaction t37>
nonce	74f02ca6
Date	2018-09-12 16:37:04.345
Hash	0000001e677caf780c4a096b929dd4fa9273f1b7e54a8824cc86b37801011e3e

Figure 10.5 - Un bloc valide pour k = 26 et sa *preuve de travail*

d) Rémunération des mineurs

Quelle motivation un mineur peut-il trouver dans l’exécution une telle corvée ? C’est simple : *l’intérêt financier*. Le premier mineur qui valide un bloc reçoit, en contrepartie de son travail, une rémunération en bitcoins. Concrètement, le mineur qui valide un bloc est autorisé à y ajouter une transaction spéciale, dite *coinbase*, qui crée et lui transfère une somme en bitcoins, selon un tarif imposé (traduction BitCoin de notre transaction OPER=dépôt de la section 10.9.1). Pour tenir compte de la croissance régulière de la puissance de calcul des mineurs, ce tarif est divisé par 2 tous les quatre ans. Il est en 2018 de 12,5 bitcoins.

26. En 2017, $k = 72$
27. Évalué à 2^k calculs de *hash* en moyenne. Les paramètres du protocole, tels que *k*, sont prévus pour que la recherche d’un *hash* valide prenne en moyenne 10 minutes, quelle que soit la puissance de calcul du réseau.
28. Les GPU sont plus simples et moins chers mais, dotés de plusieurs centaines de coeurs, sont beaucoup plus rapides que les CPU génériques pour ce type de calcul. Au grand dam des amateurs de jeux vidéos, qui sont confrontés à une pénurie de cartes graphiques haut de gamme.

D'autre part, l'émetteur d'une transaction peut (sans obligation, mais les choses vont sans doute changer) accorder une petite rémunération au mineur. Ceci peut influencer sur le choix des transactions à valider en priorité.

On sait que toute nouvelle technologie, dès qu'elle émerge, donne lieu à des usages, disons, ... *marginaux*, non prévus par leur créateur, et ce depuis la nuit des temps. Les chaînes de blocs n'échappent pas à cette malédiction. On citera par exemple le *cryptojacking*, par lequel des malicieux s'installent discrètement sur des machines privées et les forcent à miner pour le compte de pirates.²⁹

e) Résolution des conflits

Si le bloc validé du mineur le plus rapide est reçu en premier par tous les membres du réseau, la question est réglée : ce bloc devient officiellement le dernier bloc de la chaîne. Il peut cependant arriver que plusieurs blocs candidats soient envoyés presque simultanément, de sorte que tous les membres ne reçoivent pas en premier le même bloc candidat. Dans ce cas, la chaîne présente au travers du réseau plusieurs branches concurrentes. Pour résoudre ce conflit, on laisse de nouveaux blocs s'ajouter aux branches, jusqu'au moment où une des branches devient plus longue que les autres. Ses blocs deviennent dès lors les composants officiels de la chaîne. Les blocs des branches abandonnées sont désagrégés et leurs transactions sont recyclées, c'est-à-dire soumises à nouveau au réseau pour évaluation et groupement par les mineurs selon la procédure décrite ci-avant.

Les mécanismes de validation, de minage et de résolution de conflits constituent le *mécanisme de consensus par preuve de travail*. Nous en examinerons un autre plus loin.

f) Limites du protocole

L'ensemble des règles à respecter, qui assurent un très haut niveau de sécurité, entraînent en revanche des limites au fonctionnement global du réseau, notamment la quantité limitée de bitcoins qu'il est possible de créer,³⁰ une vitesse de traitement globale limitée à 7 transactions par seconde, une taille de blocs limitée à 1 Mo et une importante consommation d'énergie électrique (10 minutes de calcul intensif par bloc candidat).

En outre, le protocole prévoit de réguler la création de bitcoins en diminuant progressivement la rémunération. Lorsque la création de bitcoins sera épuisée, il est envisagé de rémunérer le mineurs par une commission imposée à chaque transaction.

Le travail des mineurs devenant de plus en plus difficile, ceux-ci se regroupent en *pools de minage* au sein desquels chaque mineur contribue pour une certaine quantité de calculs de *hash*. Si un bloc validé par le pool est gagnant, la rémunération est partagée au prorata de la contribution de chacun. Il existe un risque pour un pool de

29. On citera *Coinhive*, minant pour la crypto-monnaie *Monero* à partir d'un navigateur. Selon l'éthique du site émetteur, l'utilisateur est consulté ou non sur l'exécution du minage.

30. Le nombre total de bitcoins qu'il est possible de créer est limité par le protocole à 21 millions.

devenir trop puissant et de rassembler plus de 50% de la puissance totale du réseau (scénario dit *des 51%*), créant ainsi un monopole qui ouvrirait la porte à une possibilité de falsification.³¹ Conscients de ce danger, les mineurs pratiquent une forme d'autocontrôle qui limite la taille de leur pool.

Le protocole BitCoin comporte d'autres règles liées aux fonctionnalités offertes et à la sécurité de fonctionnement. Leur description dépasserait l'objectif de cet ouvrage.

g) Les chiffres de BitCoin

Le protocole BitCoin en quelques chiffres-clés³² :

- taille maximum de la masse monétaire : 21 millions de bitcoins
- date estimée de la création du dernier bitcoin : 2 140
- taille d'un bloc : 1 Mo maximum; soit à peu près 4 000 transactions; proposition de passer à 2 MB dès 2018
- taille de la chaîne de blocs : 515 100 blocs, 162 Go, hors index (mars 2018)
- taille des clés de chiffrement : 32 octets pour la clé privée, 65 octets pour la clé publique
- taille de l'adresse bitcoin : 34 caractères ASCII
- temps de minage : conventionnellement 10 minutes par bloc en moyenne
- nombre maximum de transactions validées par seconde : 7, soit 605 000 par jour (147 000 réalisées le 25 mars 2018)
- valeur de *k* (difficulté de minage) : révisée à la hausse tous les 2 016 blocs ajoutés; *k* = 72 en novembre 2017
- évolution du *basecoin* (rémunération par bloc gagnant) : divisé par 2 tous les 210 000 blocs (à peu près tous les 4 ans; 12,5 bitcoins de 2016 à 2020)
- Puissance du réseau (mars 2018) : 26 000 000 *terahashes* (soit $2,6 \cdot 10^{19}$) par seconde, soit 10 fois plus qu'en 2017 à la même date; pour une puissance électrique estimée à plus de 665 MW,³³ soit une consommation annuelle de 58,3 TWh.

10.9.3 Le protocole Ethereum

Lancée en 2015, Ethereum est une plate-forme basée sur une technologie de chaînes de blocs permettant de développer des applications décentralisées. L'une de ses applications est le système de crypto-monnaie *ether*, deuxième derrière le *bitcoin* en termes de popularité.

31. En 2016, les deux plus gros mineurs, AntPool et F2Pool, ont à eux deux réalisés plus de 50% des calculs de *hashes* du réseau.

32. Sources diverses, dont blockchain.info et Hileman, G., Rauchs, M., *Global Cryptocurrency Benchmarking Study* [https://www.jbs.cam.ac.uk/fileadmin/user_upload/research/centres/alternative-finance/downloads/2017-04-20-global-cryptocurrency-benchmarking-study.pdf].

33. <https://digiconomist.net/bitcoin-energy-consumption>

a) Technique de consensus³⁴

Une des principales critiques adressées au mécanisme de consensus par *preuve de travail*, outre la faible vitesse de traitement des transactions, est l'énorme quantité d'énergie consommée, qui s'ajoute aux investissements de plus en plus coûteux.³⁵ On a donc cherché d'autres mécanismes. Si, initialement, le protocole Ethereum appliquait effectivement cette technique, les dernières versions exploreront d'autres techniques, dont la *preuve d'enjeu* (*proof of stake*).

L'idée de cette famille de techniques est de confier la vérification du prochain bloc à un membre dont l'intérêt dans le bon fonctionnement de la chaîne de blocs est élevé, tout en évitant les comportements frauduleux. Pour l'application de cryptomonnaie, la mesure de cet intérêt (son *enjeu*) est le volume de monnaie que possède le candidat mineur ainsi que le temps durant lequel il a conservé ce volume. La sélection du mineur qui va valider un bloc se fait par tirage aléatoire parmi les candidats déclarés. Un poids est assigné à chaque candidat en fonction de son enjeu, poids qui représente la probabilité qu'il soit sélectionné. Ainsi, toutes choses égales par ailleurs, le candidat qui possède deux fois plus de monnaie qu'un autre aura deux fois plus de chance d'être sélectionné que ce dernier. Pour chaque bloc validé, le mineur reçoit une rémunération.

Selon cette procédure, *les plus riches vont devenir encore plus riches*, certains risquant d'occuper une position dominante réduisant la collégialité caractéristique des chaînes de blocs. Rappelons d'abord que des *petits* mineurs sont, plus rarement il est vrai, également sélectionnables. En outre les protocoles prévoient diverses règles limitant l'influence des *gros* mineurs. Deux exemples :

- Après avoir été sélectionné, un mineur ne pourra l'être à nouveau qu'après un délai déterminé (par exemple quinze jours à un mois).
- Le montant qui a servi à mesurer l'enjeu d'un mineur ne peut plus servir lors de sa prochaine candidature.

Les techniques de *preuve d'enjeu* sont vulnérables à certains comportements frauduleux. L'un d'eux est la tentation, étant donné le faible coût de la validation, d'un mineur de miner à lui seul pour plusieurs branches concurrentes, augmentant ainsi ses chances de contribuer à la branche qui sera gagnante, scénario qui est pratiquement impossible par la *preuve de travail*, étant donné le coût du minage.

D'autres critères de sélection sont actuellement étudiés, tels que le taux d'activité ou la réputation du candidat (vérifiée à partir de sources extérieures, voire certifiée par un notaire). Certains protocoles combinent les techniques de *preuve de travail* et de *preuve d'enjeu*.

b) Contrats intelligents

Un *contrat intelligent* (*smart contract*) est un programme qui exécute une opération telle que le transfert d'une somme d'un compte vers un autre. En quelque sorte, alors que la procédure du traitement d'une transaction en BitCoin est prédéfinie, elle est

34. Consulter par exemple https://fr.wikipedia.org/wiki/Preuve_d'enjeu pour plus de détails.

35. On rapporte qu'un mineur chinois dispose d'un centre de 25 000 machines, gérées par 50 personnes.

programmable en Ethereum. Ceci permet de définir des comportements personnalisés, tels que la vérification de conditions d'exécution de la transaction (par exemple, montant maximum journalier d'un transfert ou d'un retrait, tendance boursière pour un titre particulier, vérification de la notoriété d'un destinataire) ou des opérations nouvelles. Le travail des mineurs est plus complexe que ceux du BitCoin puisqu'il dépend de la nature du contrat.

L'implémentation d'Ethereum comprend une machine virtuelle exécutant un *bytecode*, selon une technique similaire à la machine virtuelle de Java. Plusieurs langages de rédaction de contrats intelligents ont été définis, dont *Solidity* est le plus riche et le plus utilisé. Il s'agit d'un langage orienté-objet dont la syntaxe s'inspire de celle de Javascript. Un contrat est semblable à une classe dont les méthodes correspondent aux opérations disponibles. Une opération d'un contrat se déclenche à la demande, lorsqu'une condition est satisfaite (par exemple, liée à la valeur d'un titre en bourse) ou lorsqu'un événement est détecté (une date déterminée par exemple). Des contrats différents peuvent interagir, soit entre eux, soit avec des programmes externes. Tout comme les transactions, les contrats sont infalsifiables.

En référence à l'architecture des SGBD, un contrat est semblable à une *procédure SQL* invocable ou exécutée à partir de *déclencheurs*, à la différence que cette procédure et ces déclencheurs seraient stockés dans les données elles-mêmes et non dans le schéma. Une chaîne de blocs peut dans une certaine mesure être modélisée par une *base de données active*.

c) Applications

Dégagée de la culture et du vocabulaire des *chaînes de blocs*, la plate-forme Ethereum apparaît comme un environnement de développement et d'exécution d'applications de gestion distribuées, doté d'un langage d'application et d'un SGBD primitif. Les contrats ne sont autres que des programmes d'applications compilés et enregistrés dans la base de données.

Cette interprétation admise, on comprend que la plate-forme permet le développement d'applications de toutes natures, classiques ou innovantes : gestion des clients, système de vote électronique, vente d'énergie électrique entre individus, cadastre immobilier, financement participatif, levée de fonds (ICO), réseau social, organisation décentralisées autonomes (DAO), gestion de contrats d'assurance, gestion des dossiers patients, chaîne d'approvisionnement, traçabilité de la vente de diamants, certification de diplômes, ventes entre particuliers, partage de véhicules, etc. Des applications de gestion complexes telles que les ERP³⁶ sont évoquées.

36. *Enterprise Resource Planning*, logiciel de gestion intégrée générique dont l'adaptation à une organisation particulière (structure, processus, règles de gestion, écrans, documents) s'effectue par la fixation de paramètres. SAP est l'un des ERP les plus connus.

10.10 POUR EN SAVOIR PLUS

Les références sur les SGBD non relationnels sont encore peu nombreuses en librairie. On trouvera dans [Sadalage, 2013] un panorama critique des problèmes et des technologies NoSQL et dans [Bruchez, 2013] un développement plus détaillé.

Les problèmes fondamentaux liés aux données distribuées sont détaillés dans [Silberschatz, 2011] et surtout dans [Gray, 1993], qui reste une référence incontournable. Ces deux ouvrages sont cependant *pré-NoSQL*. Cassandra est décrit très en détails dans [Hewitt, 2016] et MongoDB dans [Banker, 2016]. Les sites des éditeurs mettent en général à disposition gratuite des tutoriels et les manuels des SGBD qu'ils distribuent.

Les matériaux sont en revanche très nombreux mais de qualité très variable sur le web. On citera un intéressant article de Date [Date, 2015] comparant avec verve NoSQL et SQL.

En ce qui concerne les chaînes de blocs, en raison de la nouveauté, de la variété et de l'immatunité de cette technique, les sources d'information sont actuellement (mars 2018) encore rares, souvent superficielles ou à but commercial, voire contradictoires.³⁷ Les quelques tutoriels disponibles (principalement sur YouTube) sont utiles pour comprendre les grands principes mais insuffisants pour répondre à toutes les questions qu'un lecteur motivé serait en droit de se poser. Les choses vont certainement évoluer dans les prochaines années, voire dans les prochains mois. On peut en tout cas suggérer l'ouvrage d'un expert réputé, [Antonopoulos, 2017].

Des matériaux complémentaires sont disponibles à l'annexe A10, notamment des scripts SQLfast de conversion de données entre les différents modèles NoSQL.

37. Si la technologie des chaînes de blocs n'est pas nouvelle, ce n'est que tout récemment que son intérêt et son potentiel d'application a été perçu par le grand public.

Chapitre 11

Construction d'une base de données

On introduit le problème de la construction d'un schéma de base de données décrivant les besoins en information d'une communauté d'utilisateurs. Cette construction comprend deux étapes : l'élaboration du schéma conceptuel qui décrit, sans référence à la technologie informatique, les concepts du domaine d'application et la traduction de ce schéma en un schéma de base de données en SQL. Ce schéma conceptuel s'appuie sur un modèle abstrait : le modèle Entité-association.

11.1 DE L'ARTISANAT À LA MÉTHODE

Une démarche de conception de base de données doit conduire à la définition des données 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, une entreprise de vente en ligne, le département de marketing d'une entreprise, la production d'énergie, voire un réseau social.

La base de données doit contenir les données nécessaires à la représentation du domaine d'application. Ces dernières 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. La structuration de notre petite base de données de la figure 2.8 n'a pas exigé, le lecteur en conviendra, d'effort intellectuel démesuré !



Figure 11.1 - Extrait du schéma d'une base de données de taille moyenne. Les clés étrangères n'ont pas été représentées.

Cependant, lorsque le domaine d'application présente une certaine complexité, il devient impossible de raisonner à son sujet en termes de tables et de colonnes. La figure 11.1 représente une petite partie du schéma de la base de données d'une entreprise de distribution par correspondance, qui comprend 860 tables.

Sans même envisager pour l'instant de telles bases de données (nous y reviendrons au chapitre 15), il n'en reste pas moins qu'un niveau de raisonnement **indépendant des technologies de gestion de données** (SQL en est une) s'avère rapidement indispensable. 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 et leurs informations, sans s'attacher à la manière dont ces informations seront représentées 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**. Ce schéma est un document essentiel qui permet aux informaticiens et aux utilisateurs de communiquer. Il est aussi le cahier des charges précis pour l'implémentation de la base de données.

11.2 LES ACTEURS ET LES MÉTIERS

Plusieurs rôles, parfois joués par la même personne, vont apparaître dans le processus d'élaboration et d'exploitation d'une base de données. Le plus important est l'**utilisateur**. Expert du domaine d'application, celui-ci exprime ses besoins en information mais n'a pas de compétence informatique. L'**analyste** dialogue avec l'utilisateur pour cerner et exprimer précisément ses besoins, qu'il traduit sous la

forme d'un schéma conceptuel. L'analyste est familier du domaine d'application et connaît le langage de l'utilisateur. Il possède aussi de solides compétences en modélisation. L'aspect humain est essentiel dans son métier, qui réclame une grande aptitude au dialogue. L'**implémenteur** est chargé de produire le schéma d'une base de données (sous forme de tables) qui traduit de manière rigoureuse et exacte les composants du schéma conceptuel. Cette base de données doit être efficace, offrir de bonnes performances, être facile à utiliser et favoriser toutes les évolutions futures (extensions, transformations, migration, etc.). Le **développeur** va concevoir et écrire les programmes qui vont utiliser la base de données. Il doit disposer d'une documentation précise et complète de tous les composants de la base de données, notamment sous la forme de son schéma conceptuel et de son schéma de tables. L'**administrateur de la base de données** veille, au jour le jour, au bon fonctionnement de la base de données. Il suit les mises à jour du SGBD, suggère et applique des modalités d'exploitation optimale des données (choix des index par exemple), définit et applique les politiques de sécurité et de protection de la base de données. L'**administrateur des données** gère les données comme une ressource critique de l'entreprise. Il n'est pas (nécessairement) un spécialiste des technologies des bases de données, mais il connaît tous les types de données de l'entreprise, leur localisation, leur forme, leur signification, leur qualité, leurs créateurs et utilisateurs habituels. Il renseigne les utilisateurs sur l'existence et la disponibilité des données dont ils ont besoin et propose, si nécessaire, l'acquisition de nouvelles données. Analystes, implémenteurs, développeurs et administrateurs collaborent pour assurer la maintenance et l'évolution des bases de données et des programmes.

11.3 DEUX PARCOURS

Cet ouvrage aborde la construction d'une base de données selon *deux parcours complets mais complémentaires* (voir Avant-propos). Le premier parcours donne au lecteur une vision globale mais simplifiée du processus de construction d'une base de données, qui va de la compréhension du domaine d'application à la production du code SQL. Il se base sur un sous-ensemble du modèle Entité-association et propose des tâches d'analyse simples et intuitives. Ce parcours est constitué de ce chapitre, suivi des chapitres 12 à 15. Il permet au lecteur de construire de manière *rationnelle* et *systématique* des bases de données de taille et de complexité raisonnables. Le deuxième parcours est destiné au lecteur plus exigeant, désireux d'en apprendre plus sur la modélisation et la conception des grandes bases de données. Ce lecteur trouvera dans les chapitres 16 à 21 une description plus approfondie du modèle Entité-association et des processus conduisant à une base de données complètement codée en SQL. Il est à noter que ce second parcours se base sur les acquis du premier.

11.4 UNE DÉMARCHE SIMPLIFIÉE

Le schéma conceptuel est un modèle mental, abstrait, des concepts du domaine d'application et de ses informations et n'est donc pas directement implantable dans

un ordinateur. Il faut ensuite le *traduire en un schéma de base de données* sous la forme de tables, de colonnes et de contraintes d'intégrité. Nous limiterons le processus de traduction à des règles simples et systématiques. La démarche de conception de bases de données proposée consiste donc en deux phases (figure 11.2) :

- *l'analyse conceptuelle*, au terme de 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.

Nous examinerons successivement les principes du modèle Entité-association (chapitre 12), le processus de construction du schéma conceptuel (chapitre 13) et celui de production des structures de tables SQL (chapitre 14). La première étude de cas du chapitre 23 illustre l'application de ces processus.

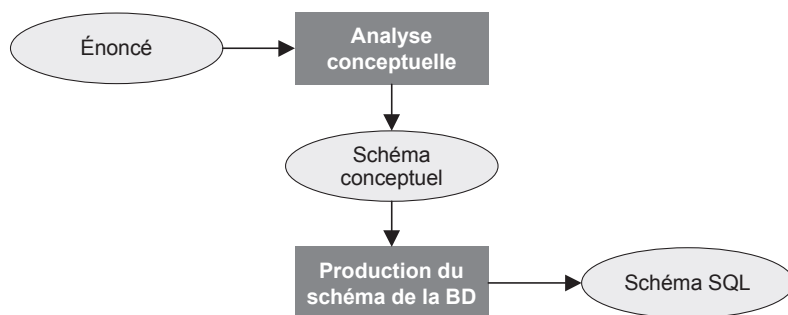


Figure 11.2 - Structure générale de la méthode de construction d'une base de données

11.5 LES OUTILS

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 AGL¹, 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 suivants de cet ouvrage ont été réalisés à l'aide de l'AGL DB-MAIN, spécialisé dans les activités d'ingénierie des applications de bases de données. Cet atelier est disponible gratuitement sur le site de l'ouvrage.

1. En anglais **CASE tools** (*Computer Aided Software Engineering*).

Chapitre 12

Le modèle Entité-association de base

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 composé d'ensembles d'entités, dotées de propriétés et en association les unes avec les autres, et ce, sans référence aux notions techniques de tables, de colonnes et autres index.

Dans ce chapitre, on présente les concepts de base du modèle Entité-association et leurs propriétés. On illustre ces concepts par des exemples de schémas complets. Des extensions à ce modèle de base seront proposées au chapitre 16.

12.1 INTRODUCTION

Le modèle Entité-association est né dans les années soixante-dix de la nécessité de décrire les besoins en information des utilisateurs d'une base de données d'une manière complètement indépendante des technologies de mise en œuvre de celle-ci.

Partant de l'idée qu'une base de données doit être une description fidèle du domaine d'application, on envisage de relever dans celui-ci les concepts principaux et leurs propriétés. L'identification et la description de ces concepts utilisent un **modèle**, c'est-à-dire une *manière de voir les choses*, auquel on associe une notation graphique. Un même modèle peut faire l'objet de plusieurs notations. C'est ainsi qu'on utilise aujourd'hui plus d'une dizaine de représentations graphiques, plus ou moins équivalentes, des objets du modèle Entité-association. Celle que nous utilisons dans cet ouvrage n'est donc pas unique.

La *manière de voir les choses*, ou *vision du monde*, du modèle Entité-association est très simple, voire simpliste. Elle consiste à structurer le monde en classes d'entités, dotées de propriétés et en association les unes avec les autres. Cette simplicité offre deux avantages majeurs : elle met ce modèle à la portée des utilisateurs/modélisateurs peu expérimentés et elle se traduit aisément en un schéma de base de données. Néanmoins, les modélisateurs plus exigeants auront besoin d'un modèle plus riche et de méthodes plus puissantes, qui seront décrits plus loin.

Dans ce chapitre, nous introduirons le lecteur successivement aux notions de type d'entités, d'attribut, de type d'associations et d'identifiant.

12.2 LES 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 et qu'on représentera d'une manière graphique comme indiqué dans la figure 12.1.



Figure 12.1 - 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 dans le temps. Il est cependant pratique, pour comprendre et analyser les caractéristiques d'un schéma, d'imaginer ces populations, classées par types.

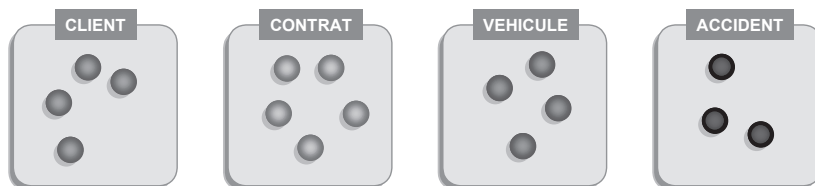


Figure 12.2 - Exemples de populations représentées symboliquement.

La figure 12.2 représente de manière imagée une population de chaque type, où une entité est représentée par une pastille (un auteur plus doué graphiquement aurait dessiné, fût-ce sommairement, quelques clients, des véhicules, des contrats, voire des accidents).

12.3 LES 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 12.3). Selon le même raisonnement, on assignera à ACCIDENT les attributs NumAcc, DateAcc et Montant.

CLIENT	CONTRAT	VEHICULE	ACCIDENT
NumClient	NumCtr	NumVéh	NumAcc
Nom	Type	Marque	DateAcc
Adresse	DateSign	Modèle	Montant
		Année	
		Cylindrée	

Figure 12.3 - Les quatre types d'entités et leurs attributs

On spécifiera le type de chaque attribut : numérique, caractère, date, etc. ainsi que sa longueur (figure 12.4). On indiquera aussi, si nécessaire, son unité de mesure.

VEHICULE
NumVéh: char (16)
Marque: char (30)
Modèle: char (30)
Année: num (4)
Cylindrée: num (6)

Figure 12.4 - À 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, de sorte que cet attribut n'a pas 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 12.5). 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 12.5 - Attributs obligatoires et facultatifs

Pour compléter la représentation graphique des entités, on associera à chacune une fiche reprenant les valeurs de ses attributs (figure 12.6).

NumClient = C400 Nom = FERARD Adresse = 65, r. du Tertre	NumClient = B332 Nom = MONTI Adresse = 112, r. Neuve	NumClient = F010 Nom = TOUSSAINT Adresse = 5, r. Godefroid

Figure 12.6 - Trois entités du type CLIENT et leurs valeurs d'attributs

12.4 LES 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 12.7. La question de la dénomination des types d'associations est complexe, de sorte que nous ne l'étudierons qu'au chapitre 16.

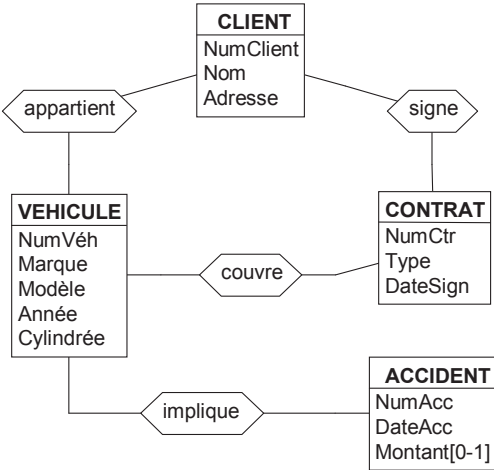


Figure 12.7 - Types d'entités, attributs et types d'associations

12.4.1 Rôles d'un type d'associations

Un type d'entités qui intervient dans un type d'associations y joue un **rôle**. On utilisera ce terme pour désigner une des extrémités d'un type d'associations. On peut assigner un nom à un rôle. En l'absence de nom explicite, c'est celui du type d'entités qui en tiendra lieu. Les deux rôles du schéma de la figure 12.8 portent les noms de signataire et CONTRAT. Ce dernier est un nom implicite. Les noms des rôles d'un type d'associations sont distincts, qu'ils soient explicites ou implicites. Deux rôles peuvent porter le même nom s'ils appartiennent à des types d'associations différents. Pour les distinguer, on utilisera leur nom absolu, formé de leur nom précédé de celui du type d'associations : signe.signataire et signe.CONTRAT par exemple.



Figure 12.8 - Noms de rôle explicite et implicite

12.4.2 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. On mesurera cette propriété en indiquant, pour chaque type d'entités, les nombres maximum et minimum d'associations auxquelles toute entité participe, nombres que nous représenterons d'abord par la *classe fonctionnelle* et le *caractère obligatoire/facultatif* des types d'associations, propriétés qui seront ensuite synthétisées sous la forme d'une propriété de *cardinalité*.

12.4.3 Classe fonctionnelle d'un type d'associations

Cette propriété décrit le nombre maximum d'entités B pour chaque entité A, et inversement. On définit ainsi trois classes fonctionnelles¹ de types d'associations : *un-à-plusieurs*, *un-à-un* et *plusieurs-à-plusieurs*.² Analysons-les de manière plus précise.

a) Type d'associations *un-à-plusieurs* (1:N)

Le schéma de la figure 12.9 représente un ensemble de quatre départements d'une entreprise, symboliquement représentés par les pastilles étiquetées de d1 à d4, ainsi qu'un ensemble de six employés, étiquetés de e1 à 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 que le département d4 occupe l'employé e3. Il s'agit d'un type d'associations **un-à-plusieurs** ou **1:N**.

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 indique qu'un département peut occuper plusieurs (N) employés, mais qu'un employé n'est occupé que par un seul (1) département.

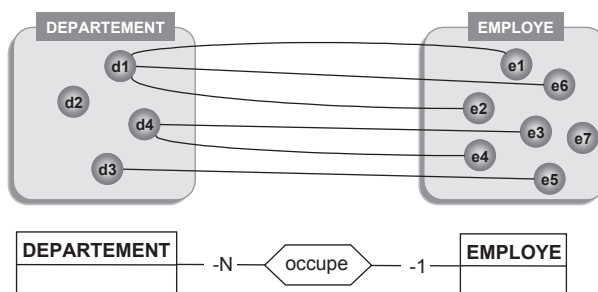


Figure 12.9 - Type d'associations *un-à-plusieurs*. Un employé est occupé par un département et un département peut occuper **plusieurs** employés

1. Le terme *fonctionnel* fait référence aux fonctions dont le type d'associations est le siège.

2. Nous ne définirons pas la classe fonctionnelle *plusieurs-à-un* ou **N:1**, qui est simplement l'inverse de la classe *un-à-plusieurs*. Elle correspond à une fonction au sens mathématique.

b) Type d'associations un-à-un (1:1)

Le schéma de la figure 12.10 représente les mêmes ensembles de départements et d'employés entre lesquels 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** ou **1:1**. À 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. On indiquera le symbole 1 sur chacune des branches du type d'associations.

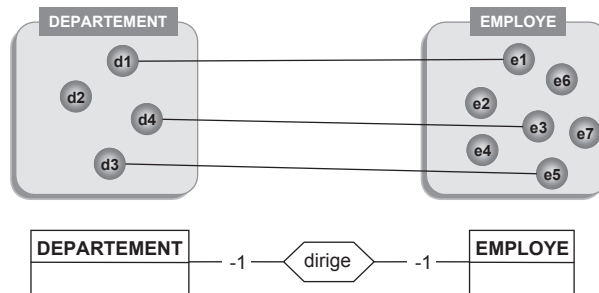


Figure 12.10 - Type d'associations *un-à-un*. Un employé peut diriger au plus un département et un département ne peut être dirigé que par un seul employé

c) Type d'associations plusieurs-à-plusieurs (N:N)

Le schéma de figure 12.11 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** ou **N:N**. À 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*, sur chaque branche du type d'associations.

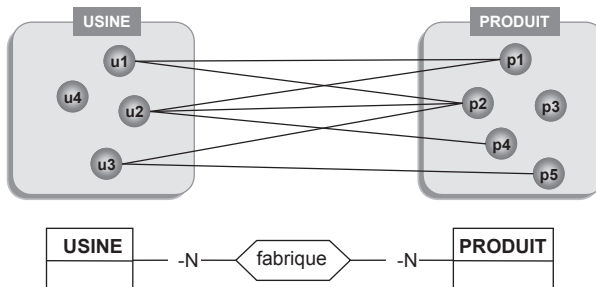


Figure 12.11 - Type d'associations *plusieurs-à-plusieurs*. Une usine fabrique **plusieurs** produits et un produit est fabriqué par **plusieurs** usines

La figure 12.12 complète le schéma de la figure 12.7 en y précisant la classe fonctionnelle de chaque type d'associations. Le lecteur est invité à interpréter précisément ces indications en termes du domaine d'application.

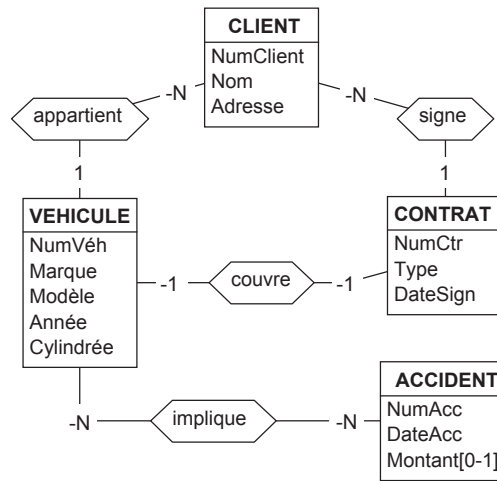


Figure 12.12 - Caractérisation des types d'associations (cardinalités partielles)

Important

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*. Que conclurait-on si on limitait l'analyse à formulation : à *une* usine correspondent *plusieurs* produits ?

12.4.4 Type d'associations (ou rôle) obligatoire ou facultatif

On peut imposer qu'un type d'associations (ou plutôt un rôle) soit **obligatoire** pour un type d'entités qui y participe. C'est ainsi que *occupe* sera déclaré obligatoire pour EMPLOYE (figure 12.13), imposant que toute entité EMPLOYE participe à une association *occupe*, ou encore qu'elle soit associée à une entité DEPARTEMENT. 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.³

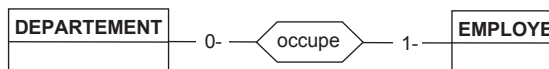


Figure 12.13 - Le type d'associations occupe est obligatoire (1) pour EMPLOYE et facultatif (0) pour DEPARTEMENT

3. Une erreur courante consiste à interpréter le caractère facultatif d'un rôle comme "il se peut qu'on ne connaisse pas l'une des deux entités d'une association". C'est évidemment tout à fait faux : **une association est toujours formée de deux entités**, quelles que soient les contraintes de cardinalité de ses rôles.

12.4.5 Cardinalités d'un type d'associations

En combinant les symboles des classes fonctionnelles et du caractère obligatoire des rôles on obtient des couples $[min-max]$ associés aux rôles, appelés (contrainte de) **cardinalité**. Dans la figure 12.14, qui reprend les cardinalités complètes de notre exemple, on dira que *appartient* (figure 12.14) est de cardinalités $[0-N, 1-1]$ de CLIENT vers VEHICULE. On n'admettra que les valeurs 0-1, 1-1 et 0-N, qui seront 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 examinerons au chapitre 16 d'autres valeurs plus précises.

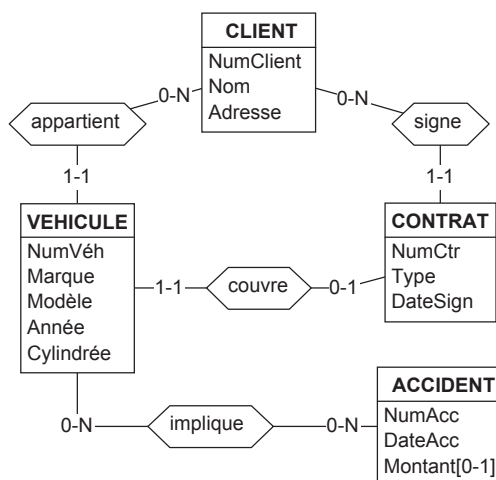


Figure 12.14 - Caractérisation des types d'associations : cardinalités complètes

12.4.6 Type d'associations cyclique

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 12.15 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. Si deux personnes ont une telle relation, on représentera celle-ci par une association *super-vise* 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 d'autres.

Ni récursif, ni réflexif, ni unaire !

On évitera trois qualificatifs fréquemment utilisés mais impropres pour désigner un type d'associations cyclique :

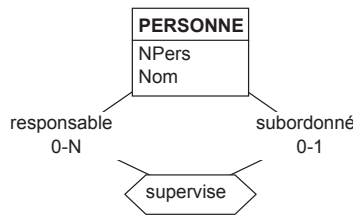


Figure 12.15 - Exemple de type d'associations cyclique

- type d'associations *récuratif* : le terme récuratif qualifie un concept dont la définition mentionne le concept lui-même; certaines définitions XML sont récuratives, mais tel n'est pas le cas des types d'associations cycliques ; la confusion vient du fait que ce type d'associations fait l'objet d'algorithmes récuratifs ;
- type d'associations *réflexif* : une relation $R(A,A)$ est réflexive si, pour tout élément a de A , elle contient le couple (a,a) ; sauf exception rarissime, ce n'est pas le cas des types d'associations cycliques ;
- type d'associations *unaire* : une relation *n-aire* (unaire, binaire, ternaire, quaternaire, etc.) est définie sur n ensembles, non nécessairement distincts, et est constituée d'un ensemble d'agréats de n éléments. Comme nous le verrons à la section 16.17, un type d'associations *n-aire* correspond à une relation *n-aire*. supervise est un type d'associations binaire puisqu'il est constitué de deux rôles.

12.4.7 Représentation graphique des populations

La figure 12.16 représente de manière imagée un exemple de populations interconnectées entre lesquelles les arcs représentent les associations conformément au schéma de la figure 12.14. Les valeurs d'attributs ont été ignorées.

12.5 UNE QUESTION DE VOCABULAIRE

Certains appelleront *entité CLIENT* ce que nous appelons *type d'entités CLIENT*. Ce que nous avons appelé *entité CLIENT* se dénommera alors *instance* ou *occurrence* de *l'entité CLIENT*. Ces deux systèmes d'appellation sont légitimes pour autant qu'on s'en tienne exclusivement à l'un d'eux. Il est important d'être très précis dans l'utilisation des termes relatifs au modèle et aux schémas Entité-association, et ne pas confondre les *types* et leurs *instances*. Dans cet ouvrage, on parlera du *type d'entités CLIENT* et de *l'entité C400*, qui est une de ses instances. L'expression *l'entité CLIENT* y est donc incorrecte. Le tableau suivant résume les expressions admises.

Type	Instance
type d'entité	entité
type d'associations	association
attribut	valeur d'attribut

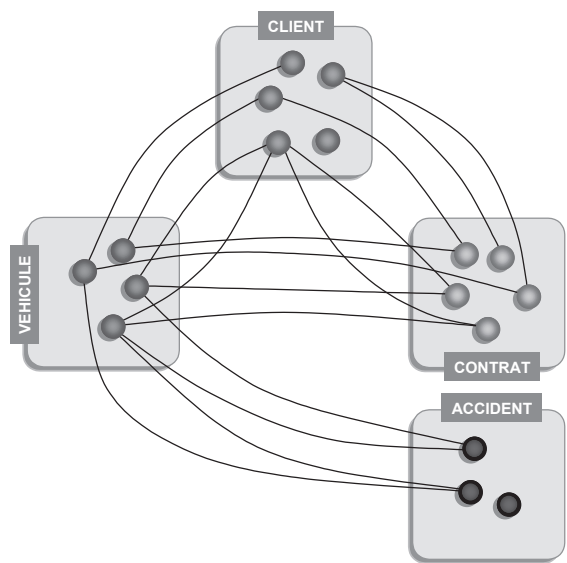


Figure 12.16 - Exemple de populations et des liens qui connectent les entités

12.6 LES IDENTIFIANTS

En général⁴, un type d’entités est doté d’un attribut qui identifie (c’est-à-dire qui distingue les unes des autres) les entités de ce type. Le type d’entités CLIENT (figure 12.17) 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. Un identifiant peut également être formé de plusieurs attributs (VOL). En outre, un type d’entités peut posséder plusieurs identifiants (PROJET). L’un d’eux peut être déclaré **primaire** tandis que tous les autres sont **secondaires**.

CLIENT	VOL	PROJET
<u>NumClient</u>	<u>Ligne</u>	<u>CodeProjet</u>
Nom	<u>Date</u>	Titre
Adresse	NumAppareil	Budget
id: NumClient	id: Ligne	id: CodeProjet
	Date	id': Titre

Figure 12.17 - 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. En outre, les attributs d’un identifiant primaire sont soulignés, du moins quand ce dernier n’est composé que d’attributs.

4. Afin de simplifier l’exposé, on admettra qu’un type d’entités possède toujours un identifiant.

12.6.1 Identifiants hybrides

Le cas du type d'entités CONTRAT (figure 12.18) est un peu plus complexe. On admet en effet que chaque contrat reçoit un numéro NumCtr qui permet de le distinguer parmi ceux que le client a signés. Les contrats d'un client déterminé portent les numéros 1, 2, 3, ... 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 d'un type d'entités voisin et d'un attribut local. On indiquera un tel identifiant par la clause :

id: signe.CLIENT
NumCtr

La figure 12.18 spécifie les identifiants des types d'entités du schéma 12.14.

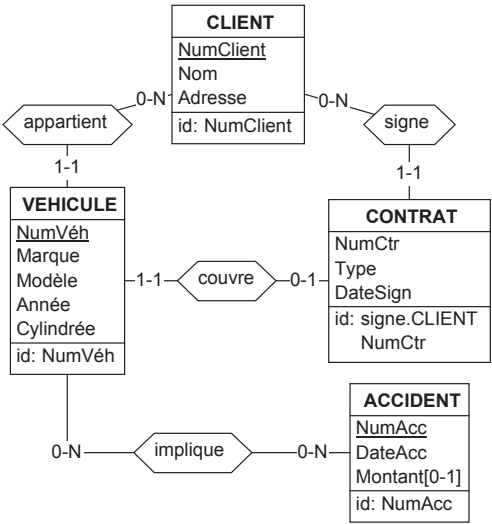


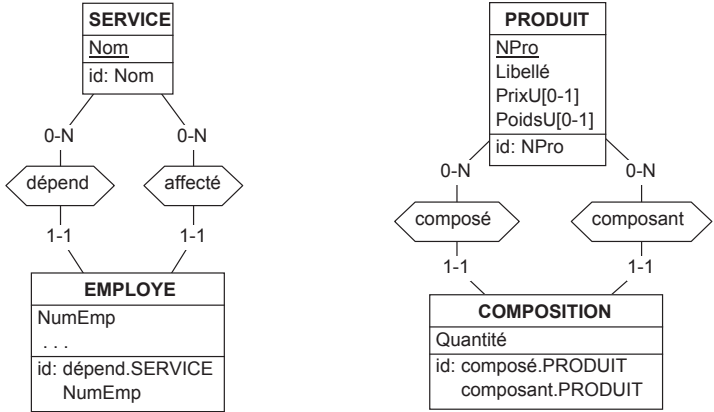
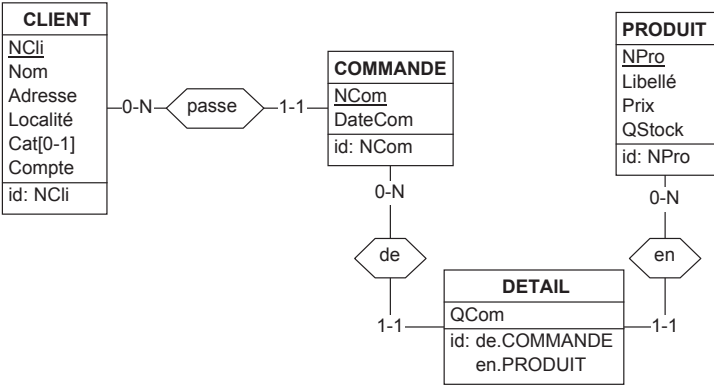
Figure 12.18 - Le schéma final

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 12.19 (on aura reconnu le schéma conceptuel d'une base de données relationnelle familière), 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. La figure 12.20 propose d'autres exemples d'identifiants hybrides. On comparera le schéma de droite avec le schéma de la base de données de la figure 8.5.

Remarque

Le lecteur pourrait se demander pourquoi on ne spécifie pas tout simplement l'identifiant de CONTRAT sous la forme des deux attributs (NumClient, NumCtr).



Ce principe serait 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 12.25 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, comme dans la figure 12.20 par exemple). Enfin, mais ceci concernera le modèle étendu du chapitre 16, un type d'entités n'a pas nécessairement d'identifiant.

12.6.2 Composition des identifiants

En résumé, un identifiant d'un type d'entités E peut être composé comme suit :

1. un ou plusieurs attribut de E,
2. un ou plusieurs attributs de E + un ou plusieurs types d'entités E_j associés à E *via* le ou les types d'associations R_j,
3. plusieurs types d'entités E_j associés *via* R_j à E.

Plutôt que de parler du *type d'entités E_j associé à E via le type d'associations R_j*, on parlera du *rôle opposé R_j.E_j* (comme dépend.SERVICE dans le schéma 12.20).

Dans les compositions 2 et 3, la cardinalité de E dans chacun des R_j est [0-1] ou [1-1], tandis que la cardinalité des E_j 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.

Important : tous les composants d'un identifiant primaire sont obligatoires.

Remarque : *type d'entités faible et identifiant relatif*

Un type d'entités dont l'identifiant primaire comprend un rôle est parfois qualifié de *faible*. Cet identifiant est aussi souvent qualifié de *local* ou *relatif*. Ces concepts sont détaillés à l'annexe A12.

12.6.3 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 que des identifiants minimaux.

Cette règle nous permet de justifier certaines des formes d'identifiants définies précédemment. Considérons par exemple le schéma du haut de la figure 12.21.

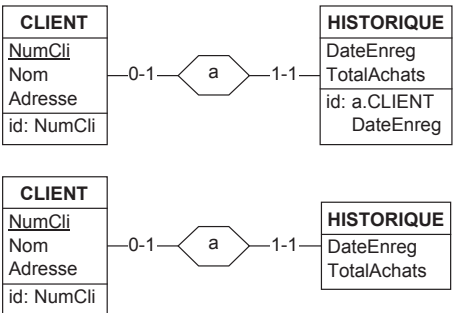


Figure 12.21 - En haut : l'identifiant d'HISTORIQUE est non minimal et doit être réduit. Ce qui reste (a.CLIENT) étant un identifiant implicite, doit aussi être supprimé (en bas).

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 12.21, bas).

12.6.4 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. Considérer qu'une commande spécifie un ensemble de produits pourrait se traduire dans le schéma de la figure 12.22.

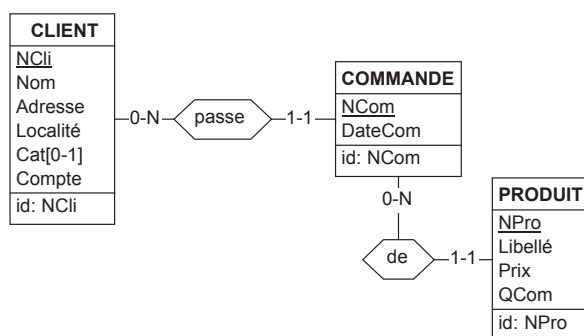


Figure 12.22 - Une mauvaise analyse conduit à confondre détail et produit. Cette erreur se traduit par un identifiant erroné

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 ici les produits, mais un autre concept qui reste à découvrir, c'est-à-dire des détails.

Il est aussi important de noter qu'il existe plusieurs natures d'identifiants. Certains identifiants n'ont pas de signification particulière. Tel est le cas de *NCLI*, *NCom* et *NPro* dans le schéma 12.19. On dira de ces identifiants qu'ils servent de **désignateur**. Ils sont en général constitués d'un *attribut technique* (numéro, code, etc.). En revanche, 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 plus d'une fois un même produit*.

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. Pour certains concepteurs professionnels, l'identifiant primaire est si possible formé d'un attribut technique non significatif, et donc stable. Cette règle favorise les identifiants primaires de désignation.

12.7 LES CONTRAINTES D'INTÉGRITÉ

Les contraintes d'intégrité formalisent certaines propriétés importantes du domaine d'application que les données devront respecter. On distinguera les contraintes de base, intrinsèques au modèle Entité-association, et les contraintes additionnelles.

12.7.1 Contraintes de base

Il s'agit des trois contraintes explicitement définies dans le modèle, auxquelles on a attribué un nom et une représentation graphique :

- 1. les identifiants (primaires et secondaires) ;
- 2. les attributs obligatoires ;
- 3. les contraintes de cardinalité des rôles.

12.7.2 Contraintes additionnelles

On désigne sous ce terme toutes les propriétés qu'on voudrait voir définies dans le schéma mais que le modèle ne permet pas d'exprimer de manière explicite. On indiquera ces contraintes sous la forme d'annotations associées aux objets du schéma, comme illustré à la figure 12.23.

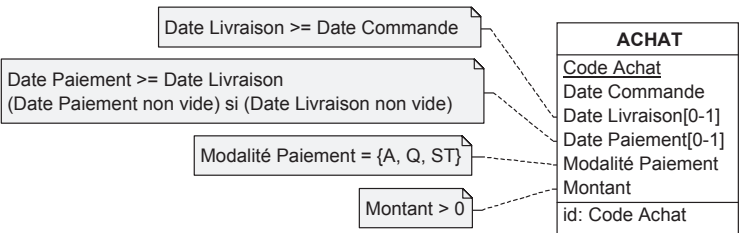


Figure 12.23 - Description de contraintes additionnelles au moyen d'annotations

12.8 CONTENU INFORMATIONNEL D'UN SCHÉMA

La représentation graphique de populations représentatives des types d'entités d'un schéma conceptuel, telle que proposée aux figures 12.2, 12.6 et 12.16, nous permet de vérifier qu'un schéma permet de répondre aux questions que se posent les utilisateurs.

Certaines questions sont à ce point élémentaires 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 12.24) :

- 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 ;

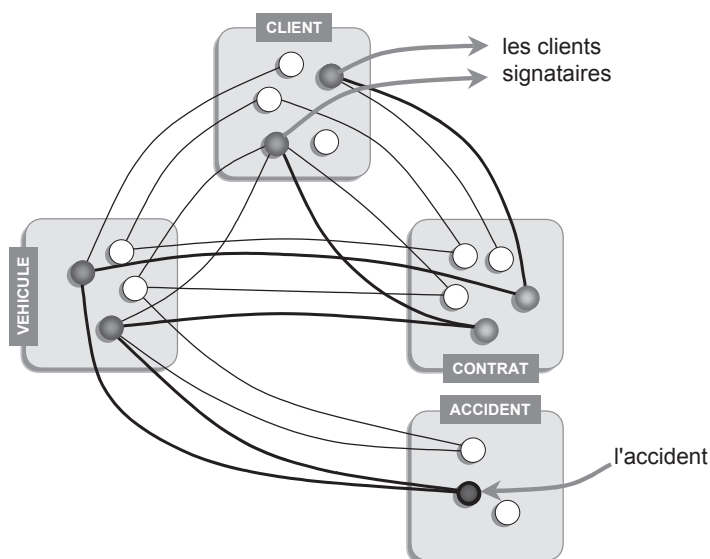


Figure 12.24 - Interrogation d'un ensemble de populations : *quels sont les clients qui ont signé les contrats couvrant les véhicules impliqués dans un accident déterminé ?*

- à partir de ces contrats, et en suivant les associations signe, on obtient finalement les deux clients signataires.

Cette manière de conduire une recherche se généralise aisément à des conditions plus élaborées 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).

À 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*, ou encore, en représentant cette fois les valeurs d'attributs, *les accidents qui impliquent des véhicules de la même marque*.

12.9 EXEMPLES

Nous proposerons trois exemples supplémentaires illustrant la notion de schéma conceptuel exprimé dans le modèle Entité-association⁵. 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 équivalents.

12.9.1 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 12.25 représente un exemple de schéma conceptuel décrivant cette situation.

Cette structure permet de répondre à des questions simples :

- 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 12.8 :

- 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 déterminé⁶.

5. Le lecteur intéressé par des schémas conceptuels d'entreprise consultera par exemple [Silverston, 1997] et [Hay, 2006].

6. L'erreur d'interprétation qui consiste à considérer que la composition des deux types d'associations *traite* et *occupe* répond à cette question porte le nom de *connection trap*.

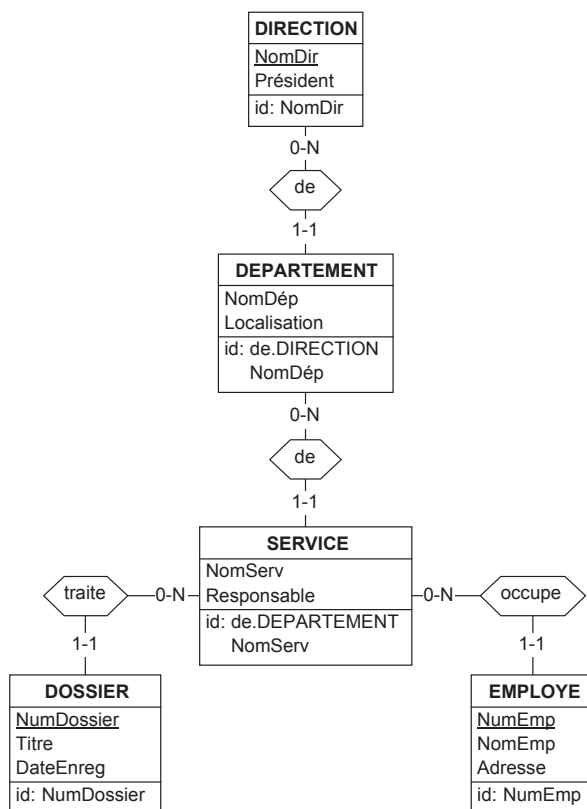


Figure 12.25 - Structure administrative élémentaire

12.9.2 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 12.26.

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 ?

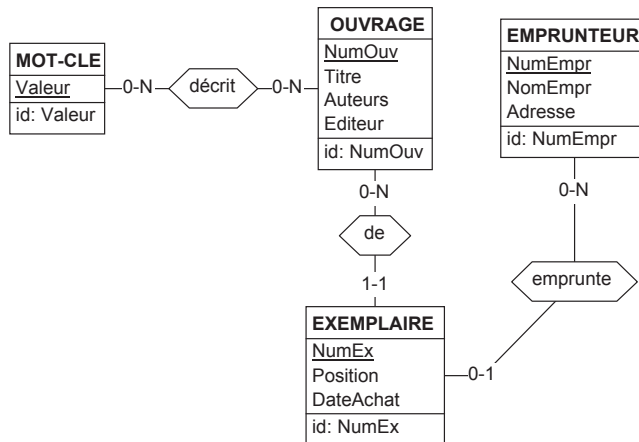


Figure 12.26 - Gestion de bibliothèque élémentaire

- 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 ?

12.9.3 Voyages en train

Ce dernier 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.

On peut proposer un schéma conceptuel des informations relatives à cette situation tel que celui de la figure 12.27.

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 effectué tel train ?
- Y a-t-il des agents qui sont à la fois conducteurs et directeurs de station ?

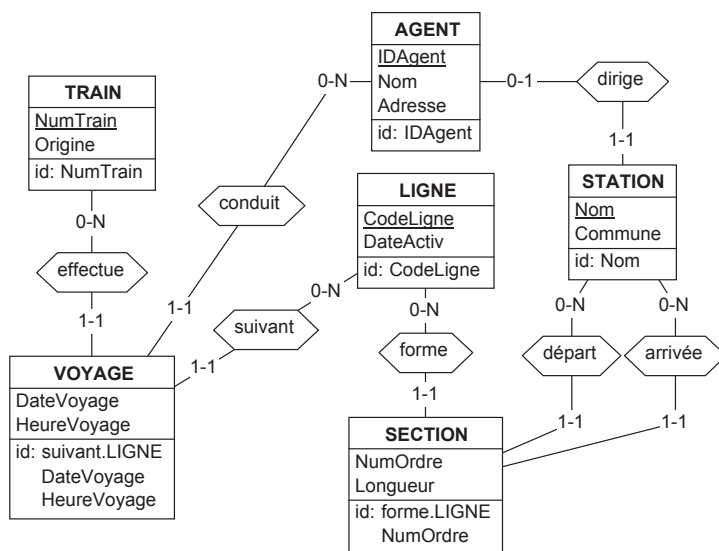


Figure 12.27 - Représentation de voyages en train

12.10 QUE RETENIR ?

Le modèle Entité-association permet de structurer les concepts d'un domaine d'application en types d'entités, attributs et types d'associations. Un **type d'entités** représente une classe d'objets (ou **entités**) remarquable du domaine d'application et chacun de ses **attributs** représente une propriété commune aux entités de ce type. Les entités peuvent être mises en relation deux à deux par des **associations** binaires, classées en **types d'associations**.

Un attribut tire ses valeurs d'un **type de valeurs** de base (entier, caractères, booléen, etc.) Un attribut est **obligatoire** si chaque entité doit posséder une valeur pour cet attribut. Sinon, il est déclaré **facultatif**.

Un type d'associations comprend deux **rôles**, joués chacun par un type d'entités. Un type d'associations dont les rôles sont joués par le même type d'entités est dit cyclique. Un rôle est caractérisé par une **contrainte de cardinalité** formée de deux nombres précisant le nombre minimum et le nombre maximum d'associations dans lesquelles une entité apparaît. Le premier nombre est 0 (rôle facultatif) ou 1 (rôle obligatoire). Le second nombre est 1 ou N (pour *infini*). Ces contraintes définissent trois **classes fonctionnelles** : *un-à-plusieurs*, *un-à-un* et *plusieurs-à-plusieurs*.

Un type d'entités possède un ou plusieurs identifiants. Un **identifiant** du type d'entités E est un groupe de composants dont les valeurs sont uniques pour l'ensemble des entités du type. Ces composants sont des attributs de E ou des rôles opposés à celui de E dans des types d'associations *un-à-plusieurs*. Un **identifiant hybride** possède au moins un rôle. Plus précisément, un identifiant est constitué (1) d'un ou plusieurs attributs, (2) d'au moins deux rôles ou (3) d'un ou plusieurs attributs et d'un ou plusieurs rôles.

12.11 POUR EN SAVOIR PLUS

Les concepts présentés dans ce chapitre forment un modèle simple mais suffisant pour traiter des situations de complexité moyenne. La résolution de problèmes plus importants exigera un modèle plus riche qui sera étudié au chapitre 16.

12.12 DESSINER UN SCHÉMA CONCEPTUEL À L'AIDE D'UN OUTIL

Il n'est pas indispensable d'utiliser un outil spécialisé pour dessiner un schéma conceptuel de petite taille. Cependant, l'usage d'un atelier de génie logiciel est fortement conseillé dès qu'on aborde des schémas plus complexes. D'autre part, la production du schéma de tables et du code SQL étudiée au chapitre suivant ne peut s'envisager de façon raisonnable qu'au moyen d'un tel outil. On trouvera sur le web de nombreux outils, certains gratuits, permettant de dessiner des schémas conceptuels selon une des variantes du modèle Entité-association ou sous forme d'un diagramme de classes UML. Dans cet ouvrage, nous utiliserons l'outil DB-MAIN disponible sur le site de l'ouvrage. Cet outil inclut un petit tutoriel relatif aux concepts décrits dans ce chapitre et à la création de schémas simples.

La figure 12.28 montre la phase terminale de la création du schéma conceptuel illustrant ce chapitre. L'utilisateur a sélectionné le rôle signe.CLIENT et l'attribut NumCtr et s'apprête à cliquer sur le bouton  dans la barre d'outils à droite pour créer l'identifiant de CONTRAT.

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, suggérant la structure hiérarchique induite par ce type d'associations (cf. schéma 12.25).
- 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 (cf. schéma 12.26).
- Si deux types d'entités sont reliés par un type d'associations *un-à-un*, on les placera également au même niveau. Mais si l'un joue un rôle de cardinalité 1-1, on le disposera plus bas que l'autre, indiquant ainsi la relation de dépendance.
- On adoptera des conventions cohérentes dans le choix des noms des types d'entités, des types d'associations et des attributs. Exemples :
 - *types d'entités* : substantifs en capitales ;
 - *attributs* : substantifs en minuscules, première lettre en capitale ;
 - *types d'associations* : verbes, prépositions ou conjonctions en minuscules.

7. Un schéma réaliste contiendra plusieurs centaines à plusieurs milliers de types d'entités.

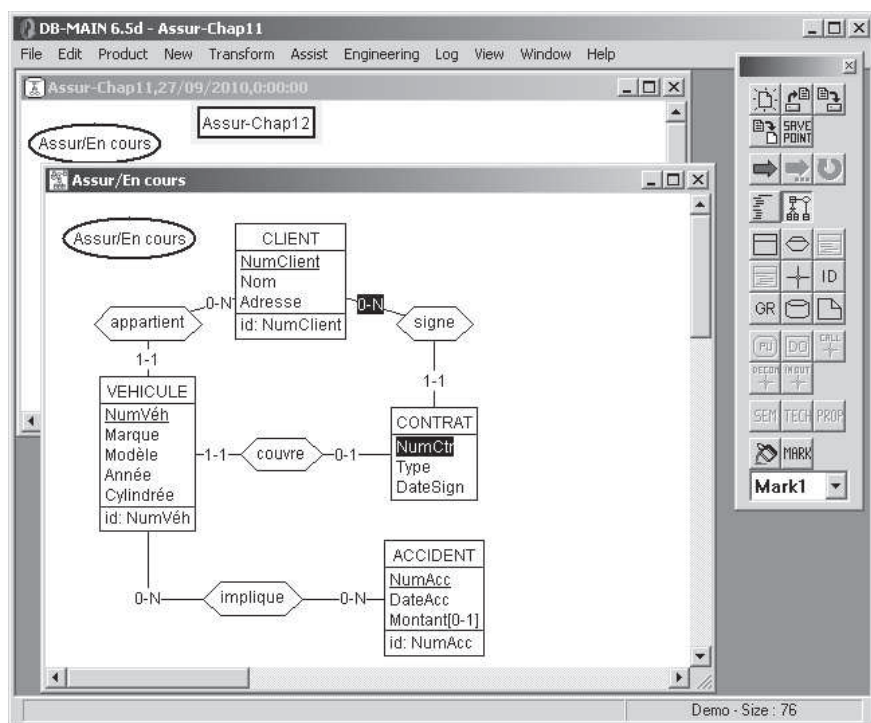


Figure 12.28 - Dessin d'un schéma conceptuel à l'aide de l'outil DB-MAIN

- Dans la liste des attributs d'un type d'entités, on placera les composants de l'identifiant primaire en premier.

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.

12.13 EXERCICES

- 12.1 Modifier le schéma de la figure 12.18 pour qu'il représente aussi les faits suivants (a) un véhicule peut être couvert par un nombre quelconque de contrats, (b) on désire connaître, chaque fois qu'un véhicule est impliqué dans un accident, le pourcentage de responsabilité qui lui a été attribué.
- 12.2 Dans l'interprétation du schéma 12.25, on précise qu'un service est réputé traiter un dossier dès qu'un de ses employés est en charge de ce dossier. Modifier le schéma pour tenir compte de cette précision.
- 12.3 Modifier le schéma de la figure 12.26 de manière à y représenter
 - la réservation d'un ouvrage (ou d'un exemplaire ?) par un emprunteur;
 - l'historique des emprunts d'ouvrages (ou d'exemplaires ?).

- 12.4 Le schéma de la figure 12.27 représente des sections de ligne de telle manière que deux lignes empruntant le même tronçon de voie définissent deux sections différentes. Définir le concept de *tronçon de voie* et l'intégrer au schéma mentionné ci-dessus.
- 12.5 On considère les deux types d'associations *r* entre A et B et *s* entre B et C. On considère aussi le type d'associations *rs* défini comme la composition de *r* et *s* (figure 12.29). Déterminer la classe fonctionnelle de *rs* à partir de celles de *r* et de *s*. Cette question est très importante pour comprendre l'information contenue dans un schéma. En considérant le schéma de la figure 12.18, elle pourrait par exemple se concrétiser comme suit :
- combien de véhicules sont couverts par les contrats signés par un client ?
 - combien de contrats couvrent les véhicules appartenant à un client ?
 - combien de clients sont propriétaires des véhicules couverts par un contrat ?
 - combien de propriétaires sont impliqués dans un accident ?
 - combien de clients ont signé les contrats couvrant les véhicules d'un propriétaire (double composition) ?

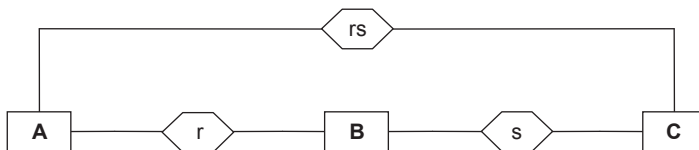


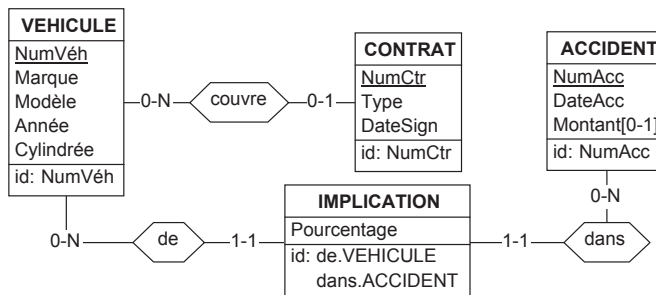
Figure 12.29 - Etude de la composition de deux types d'associations

La solution de cet exercice est proposée dans l'annexe A12 de ce chapitre.

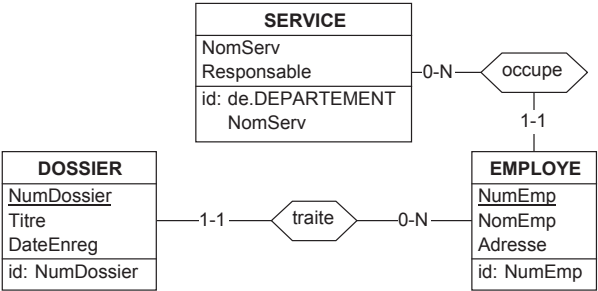
Des exercices et solutions additionnels sont disponibles à l'annexe A12.

Exercices corrigés

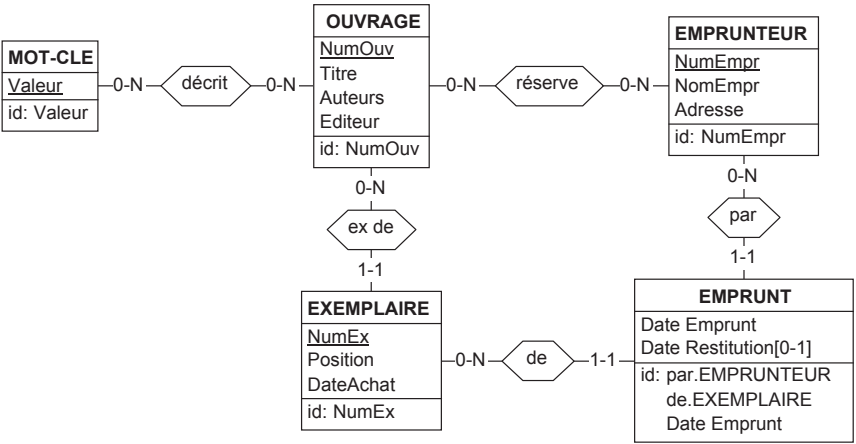
• Exercice 12.1



• Exercice 12.2

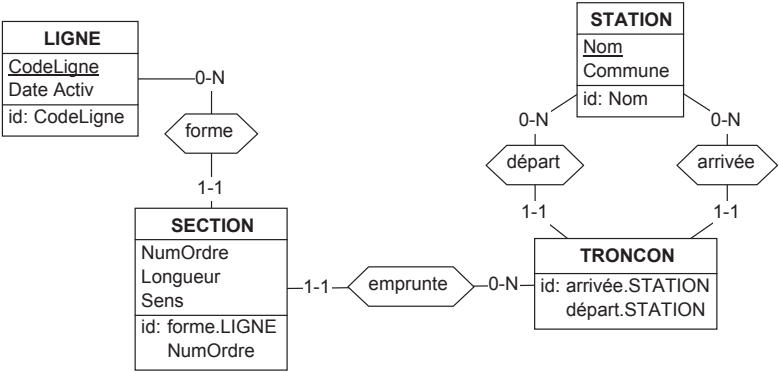


• Exercice 12.3



• Exercice 12.4.

Un **tronçon** est une liaison directe entre deux stations ; il n'existe pas de station au milieu d'un tronçon. Un tronçon possède une station de **départ** et une station d'**arrivée**. Il n'existe au plus un tronçon entre deux stations, bien qu'un tronçon puisse comporter plusieurs portions de voies physiques (= rails). Une section d'une ligne emprunte un tronçon dans le **sens direct** (départ → arrivée) ou dans le **sens inverse** (arrivée → départ).



Chapitre 13

Élaboration d'un schéma conceptuel

Ce chapitre étudie 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 normalisation, la validation et la finalisation du schéma.

13.1 INTRODUCTION

L'élaboration du schéma conceptuel d'une base de données est à la fois cruciale, car elle conditionne la qualité de celle-ci, et complexe, en ce qu'elle 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 représentent, contrôlent ou pilotent ce domaine d'application.

Le formalisme de représentation que nous utiliserons est le modèle Entité-association de base² présenté au chapitre 12. Conformément à l'approche simplifiée que nous adoptons dans cette première approche de la conception d'une base de données, nous nous limiterons à une démarche simple et intuitive. Le lecteur dési-

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 pragmatique de *domaine d'application*.

reux d'approfondir cette démarche trouvera au chapitre 18 des techniques et des outils prolongeant ceux qui sont présentés dans le présent chapitre.

Les sources d'information utilisées lors de l'élaboration d'un schéma conceptuel peuvent être extrêmement diverses. En accord avec l'objectif de parcours rapide, nous les 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. Ces sources additionnelles seront examinées au chapitre 18.

Le processus peut s'exprimer comme suit : *à partir d'un énoncé, construire un schéma Entité-association représentant les concepts et les faits exprimés, explicitement ou implicitement, dans cet énoncé, 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 erronées.

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.

À la fin du processus, le schéma courant est le schéma final.

13.2 UN PREMIER EXEMPLE

Nous illustrerons le processus de construction d'un schéma conceptuel par un premier exemple élémentaire. Supposons qu'on dispose du texte ci-après, décrivant la manière dont un employé d'une bibliothèque d'entreprise voit son environnement de travail :

2. 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. Nous analyserons ces concepts un peu subtils et leurs relations au chapitre 16.

un employé peut emprunter des livres ou en réserver ; l'employé a un nom et une adresse ; il possède un numéro unique et travaille dans un service de l'entreprise, identifié par son nom et dont on connaît la localisation ; un ouvrage porte un numéro ISBN unique et possède un titre, un prix et une date d'achat ;

La figure 13.1 montre un schéma conceptuel qui pourrait être tiré de cet énoncé.

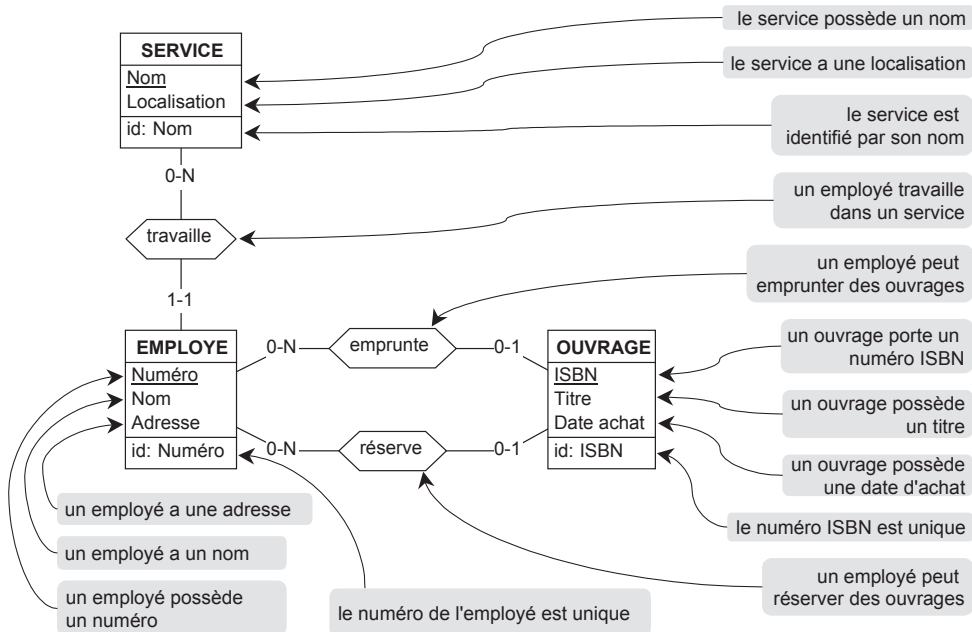


Figure 13.1 - Du texte au schéma conceptuel

Un premier coup d'œil nous montre que chaque construction du schéma est la traduction d'un fragment du texte. Cependant, une observation plus fine nous apprend aussi que :

- certains fragments ont été **reformulés** (*le numéro de l'employé est unique*) ;
- certains fragments n'ont **pas** été **utilisés** (*un prix*) ;
- on perçoit bien comment les attributs et les types d'associations ont été détectés, mais la chose est moins évidente pour les **types d'entités** ;
- certains fragments fournissent la même information et sont donc **redondants** (*livres* et *ouvrages* apparaissent comme des synonymes) ;
- il semble que le concept d'*ouvrage* ne soit pas bien compris et que son expression soit erronée (on retrouve le problème de la section 2.11 : confusion entre *ouvrage* et *exemplaire*) ; le texte contient donc des **formulations incorrecte** ;
- enfin, un schéma ainsi annoté est **plus lisible** et compréhensible que si les fragments étaient retirés ; ceux-ci contribuent donc à la **documentation** du schéma.

C'est donc par une *analyse critique* des informations que nous pouvons tirer de l'énoncé que nous identifierons les éléments pertinents du schéma. C'est à cette question, et à quelques autres, que sont consacrées les sections qui suivent.

13.3 DÉCOMPOSITION DE L'ÉNONCÉ

L'énoncé est décomposé en propositions élémentaires décrivant chacune un concept ou un type de faits qu'on pourra traduire facilement en composants du schéma conceptuel.

13.3.1 Notion de proposition élémentaire

La forme standard que nous privilégierons correspond à la composition suivante³ :

sujet verbe objet

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 l'objet, 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 décomposée en trois propositions élémentaires :

3. Nous avons vu au chapitre 10 que l'expression sous forme de triplets est à la base des langages de description sémantique du Web, en particulier RDF (*Resource Description Framework*), où un fait élémentaire s'exprime sous la forme <sujet> <prédicat> <objet>.

- 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 ...)

13.3.2 Cardinalités

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 plusieurs⁴).

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*.

4. 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.

La réponse à de telles questions se trouve parfois dans la formulation même de la proposition. Des locutions telles que *tout*, *certains*, *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 à des informations complémentaires (interview dirigée), à notre connaissance du domaine d'application ou ... au bon sens !

13.3.3 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 :

- ma voiture a le numéro minéralogique HBG910-82

est une proposition *particulière* qui n'est qu'un exemple de la première⁵.

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 généralisation, l'existence d'un type de faits intéressant : si ma voiture a un numéro, n'en serait-il pas de même de toutes les voitures ?

13.3.4 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,

5. Notons que le terme de *proposition* ne fait pas référence au calcul propositionnel (annexe A24), auquel cas les propositions générales devraient s'appeler *prédicats*.

- 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 :

```
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 fausses⁶. 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 les remplacer. On ne peut donc pas les décomposer sans perdre de l'information.

Que faire alors si on désire s'en tenir à des propositions élémentaires⁷? 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 propositions⁸.

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.

6. 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 aussi comment un triangle accidentel (dont les trois côtés ne sont pas issus d'une même étoile d'origine) se transforme en une fausse étoile.

7. Le modèle Entité-association étendu (chapitre 16) permet de représenter explicitement des faits irréductibles basés sur plus de deux concepts.

8. Ce procédé, qui consiste à percevoir un concept abstrait comme un objet concret s'appelle *réification*. On le rencontrera encore plus loin dans ce chapitre.

13.3.5 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 précédemment. Considérons à titre d'exemple la phrase

- 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. À 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)⁹. De même, la seconde proposition suppose qu'un employé travaille dans *une seule* a, ce qui permet d'y repérer les deux propositions binaires *une entreprises ont des employés* et *un employé est inscrit à une 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.

13.4 PERTINENCE D'UNE PROPOSITION

Certaines propositions apportent clairement une contribution au schéma conceptuel. D'autres en revanche doivent être considérées comme du *bruit* et sont ignorées.

9. Ces raisonnements seront étudiés de manière plus détaillée au chapitre 18 à partir du principe de décomposition d'une relation (chapitre 3). On s'en tiendra ici à des principes intuitifs.

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

On précise une contrainte d'exécution d'activités de gestion qui sera surtout utile lors de la conception des programmes. Cependant, une analyse plus approfondie pourrait conduire à en retenir les propositions dérivées suivantes, permettant de gérer la règle d'antériorité suggérée :

- un véhicule a une date d'enregistrement,
- une prime a une date de paiement.

Exemple 3

- les véhicules que la compagnie assure ...

Étant donné qu'on ne s'intéresse qu'aux véhicules assurés par la compagnie, cette proposition n'apporte aucune information, sinon l'existence, probablement déjà connue, de la notion de véhicule.

13.5 REPRÉSENTATION D'UNE PROPOSITION

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 n'est pas déjà exprimé dans le schéma courant, ainsi que les liens qui l'unissent aux autres concepts. Nous donnerons ci-après 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.

13.5.1 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 suivant, il est raisonnable de représenter les dossiers par un type d'entités DOSSIER et les titres par un attribut Titre attaché au premier (figure 13.2) :

- tout dossier possède un titre

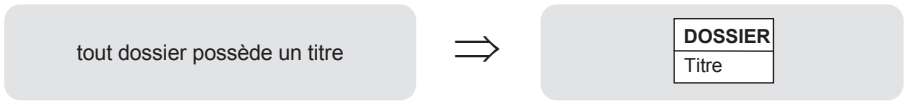


Figure 13.2 - Traduction d'une proposition exprimant un concept nouveau et une de ses propriétés

13.5.2 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 13.3. 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

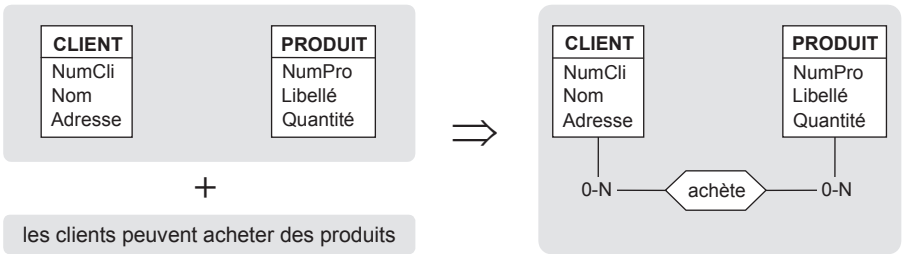


Figure 13.3 - La proposition se traduit par un type d'associations entre types d'entités existant

13.5.3 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 13.4 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 13.4 - La proposition se traduit par l'ajout d'un attribut à un type d'entités existant

13.5.4 Nouveau type d'entités et ses deux attributs

Si une proposition exprime un lien entre deux concepts se traduisant naturellement par deux attributs. On représente 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 identifié, se représente par les attributs Date et Montant qu'on affecte à un nouveau type d'entités représentant les *dépenses* (figure 13.5).

- le montant est dépensé à une date déterminée



Figure 13.5 - Représentation d'un lien entre concepts par un type d'entités

13.5.5 Restructuration pour intégration

L'intégration d'un nouveau type de faits 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 relier que deux types d'entités).

Les problèmes 1 et 2 se résoudraient si l'attribut A était au préalable transformé en un type d'entités. Les problèmes 3 et 4 se résoudraient si le type d'associations R était au préalable transformé en un type d'entités. Nous suggérerons donc deux *transformations* qui permettent de résoudre les problèmes évoqués précédemment.

Transformation d'un type d'associations en type d'entités

Cette transformation (figure 13.6) remplace le 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. Ce type d'entités n'a pas d'attribut.

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'entités.

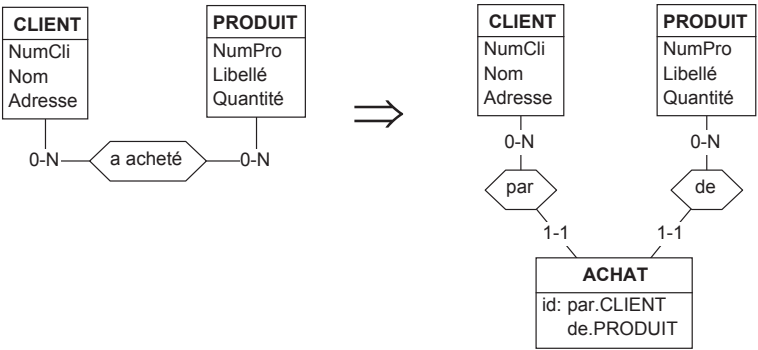


Figure 13.6 - Transformation du type d'associations *a acheté* en type d'entités ACHAT

On vérifiera, à l'aide de l'exemple de la figure 13.7 (pour laquelle nous avons simplifié la représentation graphique) que les deux schémas ont exactement le même contenu informationnel, au sens de la section 12.8 : il n'existe aucune question à laquelle l'un des schémas pourrait répondre et pas l'autre.

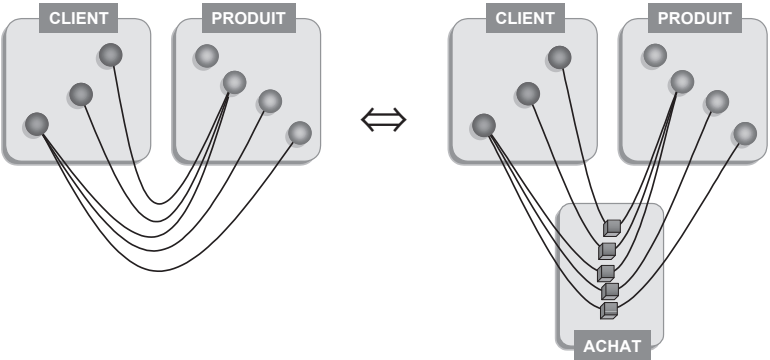


Figure 13.7 - Illustration de la transformation d'associations en entités

Transformation d'un attribut en type d'entités

La deuxième transformation (figure 13.8) extrait un attribut (ou 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*¹⁰. 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.

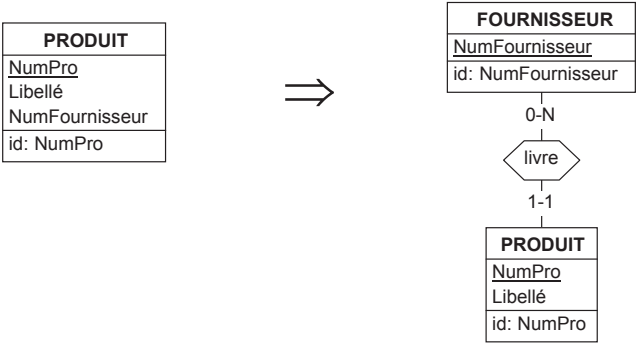


Figure 13.8 - Transformation d'un attribut en type d'entités

La figure 13.9 montre comment se distribuent les informations selon cette transformation. Ici encore, on observe l'équivalence des informations dans ces schémas.

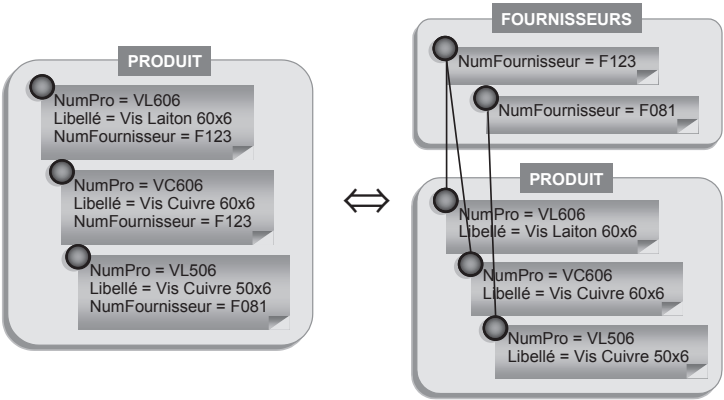


Figure 13.9 - Illustration de la transformation de valeurs d'un attribut en entités

On observe qu'il y a un et un seul fournisseur par produit (de même qu'il y a une et une seule valeur de NumFournisseur par entité PRODUIT) et qu'il peut y avoir plusieurs produits par fournisseur (de même que plusieurs entités PRODUIT peuvent 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

10. 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 ce chapitre. Nous reviendrons sur cette *tolérance*.

schéma de droite.

Application

Le schéma de la figure 13.10 exprime que *des unités (UNITE) fabriquent des types de pièces (PIECE)* et qu'*un type de pièces est stocké dans un dépôt (NomDépôt)*.

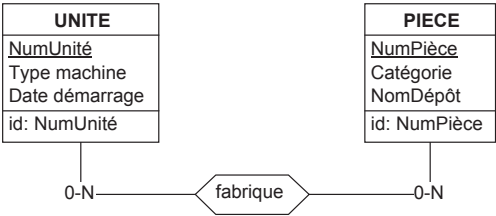


Figure 13.10 - Le schéma avant enrichissement

- Supposons 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.*

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 13.11). Il est alors permis d'ajouter les trois nouveaux attributs (figure 13.12).

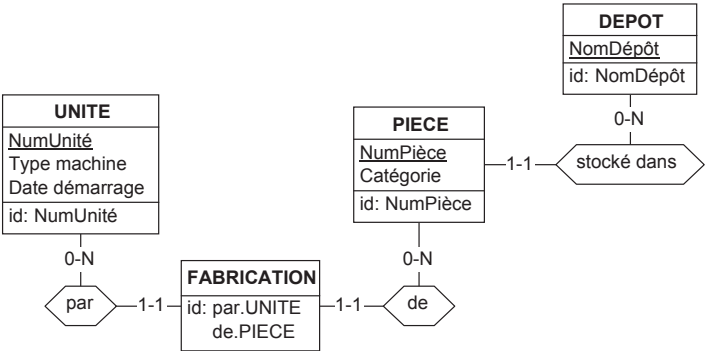


Figure 13.11 - Le schéma est transformé afin d'accueillir les nouveaux attributs

13.5.6 Propriétés multivaluées

Un problème particulier va se poser lors de la mise en évidence de **propriétés multivaluées** attachées à un concept. Considérons deux cas représentatifs :

- une personne a des prénoms
- un ouvrage a des mots-clés

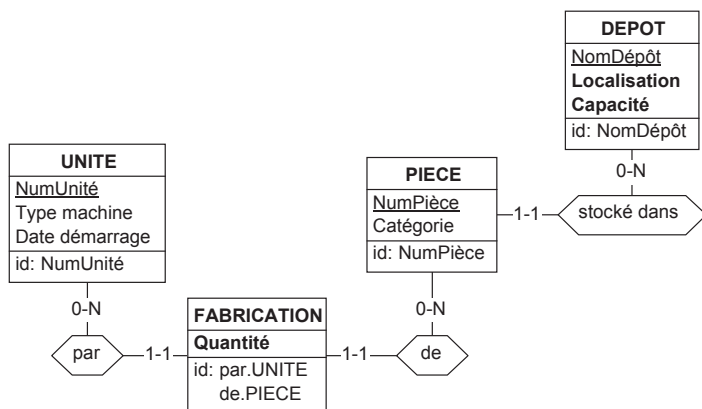


Figure 13.12 - Les nouveaux attributs ont été ajoutés

Un concept majeur, représenté par un type d'entités tel que PERSONNE ou OUVRAGE, est caractérisé par une propriété (*prénom*, *mot-clé*) se présentant non pas comme une simple valeur, mais comme un **ensemble de valeurs**. Le modèle de base utilisé dans ce chapitre ne permet pas d'exprimer directement ces propriétés multivaluées¹¹. Si nous ne pouvons attacher plusieurs valeurs d'un même attribut à une entité **e**, nous pouvons en revanche attacher plusieurs entités à cette entité **e**. C'est donc ainsi que nous allons résoudre ce problème de modélisation. Nous suggérons donc 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). La figure 13.13 illustre la représentation de la propriété **mot-clé** par un type d'entités. Il faut cependant être attentif à la signification de ce type d'entités. Il existe en effet deux représentations distinctes.

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 13.13, 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 pour laquelle Libellé = "jardinage", quel que soit le nombre d'ouvrages disponibles sur ce thème.

Représentation des instances

La seconde technique (figure 13.13, droite) représente *la présence de chaque mot-clé* décrivant chaque ouvrage par une entité distincte du type d'entités DESCRIPTION doté d'un attribut Libellé. C'est ainsi qu'il y aura autant d'entités DESCRIPTION dont Libellé = "jardinage", qu'il y a d'ouvrages de jardinage. Les mots-clés d'un ouvrage étant distincts, on en déduit l'identifiant de DESCRIPTION.

11. En revanche, le modèle Entité-association étendu du chapitre 16 offrira le concept d'*attribut multivalué*. Ce dernier réclamera cependant un traitement complexe lors de la production du schéma SQL, ce qui justifie sa mise à l'écart dans le modèle de base.

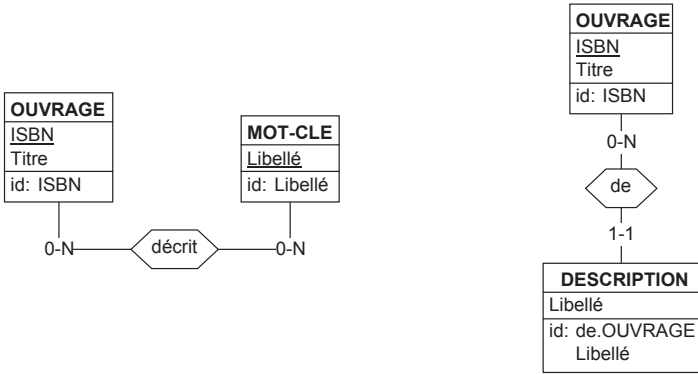


Figure 13.13 - Deux techniques de représentation d'une propriété multivaluée par un type d'entités¹²

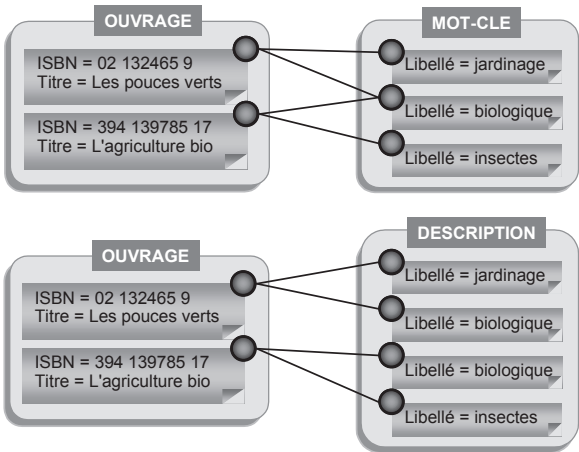


Figure 13.14 - Illustration d'une propriété multivaluée sous forme d'entités

La figure 13.14 montre sur un exemple la nature différente des entités MOT-CLE et DESCRIPTION. Cette représentation peut paraître lourde et maladroite. Dans le cas des prénoms par exemple, 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 sera peut-être tenté de lui préférer une représentation plus modeste (figure 13.15), 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 importants qui nous feront préférer la solution initiale, malgré sa lourdeur apparente.

12. Ces deux schémas sont *presque* équivalents : à gauche, un mot-clé peut exister indépendamment des ouvrages alors qu'à droite, seuls les mots-clés attachés à un ouvrage sont représentés.

PERSONNE	PERSONNE
NumPers	NumPers
Nom	Nom
Prénom1[0-1]	Prénoms
Prénom2[0-1]	
Prénom3[0-1]	
Prénom4[0-1]	

Figure 13.15 - À éviter : deux représentations problématiques d'une propriété multiple

13.5.7 Recommandations pratiques

Nous terminerons cette section par quelques recommandations pratiques.

- Il est parfois utile de faire un repérage préliminaire des concepts importants avant même l'analyse fine des propositions élémentaires. Lors d'une première lecture du texte, on souligne les mots importants et on les exprime sous forme de types d'entités. La suite de l'analyse confirmera, infirmera ou corrigera ce premier schéma et lui ajoutera les attributs, les types d'associations et les contraintes.
- 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 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 13.9.
- 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 directement 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.

13.6 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 structurelle** le phénomène selon lequel une construction du schéma courant et une proposition expriment le même type de faits. Nous distinguerons divers cas de figure.

13.6.1 Redondance explicite

Dans les cas simples, la redondance apparaît explicitement, schéma et proposition exprimant le type de faits de manière similaire. Dans la figure 13.16, il est clair que la proposition est déjà représentée dans le schéma courant, et doit être ignorée.

13.6.2 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 13.17, le schéma indique qu'*un client possède des véhicules* ; la proposition exprime la même idée, mais relativement aux véhicules. Elle est donc redondante.

13.6.3 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 13.18 montre un premier exemple. La proposition parle des employés des départements, alors que le schéma, qui inclut pourtant les types d'entités **DEPARTEMENT** et **EMPLOYEE**, n'exprime aucun lien direct entre ceux-ci. À 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.

L'exemple de la figure 13.19 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.

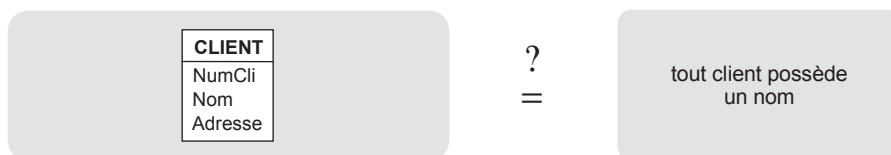


Figure 13.16 - La proposition est explicitement redondante par rapport au schéma



Figure 13.17 - La proposition décrit une variante du type de faits exprimé dans le schéma

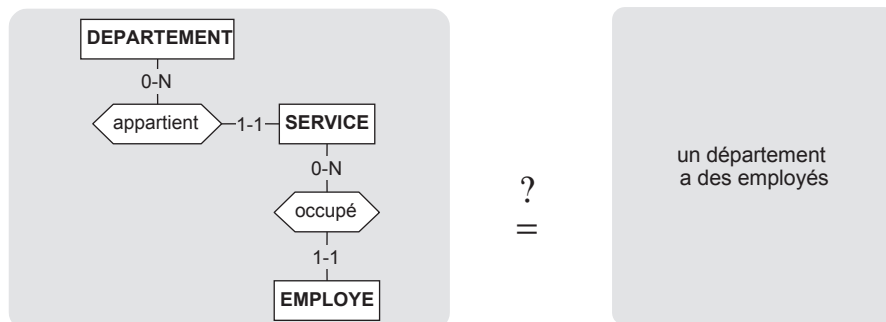


Figure 13.18 - La proposition recouvre un type de faits dérivable (par composition) des constructions de schéma courant

13.6.4 Redondance apparente

En revanche, la redondance peut n'être qu'apparente, et par conséquent inexistante. L'exemple de la figure 13.20 suggère que le concept d'adresse de client est déjà présent dans la schéma courant. À 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.

13.6.5 Synonymes et homonymes

Dans l'analyse du domaine d'application, deux phénomènes vont apparaître, qui devront être identifiés par une compréhension en profondeur 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. On choisira une et une seule dénomination pour chaque concept.

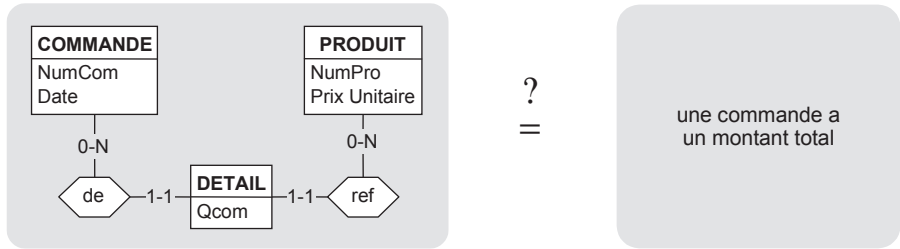


Figure 13.19 - La proposition exprime une propriété des commandes qui est dérivable des constructions existant dans le schéma courant

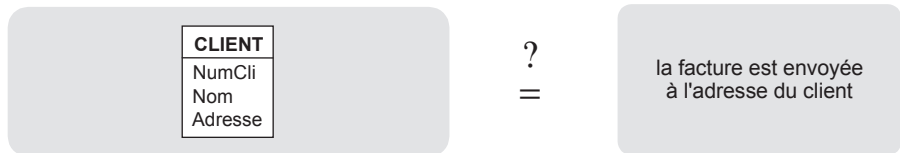


Figure 13.20 - Il faut distinguer *adresse d'expédition* (dans le schéma) et *adresse de facturation* (dans la proposition)

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 précédemment, 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.

13.7 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 ». À 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 12.18, qui indique que ce numéro est unique pour les contrats d'un même client seulement.

Dans les deux exemples de la figure 13.21, 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.

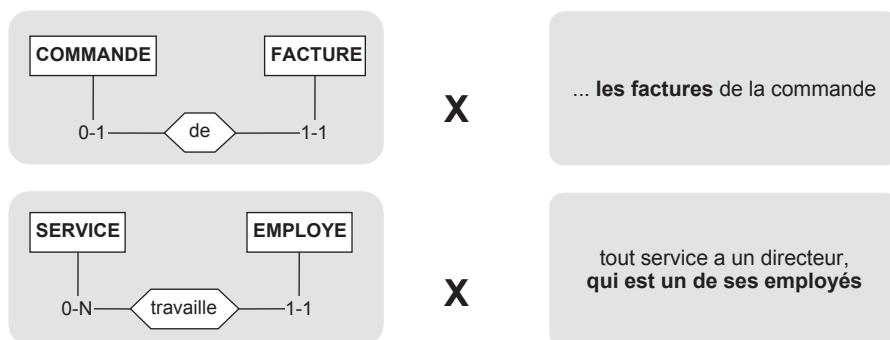


Figure 13.21 - Deux exemples de contradiction

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.

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 que nous étudierons plus en détail au chapitre 18, section 18.9. Nous nous contentons ici d'attirer l'attention du lecteur sur le problème.

13.8 LES CONTRAINTES D'INTÉGRITÉ

Le plus souvent, c'est l'énoncé lui-même qui spécifiera les 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 de base et additionnelles 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 nécessaires 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.

13.9 NORMALISATION DU SCHÉMA

À ce stade, le schéma obtenu est présumé correct. On peut souvent effectuer un *nettoyage* de celui-ci, soit dans un but de simplification, soit pour éliminer certaines anomalies qui pourraient subsister.

13.9.1 Simplification du schéma

Il s'agit ici de s'assurer que l'information a été représentée de manière adéquate, conforme à la philosophie du modèle Entité-association. Observons à titre de premier exemple le type d'entités LOCALITE du schéma de gauche de la figure 13.22. À chaque entité est associée une valeur représentant le nom d'une localité. D'autre part, une entité LOCALITE est associée à chaque entité CLIENT.

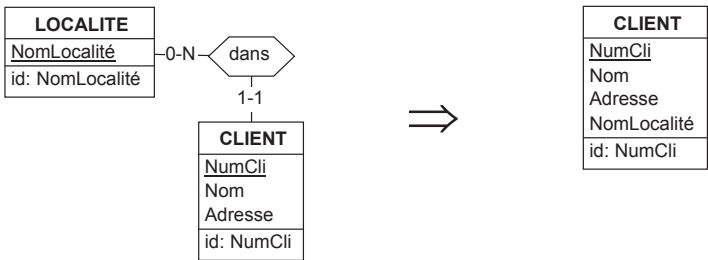


Figure 13.22 - Si le type d'entités LOCALITE n'a pas d'autre raison d'être que d'indiquer la localité de chaque client, alors on peut le remplacer par un simple attribut.

Au total, LOCALITE n'a d'autre intérêt que d'associer un nom de localité à chaque entité CLIENT. On pourrait donc remplacer ce type d'entités, sans y perdre¹³, par un simple attribut de CLIENT (schéma de droite). La simplification consiste donc 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. Un tel type d'entités est appelé *type d'entités attribut*. De même, un schéma pourra contenir un type d'entités sans attributs, connecté à deux autres types d'entités, et identifié par ceux-ci (ou implicitement par l'un d'entre eux seulement). Ainsi, le type d'entités EXPORTATION de la figure 13.23 peut-il être remplacé par le type d'associations *exporte*. EXPORTATION est un *type d'entités association*.

13.9.2 Élimination des redondances internes

La redondance interne est une erreur très fréquente chez les débutants. Elle est illustrée par la figure 13.24 (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. Tous les employés qui appartiennent au même département ont donc même localisation.

13. Remarquons que l'équivalence n'est pas totale, à cause de la cardinalité 0-N à gauche, qui permet de représenter les localités sans client.

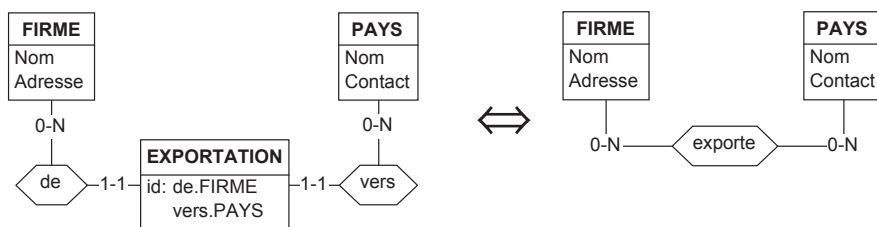


Figure 13.23 - Si le type d'entités EXPORTATION sert uniquement de lien entre les types d'entités FIRME et PAYS, alors on peut le remplacer par un type d'associations

Il en résulte qu'on enregistrera l'adresse d'un département déterminé autant de fois que ce dernier possède d'employés, ce qui constitue une redondance d'information, qu'on qualifiera d'interne, car elle est locale aux attributs d'un type d'entités. Il y a à l'origine une erreur qui consiste à affecter Localisation à EMPLOYE, alors que cet attribut dépend en fait de NomDépart. La situation est alors celle d'un attribut (NomDépart) auquel on tente d'affecter un autre attribut (Localisation), situation qui a été traitée à la figure 13.8. Le schéma de droite de la figure 13.24 montre comment cette anomalie peut être éliminée par l'extraction des attributs litigieux sous la forme d'un nouveau type d'entités.

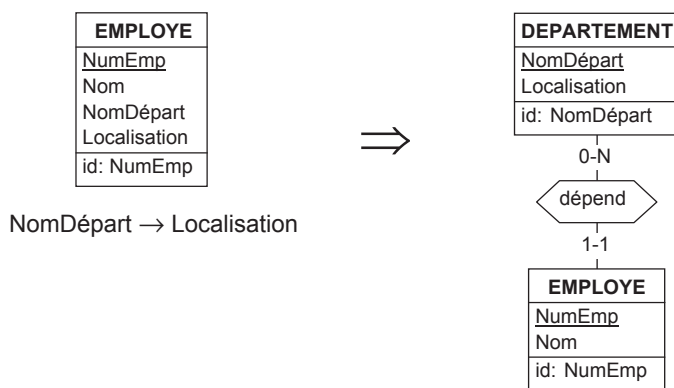


Figure 13.24 - Élimination d'une redondance interne. L'expression "NomDépart → Localisation" indique qu'à une même valeur de NomDépart sera toujours associée la même valeur de Localisation

Nous proposerons un autre exemple de ce type d'anomalie, basé sur le schéma 12.26 relatif à la gestion d'une bibliothèque. Supposons que l'analyse du domaine d'application conduise au schéma de gauche de la figure 13.25, 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 répètera 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.

Le lecteur attentif aura certainement fait le lien entre ce type de situation et le principe de normalisation d'une table (section 2.11).

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.

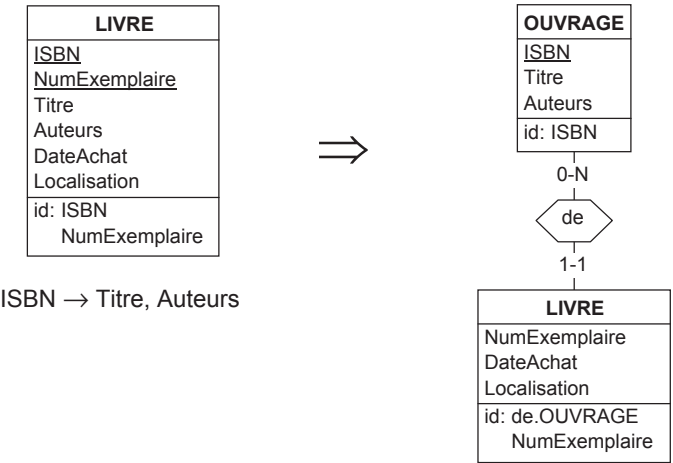


Figure 13.25 - Un autre cas d'élimination d'une redondance interne

13.10 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 suivante¹⁴ :

- 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...

Lorsque le schéma a été construit à partir d'énoncés en langage courant, il conviendrait d'associer les fragments de ces énoncés aux objets du schéma qu'ils ont permis de définir. Cette étape est essentielle pour l'évolution du schéma. Il n'est pas certain en effet que la personne qui modifiera notre base de données dans quelques années percevra le domaine d'application comme nous l'avons fait. Enfin, rédiger une définition précise est un processus de validation du schéma très efficace, particulière-

14. Nous désignerons plus loin ces définitions de *sémantique externe* du schéma (section 16.3).

ment chez les concepteurs débutants : un concept peu clair ou mal défini est impossible à décrire de manière précise.

13.11 COMPLÉTUDE DU SCHÉMA

Certaines parties du schéma ne sont peut-être encore qu'à l'état d'ébauche. On les complètera 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).

13.12 QUE RETENIR ?

Une base de données est construite à partir du schéma conceptuel qui décrit les concepts du domaine d'application. Il est donc important que ce schéma représente naturellement, complètement et sans redondance ce domaine d'application.

La connaissance de ce dernier s'appuie généralement sur un ensemble de textes qui décrivent les concepts du domaine, leurs propriétés et leurs associations. Chaque texte est décomposé en **propositions élémentaires** obéissant pour la plupart à la forme binaire *sujet verbe objet* établissant une relation entre deux concepts (le sujet et l'objet). Certaines propositions associant plus de deux concepts ne sont pas immédiatement réductibles en propositions binaires. On sera alors amené à définir un nouveau concept correspondant au verbe de la proposition et ainsi à décomposer la proposition originale en un ensemble de propositions binaires.

De l'**interprétation** de ces propositions on tire les types d'entités, les attributs, les types d'associations et les contraintes du schéma conceptuel. On vérifie au préalable que la proposition est pertinente (elle nous apporte une information utile), qu'elle est non redondante (elle nous apporte une information nouvelle) et qu'elle n'entre pas en contradiction avec les informations déjà introduites dans le schéma.

L'ajout à un schéma des objets identifiés par une proposition pertinente peut exiger de **restructurer** certains objets de ce schéma. C'est ainsi qu'on sera amené à transformer des attributs en types d'entités ou des types d'associations en types d'entités.

La **normalisation** du schéma consiste à simplifier certaines structures (*types d'entités attribut* et *types d'entités association* notamment). On décomposera également les types d'entités qui sont le siège de redondances internes.

Enfin, on veillera à **documenter** soigneusement tous les objets du schéma en spécifiant la signification des instances de chacun.

13.13 POUR EN SAVOIR PLUS

Les raisonnements et les techniques présentés dans ce chapitre constituent une méthode simple mais suffisante pour construire le schéma conceptuel d'une petite base de données. Les projets de plus grande ampleur vont exiger, non seulement un modèle plus riche, mais aussi des techniques et des méthodes de construction de schémas conceptuels plus élaborées. L'analyse de sources d'information plus diversifiées et plus complexes, la normalisation de schémas, la résolution de conflits ne pourront plus s'appuyer exclusivement sur les raisonnements intuitifs et de simple bon sens que nous avons développés dans ce chapitre. D'autre part, le schéma conceptuel sera en général construit par la fusion de schémas partiels. Ce processus, dit *intégration de schémas*, réclamera lui aussi des techniques et des raisonnements spécifiques que nous n'avons pas abordés dans le chapitre. Le chapitre 18 prendra donc le relai de celui-ci pour donner au lecteur les ressources nécessaires pour aborder la conception de schémas conceptuels de grande taille.

13.14 EXERCICES

- 13.1 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 du 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. À tout instant, un support peut être loué par un membre du club.

13.2 Proposer un schéma conceptuel qui représente le domaine d'application suivant :

Les patients de l'hôpital sont répartis dans les services (caractérisés chacun par un nom identifiant, sa localisation, sa spécialité) de ce dernier. Chaque médecin appartient à un service. Il est identifié par son nom et son prénom. Un patient passe des visites. Chaque visite est effectuée chez un médecin à une date déterminée. Un patient ne peut passer plus d'une visite par jour. Lors d'une visite, une ou plusieurs prescriptions peuvent être rédigées. Chaque prescription mentionne le nom d'un médicament et la posologie à respecter par le patient (posologie = simple ligne de texte). Il est évident que le médecin ne prescrit pas deux fois le même médicament lors d'une même visite ! Un patient possède un numéro d'inscription (identifiant), un nom, un prénom et une adresse.

13.3 Proposer un schéma conceptuel qui représente le domaine d'application suivant :

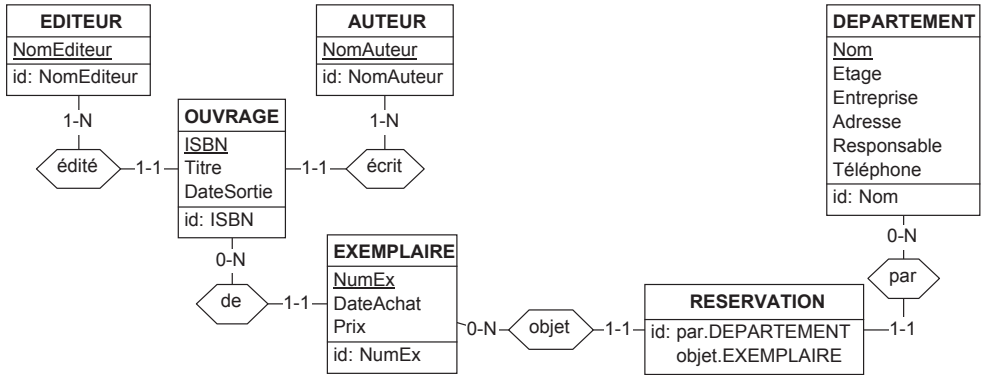
*Une entreprise de distribution dispose d'un certain nombre de véhicules (identifiés par leur numéro et caractérisés par leur capacité et le nom du conducteur). Chaque jour, chaque véhicule effectue une (et une seule) tournée de distribution, d'une longueur déterminée. Durant cette tournée, le véhicule 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.*

13.4 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.*

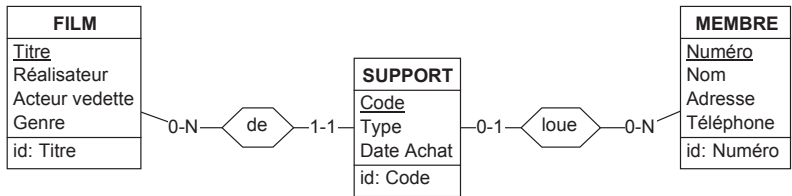
13.5 Normaliser le schéma suivant. Parmi les attributs de **DEPARTEMENT**, on observe les dépendances suivantes :

Entreprise $\longrightarrow$ Adresse, Responsable; Responsable $\longrightarrow$ Téléphone

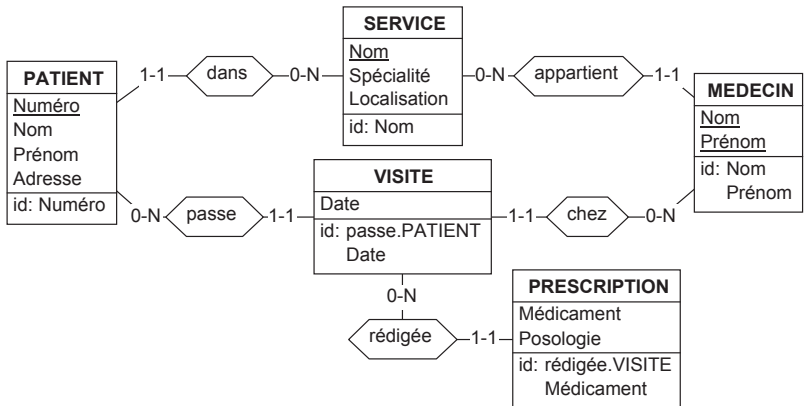


Exercices corrigés

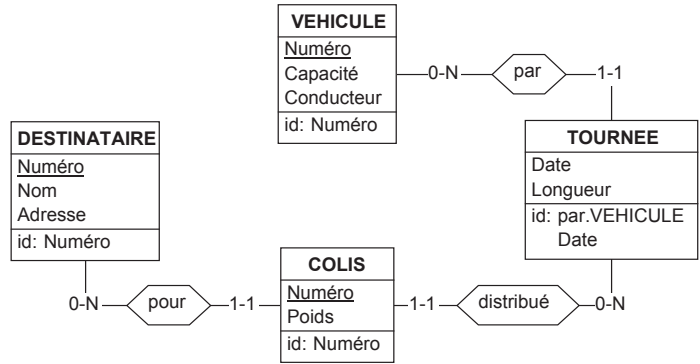
• **Exercice 12.1**



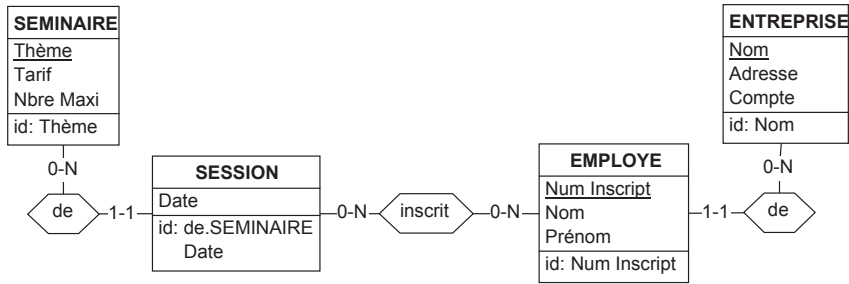
• **Exercice 12.2**



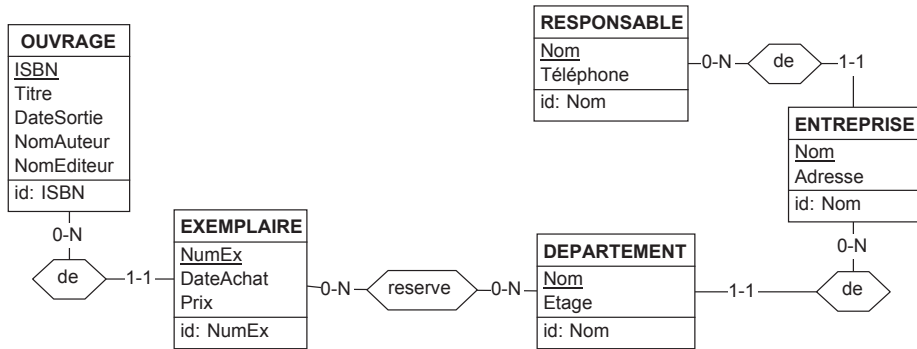
• Exercice 12.3



• Exercice 12.4



• Exercice 12.5



Des exercices et solutions additionnels sont disponibles à l'annexe A12.

Chapitre 14

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 DDL). Pratiquement, on traduira chaque composant du schéma Entité-association en structures de tables, colonnes, identifiants et clés étrangères. Ces structures seront ensuite exprimées dans le langage SQL.

14.1 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.

Ces structures relationnelles, décrites au chapitre 2, doivent constituer une traduction fidèle, complète et exacte du schéma conceptuel. Nous proposerons des règles de traduction systématiques de chaque type de composants Entité-association en structures de bases de données. Les structures ainsi obtenues seront alors exprimées aisément en SQL. Nous élaborerons une procédure simplifiée adaptée au modèle Entité-association de base. En particulier, nous ne nous préoccupons pas de critères de performance, qui seront abordés aux chapitres 19 et 20.

14.2 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 14.1). Chaque entité de ce type sera décrite par une ligne de cette table. Conformément aux conventions graphiques du chapitre 2, les tables sont représentées par un rectangle ombré.



Figure 14.1 - Représentation d'un type d'entités par une table. À chaque attribut correspond une colonne

14.3 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 14.1). 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.

14.4 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*.

14.4.1 Types d'associations un-à-plusieurs (1:N)

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. 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 l'exemple du type d'associations occupe entre DEPARTEMENT et EMPLOYE (figure 14.2). 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, un arc orienté indique la présence d'une **contrainte référentielle** à laquelle est soumise la clé étrangère (ici *NOMDEPART*).

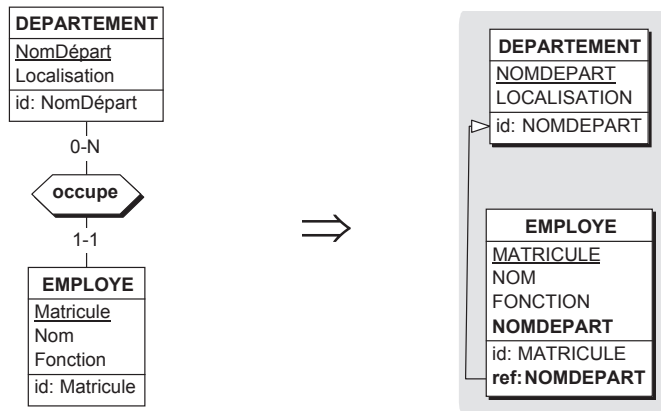


Figure 14.2 - Représentation d'un type d'associations *un-à-plusieurs* par une clé étrangère

Justification

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* ;
2. à chaque entité *DEPARTEMENT* il associe un nombre quelconque [0-N] d'entités *EMPLOYE*.

Que décrit le schéma de droite ?

1. 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).
2. 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 étrangère¹. 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*²). La figure 14.3 propose un exemple de ces deux situations.

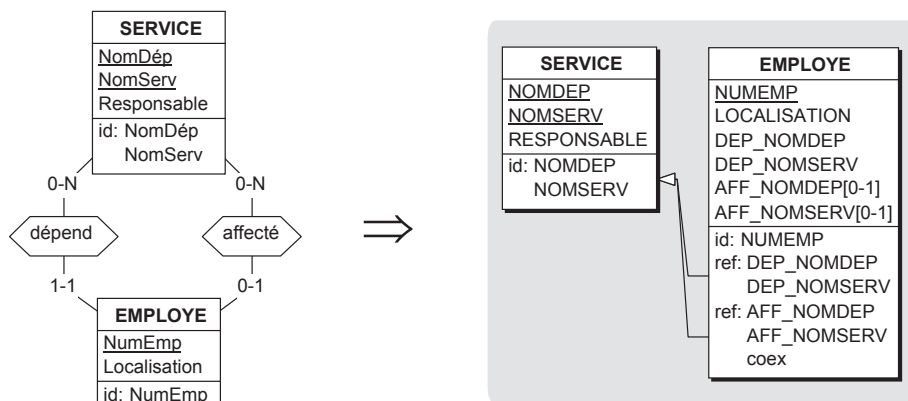


Figure 14.3 - Traduction sous forme d'une clé étrangère multi-composant

Les identifiants hybrides

Qu'en est-il des **identifiants hybrides**, qui comprennent un ou plusieurs types d'entités associés, lorsqu'on traduit les types d'associations en clés étrangères ? Considérons par exemple le schéma de la figure 12.25, dont un fragment est repris à la figure 14.4. 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 de DIRECTION par l'attribut NOMDIR de la clé étrangère qui vient d'être ajoutée. 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, son rôle dans le type d'associations ou encore le nom de son identifiant primaire.

14.4.2 Types d'associations un-à-un (1:1)

La traduction est similaire à celle du cas précédent, mais elle admet deux variantes.

1. Le non respect de cette règle est très fréquent chez les débutants. Il témoigne d'une mauvaise compréhension du concept de clé étrangère.
2. La contrainte de *coexistence* stipule que les composants, ici deux colonnes, ont pour toute ligne, soit chacun une valeur, soit chacun la valeur null. Elle sera étudiée au chapitre 16.

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.³

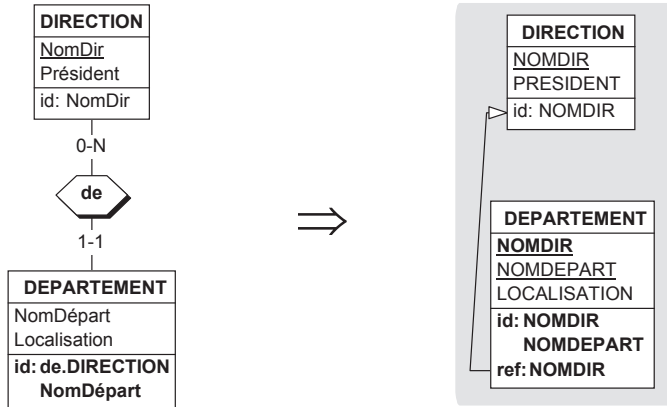


Figure 14.4 - Traduction d'un identifiant hybride : chaque composant *type d'entités* y est remplacé par l'identifiant de celui-ci

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 identifiante. Si l'identifiant de la table référencée est constitué de plusieurs attributs, la clé étrangère sera constituée d'autant d'attributs.

Plus précisément

- Si R est obligatoire pour A et facultatif pour B, on ajoutera à TA une nouvelle colonne obligatoire RB, de même type que IB, et qu'on déclare clé étrangère vers TB.
- Si R est facultatif pour A et B, on procède comme précédemment : 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(nt) en outre un **identifiant** supplémentaire pour sa table.
- Le cas où R est obligatoire pour A et B n'est pas admis dans le modèle de base. Il sera étudié au chapitre 19.

La figure 14.5 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.

3. L'un des types d'entités peut n'avoir qu'un identifiant implicite (voir section 12.6.3).

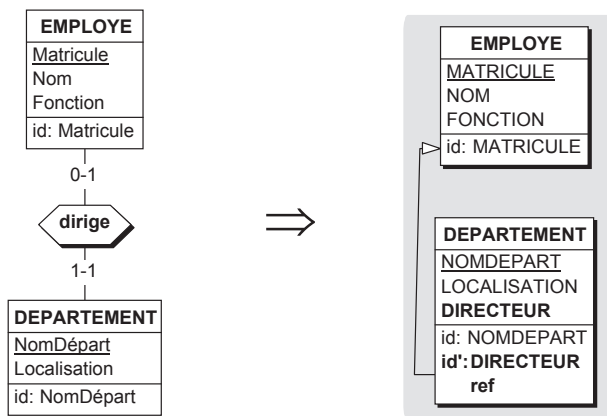


Figure 14.5 - Représentation d'un type d'associations *un-à-un* par une clé étrangère identifiante

Justification

Cet exemple nous permet de justifier les règles proposées précédemment. Supposons qu'on représente *dirige* par une colonne **DIRECTEUR_DE** ajoutée à la table **EMPLOYE**, 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 9.12) ;
- nous définissons aussi un identifiant facultatif, qui peut poser des problèmes de mise en œuvre dans certains SGBD ;
- enfin, cette clé étrangère doit être soumise à une contrainte très difficile à contrôler : à chaque ligne de **DEPARTEMENT** doit correspondre une ligne d'**EMPLOYE** dont les valeurs de **NOMDEPART** et **DIRECTEUR_DE** sont égales. SQL est incapable de prendre en charge une telle contrainte.

14.4.3 Types d'associations plusieurs-à-plusieurs (N:N)

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 section 13.5.5, figure 13.6, à la transformation de **R** en un type d'entités **R'** et deux types d'associations *un-à-plusieurs*. On poursuit ensuite comme décrit précédemment en représentant **R'** par une table et les deux types d'associations par des clés étrangères.

La figure 14.6 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**.

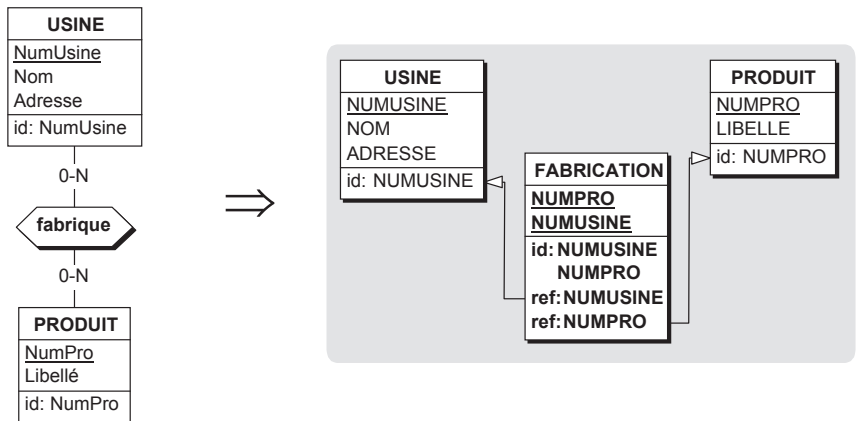


Figure 14.6 - Représentation d'un type d'associations *plusieurs-à-plusieurs* par une table associative munie de deux clés étrangères. Le procédé dérive de celui qui a été décrit à la figure 13.6.

14.4.4 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 précédemment 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* supervisé de la figure 14.7, qui suit les règles énoncées en 14.4.1.

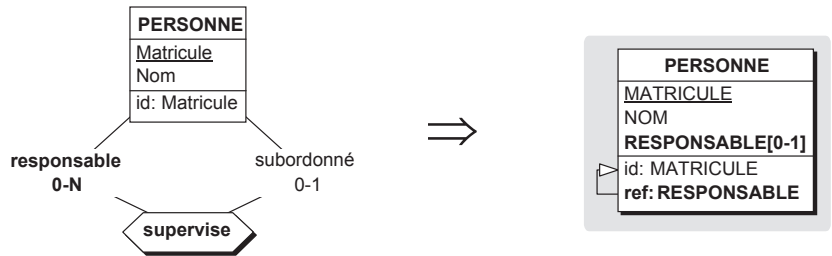


Figure 14.7 - La représentation d'un type d'association *un-à-plusieurs* cyclique obéit aux mêmes règles que celles des autres types d'associations

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 14.4.3 sont d'application (figure 14.8).

14.5 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.

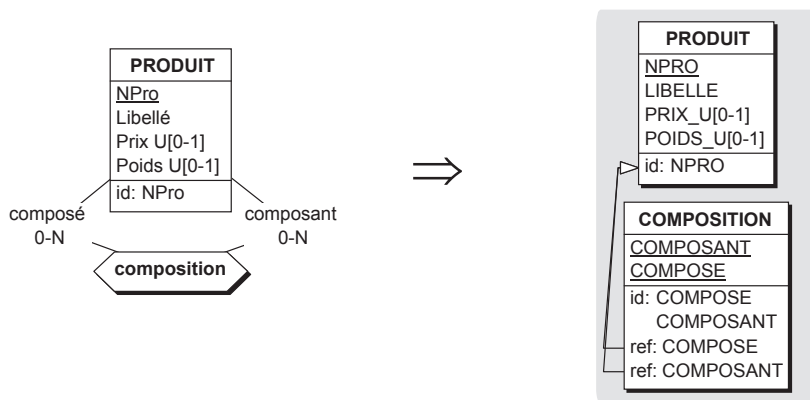


Figure 14.8 - La représentation d'un type d'associations cyclique *plusieurs-à-plusieurs* passe par la définition d'une table associative

Remarque pratique

On veillera à ne pas confondre deux situations qui pourraient sembler similaires tout en étant fort différentes⁴ :

- un identifiant constitué de deux (ou plusieurs) composants (figure 14.6 par exemple) ;
- une table possédant deux (ou plusieurs) identifiants (figure 14.5 par exemple).

Les conventions graphiques distinguent clairement ces deux cas.

14.6 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 admissibles. 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

4. De même qu'il ne faut pas confondre 1 véhicule à 4 roues avec 4 véhicules à 1 roue.

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.

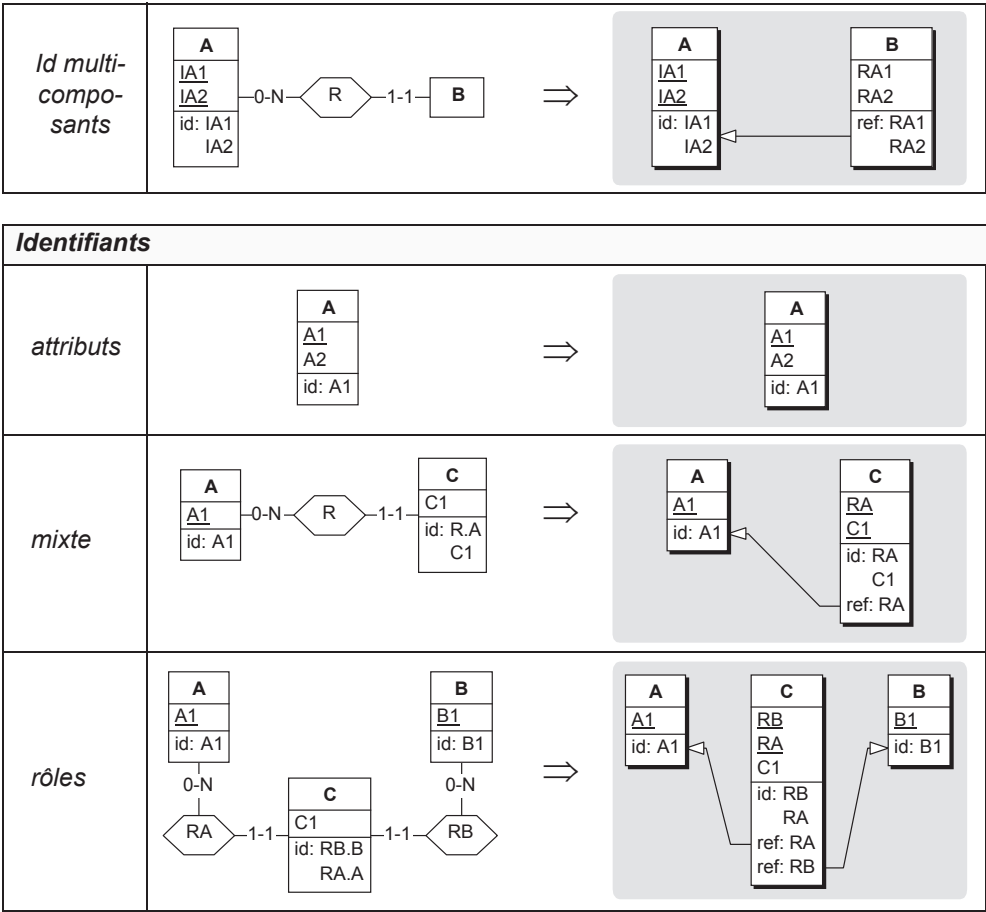
Exemple

L'application de ces règles de traduction nous permet de construire le schéma des tables correspondant au schéma conceptuel de la figure 12.18. Les deux schémas sont repris à la figure 14.9. 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) ?

14.7 SYNTHÈSE DES RÈGLES DE TRADUCTION

Les tableaux ci-après récapitulent les principales règles de traduction d'un schéma conceptuel en structures de tables.

Types d'entités et attributs		
	A A1 A2 A3[0-1] ⇒ A A1 A2 A3[0-1]	
Types d'associations		
1:N	A <u>IA</u> id: IA 0-N R 1-1 B ⇒ A <u>IA</u> id: IA ← B RA ref: RA	
1:1	A <u>IA</u> id: IA 0-1 R 1-1 B ⇒ A <u>IA</u> id: IA ← B <u>RA</u> id: RA ref	
N:N	A <u>IA</u> id: IA 0-N R 0-N B <u>IB</u> id: IB ⇒ A <u>IA</u> id: IA ← R <u>RB</u> <u>RA</u> id: RB RA ref: RA ref: RB → B <u>IB</u> id: IB	



14.8 LES STRUCTURES PHYSIQUES

Nous avons brièvement décrit dans le chapitre 2 (section 2.12) des constructions physiques destinées à définir l'implantation des données dans une machine et en particulier à contrôler les performances (notamment le temps d'exécution) des requêtes. Afin d'illustrer la définition de ces structures, nous ajouterons au schéma de tables de la figure 14.9 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 (voir chapitre 19). 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*.

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.

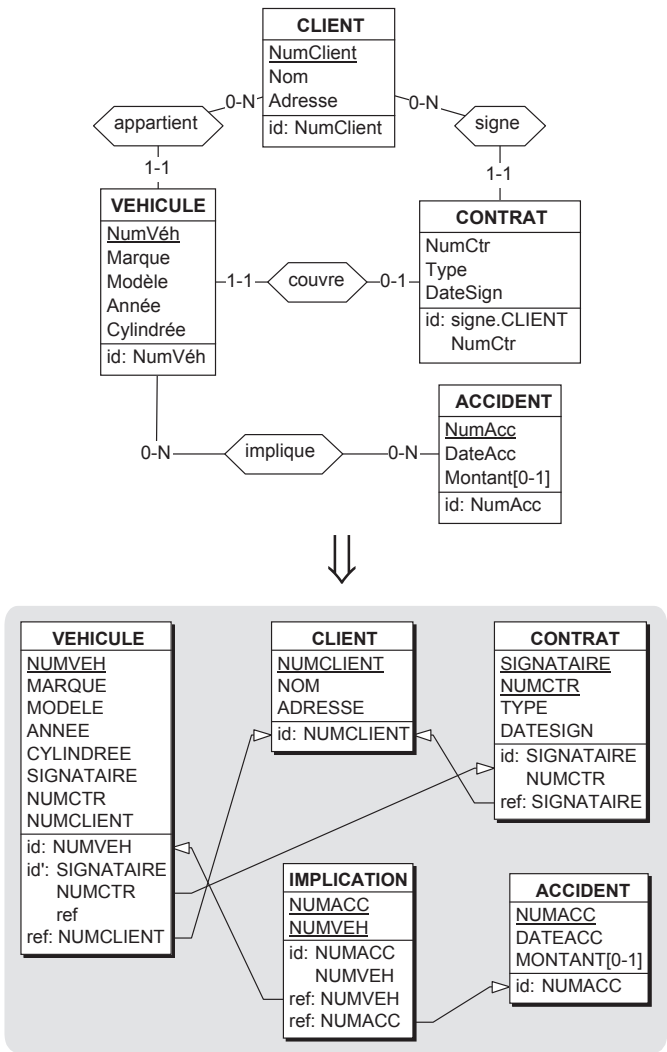


Figure 14.9 - Traduction d'un schéma conceptuel en structure de tables

Le schéma de la figure 14.10 a été obtenu à partir du schéma précédent (14.9, bas), auquel les index suggérés précédemment 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 ce chapitre (voir en particulier la section 20.4.3), 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 (chapitre 5).

14.9 TRADUCTION DES STRUCTURES EN SQL

La traduction de structures de tables en SQL est immédiate. Elle obéit aux règles énoncées au chapitre 6, 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 14.10. Seuls deux index ont été traduits, à titre d'illustration.

La question des modes de suppression et de modification attachés aux clés étrangères est liée au type de dépendance des lignes de la table référençante. Il paraît naturel d'imposer aux deux clés étrangères de la table IMPLICATION les modes on delete cascade et on update cascade. Quant aux trois autres clés étrangères, on suggère les modes on delete no action et on update cascade.

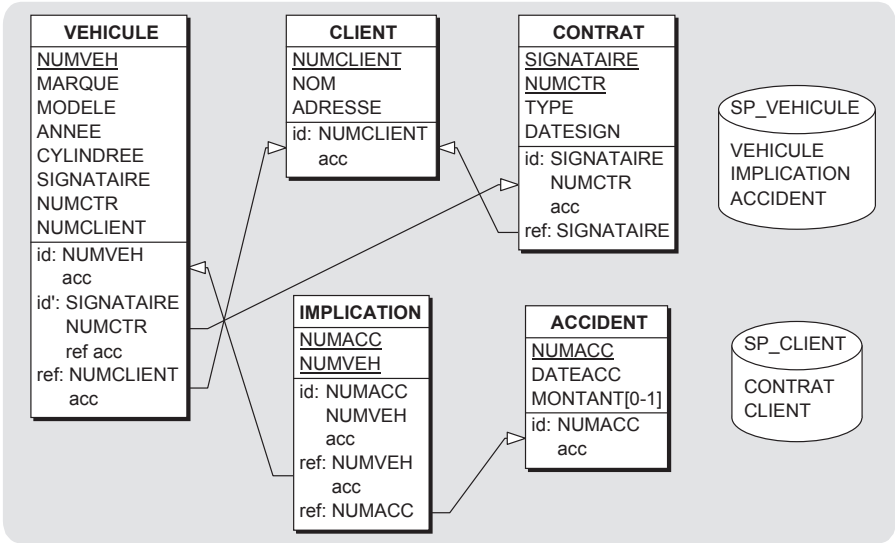


Figure 14.10 - Le schéma relationnel a été enrichi par l'adjonction d'index et d'espaces de stockage

```

create schema ASSURANCE;

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
                          on delete no action on update cascade)
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
                          on delete no action on update cascade,
                       foreign key (SIGNATAIRE, NUMCTR) references CONTRAT
                          on delete no action on update cascade)
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
                              on delete cascade on update cascade,
                           foreign key (NUMACC) references ACCIDENT
                              on delete cascade on update cascade)
in SP_VEHICULE;

create unique index XCLI_NUMCLI on CLIENT(NUMCLI);
...
create index XIMPL_NUMVEH on IMPLICATION(NUMVEH);

```

14.10 QUE RETENIR ?

Le schéma d'une base de données est obtenu à partir de son schéma conceptuel en deux phases : traduction du schéma en structures de tables puis expression de ces structures en SQL DDL.

La production des structures de tables se base sur un ensemble réduit de règles selon lesquelles :

1. un type d'entités est représenté par une table ;
2. un attribut est représenté par une colonne ;
3. un type d'associations *un-à-plusieurs* est représenté par une clé étrangère ;
4. un type d'associations *un-à-un* est représenté par une clé étrangère identifiante ;
5. un type d'associations *plusieurs-à-plusieurs* est représenté par une table associative incluant deux clés étrangères ;
6. les identifiants composés d'attributs sont traduits en identifiants de table composés des colonnes correspondantes ;
7. dans un identifiant hybride, chaque composant *type d'entités* est remplacé par la clé étrangère référençant la table qui le représente ;
8. les noms sont modifiés pour respecter les conventions syntaxiques d'SQL ;
9. un index est créé pour chaque identifiant et pour chaque clé étrangère.

14.11 POUR EN SAVOIR PLUS

Les chapitres 19 à 20 détailleront les principes et les règles décrits dans ce chapitre. Ils traiteront en particulier de la traduction des extensions du modèle Entité-association qui seront décrites au chapitre 16. Le chapitre 19 est consacré à la traduction d'un schéma Entité-association en structures relationnelles et relationnelle objet. La détermination des index et des autres constructions physiques est abordée au chapitre 19 et la production du code DDL est détaillée au chapitre 20.

14.12 PRODUIRE UN SCHÉMA DE BASE DE DONNÉES À L'AIDE D'UN OUTIL

L'atelier DB-MAIN offre plusieurs méthodes de production d'un schéma de tables. Dans le cadre de la méthode simplifiée de ce chapitre, nous procéderons comme suit.

Transformation du schéma conceptuel

1. *On prend une copie du schéma conceptuel* : dans la fenêtre du projet, on sélectionne le schéma conceptuel et on exécute la fonction : Product / Copy schema. On assigne la valeur Relationnel au champ Version de ce nouveau schéma.
2. *On transforme ce schéma selon le modèle relationnel* : on ouvre ce schéma et on exécute la fonction Transform / relational model.

On obtient ainsi un schéma proche de celui de la figure 14.10, tout en conservant le schéma conceptuel source.

Génération du code SQL DDL

On ouvre le schéma relationnel et on exécute la fonction File/Generate/Standard SQL ou /SQLite pour ensuite travailler avec SQLfast. Le résultat est stocké sous la forme d'un fichier texte qui peut ensuite être soumis à SGBD relationnel.

14.13 EXERCICES

- 14.1 Construire le schéma des tables correspondant au schéma conceptuel d'une structure administrative de la figure 12.25.
- 14.2 Dériver un schéma de tables à partir du schéma conceptuel de la figure 14.11.⁵

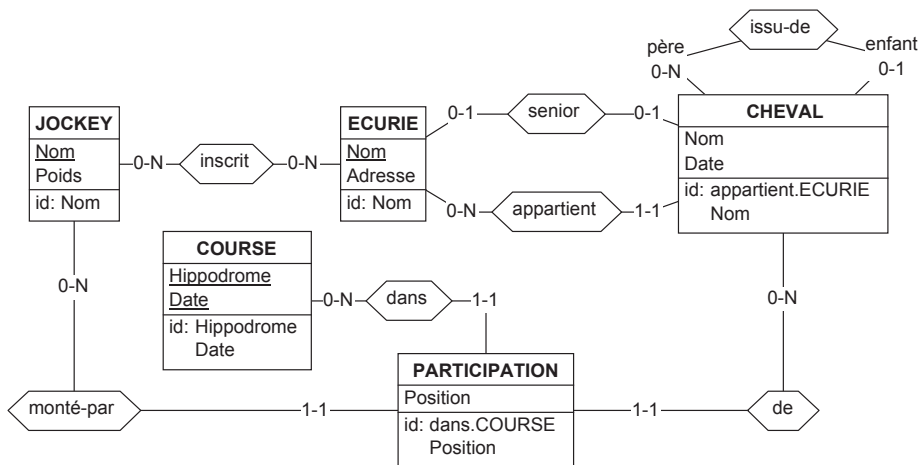


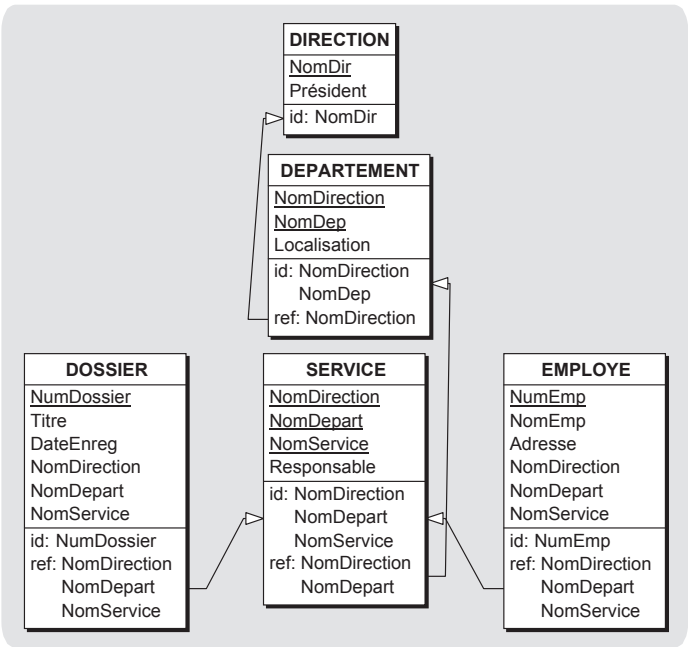
Figure 14.11 - Un schéma conceptuel à traduire

Des exercices supplémentaires accompagnés chacun d'une suggestion de solution sont disponibles à l'annexe A14. Une étude de cas complète est également proposée au chapitre 23.

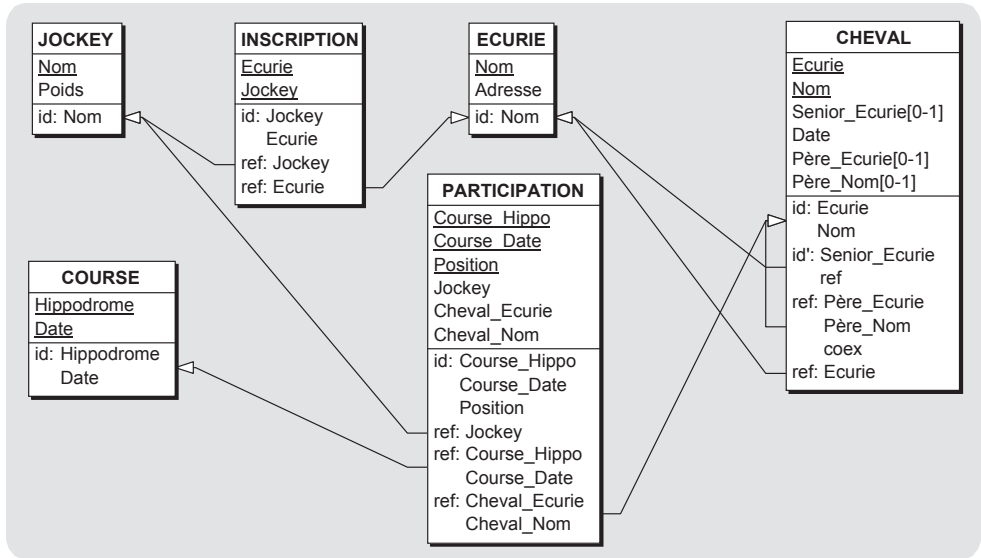
5. On notera que le schéma ne dit rien sur les liens éventuels entre les types d'associations *senior* et *appartient* (le cheval *senior* d'une écurie *appartient-il* à celle-ci ?) Il conviendrait que la traduction demandée ne fasse pas non plus d'hypothèse sur ces liens.

Exercices corrigés

• Exercice 14.1



• Exercice 14.2



Chapitre 15

Méthodologie des bases de données

Ce chapitre présente une méthode de développement de bases de données permettant d'aborder la construction de systèmes complexes et de grande taille. Il redéfinit et étend les processus étudiés précédemment et il introduit de nouveaux processus et de nouveaux produits qui seront détaillés dans les chapitres suivants.

15.1 INTRODUCTION

La méthode décrite au chapitre 11 est parfaitement adaptée au développement de bases de données de petite taille, destinées à un usage personnel ou limitées à une petite communauté. Aborder la construction de bases de données de grande taille, ou fonctionnant dans un environnement exigeant réclame des outils plus professionnels que ceux que nous connaissons, tant en ce qui concerne la modélisation des données que le processus de développement.

Dans ce chapitre, nous ferons référence au concept de *système d'information*. Ce terme a reçu au fil des années plusieurs acceptions qui vont du sous-système socio-technique de l'entreprise au simple synonyme de base de données. Nous lui donnerons ici une signification appropriée à l'objectif de l'ouvrage.

Un système d'information (SI) est un système fortement couplé constitué d'équipements (ordinateurs, serveurs, réseaux), de logiciels de base, de logiciels d'application et de données. Le SI a pour objectif de satisfaire les besoins en information d'une organisation dans ses activités de production et de gestion. Il assure principalement les tâches de collecte, de transmission, de présentation, de mémorisation et de traitement des données relatives à l'organisation et à son environnement. Les systèmes d'information d'aujourd'hui sont fortement distribués. Il doivent assumer

et soutenir la virtualisation des organisations ainsi que l'interopérabilité avec les systèmes externes avec lesquels ils collaborent.

Il existe une définition plus large du concept, vu comme un sous-système partiellement automatisé de l'organisation qui inclut les ressources humaines, comprenant les utilisateurs, les décideurs, le personnel informatique et les tiers de l'organisation [Bodart, 1994], [Vidal, 2005], [Delmond, 2007]. Pour ce qui concerne cet ouvrage, la perception des systèmes d'information sera limitée aux applications informatiques constituées de *bases de données*, de *programmes* sous toutes leurs formes et d'*interfaces* entre programmes et utilisateurs.

Les chapitres qui suivent développent un ensemble de techniques et de méthodes destinées à résoudre une vaste gamme de problèmes. Il est sans doute confortable pour le lecteur de se voir proposer des procédures toutes faites, analogues à des *recettes de cuisine*, qui conduisent de façon certaine à la solution de tous les problèmes qu'il va rencontrer dans l'élaboration des schémas et du code d'une base de données. Sans nier l'intérêt d'une telle approche, nous voudrions aussi introduire dans ces chapitres des pistes de réflexion méthodologiques qui permettent au lecteur de développer une vue critique des processus étudiés, d'élaborer ses propres techniques et donc d'aborder avec confiance les problèmes nouveaux que cet ouvrage ne traite pas, ou trop superficiellement, et qu'il rencontrera dans l'avenir.

15.2 MÉTHODES, PROCESSUS ET PRODUITS

Toutes les méthodes de développement de composants logiciels sont constituées d'un enchaînement de processus qui, partant de l'expression des besoins des utilisateurs, conduisent à une version exécutable du composant supposé satisfaire ces besoins. Une base de données est un composant logiciel un peu particulier, mais sa construction respecte ce modèle. Le chapitre 11, et plus particulièrement la figure 11.2, décrit une méthode élémentaire constituée de deux processus. Les besoins des utilisateurs sont représentés symboliquement par un document dénommé *énoncé*, et le résultat se présente sous la forme d'un script SQL DDL. Les méthodes appliquées par les développeurs professionnels suivent un canevas similaire, bien que plus détaillé. Avant de décrire ces méthodes, nous dirons quelques mots sur leur structure.

En toute généralité, une méthode est un **processus**¹, lui-même constitué d'un enchaînement de processus plus élémentaires. Ces derniers sont à leur tour décomposés en sous-processus jusqu'à ce qu'ils représentent des actions élémentaires. Vu comme une boîte noire, un processus est un *mécanisme* qui utilise un document et produit un nouveau document ou modifie un document existant. Plus généralement, on dira qu'un processus P transforme un ensemble de produits produits_i en entrée en un autre ensemble de produits produits_j en sortie. La figure 15.1 représente de manière graphique la description externe de P .

1. Un *processus* est en réalité une activation particulière d'une *procédure*. Nous devrions donc plutôt parler de procédure. Pour simplifier, nous ne ferons pas la distinction.

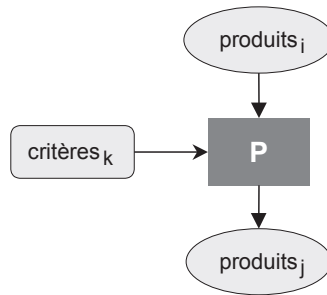


Figure 15.1 - Description externe d'un processus

Il est utile d'expliciter les critères que les produits en sortie doivent satisfaire, de sorte qu'on indiquera également l'ensemble des critères critères_k que P prend en compte. En toute généralité, on peut écrire :

$$\text{produits}_j = P(\text{produits}_i; \text{critères}_k)$$

15.3 MÉTHODES DE DÉVELOPPEMENT D'UNE BASE DE DONNÉES

La figure 15.2 applique cette représentation au processus global de développement d'une base de données.

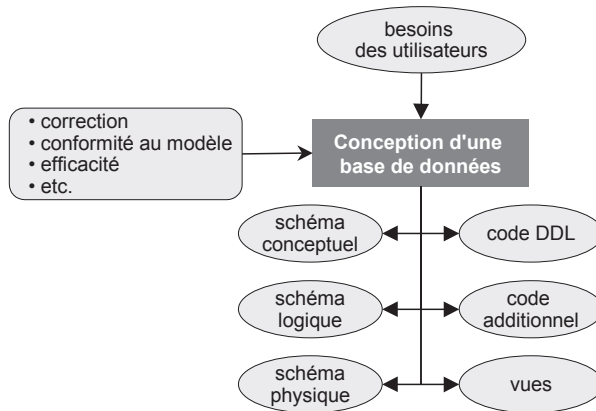


Figure 15.2 - Méthode de développement d'une base de données - Description externe

Elle nous dit que ce processus, dénommé Conception d'une base de données, utilise en entrée un ensemble de produits dénommé besoins des utilisateurs, et produit en sortie une série de documents. Parmi ceux-ci, on relève le schéma conceptuel et le code DDL, déjà mis en évidence à la figure 11.2. On y trouve en outre de nouveaux types de documents, que nous rencontrons pour la première fois : le schéma logique, le schéma physique, le code additionnel et les vues.

Les documents produits par le processus de développement doivent respecter, en tout ou partie, un ensemble de **critères de qualité** : correction, conformité au modèle du SGBD choisi et efficacité. Détaillons ces produits et ces critères.

- **Besoins des utilisateurs.** Ces besoins prennent la forme de documents écrits, imprimés, formulaires, plaquettes, écrans de saisie, bordereaux, règlements, rapports d'entretiens, enregistrements sonores, séquences vidéos, etc. Ces documents apportent chacun des informations sur les objets pertinents du domaine d'application et sur leurs propriétés.
- **Schéma conceptuel.** Ce produit traduit de manière formelle les besoins des utilisateurs sous forme d'un graphe décrivant les concepts statiques pertinents du domaine d'application.
- **Schéma logique.** Ce schéma est la traduction du schéma conceptuel selon le modèle d'une famille de SGBD. Un même schéma conceptuel peut conduire à un schéma relationnel, un schéma relationnel objet ou un schéma XML, pour citer quelques exemples.
- **Schéma physique.** Le schéma physique est un schéma logique particularisé à un SGBD spécifique (Oracle, MySQL, DB2, etc.) Il inclut en outre les constructions techniques susceptibles de doter la future base de données de performances optimales ou simplement satisfaisantes.
- **Code DDL.** Le schéma physique est traduit dans le langage de description de données du SGBD (ici SQL-DDL).
- **Code additionnel.** La traduction en DDL est souvent insuffisante pour exprimer la totalité des structures et des contraintes d'intégrité du schéma logique. Les constructions non traduites de manière explicite feront l'objet d'un codage dans un ou plusieurs langages appropriés. Typiquement, les contraintes additionnelles seront traduites, pour une base de données relationnelle, sous la forme de prédicats, de déclencheurs ou de procédures SQL.
- **Vues.** Un utilisateur ou une catégorie d'utilisateurs (un département ou un service par exemple) n'est concerné que par une partie seulement des données. Chaque vue, parfois appelée également *sous-schéma*, décrit le sous-ensemble des données nécessaire et suffisant pour une catégorie d'utilisateurs.

La figure mentionne également un ensemble de critères :

- **Correction.** Chacun des schémas doit être correct. Il doit représenter toute l'information nécessaire pour satisfaire les besoins des utilisateurs et elle seulement. Cette propriété est évidemment d'application pour le schéma conceptuel. Elle l'est également pour les schémas logique et physique.
- **Conformité au modèle.** Tout schéma doit être constitué exclusivement de constructions conformes au modèle dans lequel il est supposé être exprimé. Cette condition est en particulier d'application pour le schéma logique, qui doit être exprimé dans le modèle du SGBD choisi.

- **Efficacité.** Ce critère s'applique en principe exclusivement au schéma physique. La base de données doit être construite de manière telle que les programmes et les requêtes qui l'exploitent s'exécutent dans un temps minimum, ou tout au moins acceptable. L'efficacité peut également être exigée pour l'espace occupé sur disque ou en mémoire centrale.
- **etc.** D'autres critères de conception seront envisagés, mais leur étude dépasserait la portée de cet ouvrage. Citons par exemple les *coûts* de développement et d'exploitation, la *robustesse* (ou résistance aux pannes), la *sécurité* (résistance aux attaques), la *confidentialité* des données, l'*interopérabilité* avec des bases de données existantes, le respect des *standards méthodologiques* en vigueur, la *portabilité* (facilité de porter la base de données vers un autre SGBD de la même famille), l'*évolutivité* (facilité de modification des structures de la base de données suite à l'évolution du domaine d'application). Ces critères sont souvent désignés sous le terme de *besoins non fonctionnels*².

Jetons à présent un coup d'œil au contenu de la boîte noire. En d'autres termes, nous examinons *comment* le processus réalise ses objectifs.

Ce *comment* s'exprime sous la forme d'un enchaînement de sous-processus. La figure 15.3 représente la méthode la plus standard de développement d'une base de données. On y repère cinq sous-processus et de nouveaux produits internes. On observe également que l'ensemble des critères externes a été redistribué et affiné parmi les sous-processus, chacun de ceux-ci ne prenant en charge qu'une partie de ces critères.

Examinons chacun des cinq processus et les produits qui les concernent. À l'exception de la **Définition des vues**, ces processus ainsi que certains de leurs produits ont déjà été décrits dans une version simplifiée au chapitre 11.

- **Analyse conceptuelle.** Les documents au travers desquels les besoins des utilisateurs s'expriment sont analysés. Lorsque le domaine d'application est très large, il est généralement décomposable en sous-domaines homogènes. Par exemple, une entreprise est formée de services et départements. Dans chacun, il existe un même mode de pensée, une même culture, un même vocabulaire et des objectifs communs. Il est donc naturel de procéder d'abord à l'analyse conceptuelle de chacun des sous-domaines, de manière à obtenir autant de sous-schémas conceptuels. Ces derniers seront ensuite intégrés en un unique schéma conceptuel couvrant la totalité du domaine.

Parallèlement à cette analyse, dite *statique*, l'analyse conceptuelle produira également des documents décrivant les aspects dynamiques et comportementaux du système d'information (désignés collectivement sous le nom schémas

2. Les besoins fonctionnels font référence aux **fonctions** du système d'information liées au domaine d'application (le *métier*). On qualifie de non fonctionnelles des caractéristiques liées aux **modalités de fonctionnement** des composants du système d'information, telles que l'utilisabilité, les performances, la sécurité, la robustesse, la maintenabilité, l'évolutivité ou le coût de développement [Chung, 1999]. Dans la pratique, cette distinction n'est pas toujours aussi claire.

dynamiques) qui, *in fine*, produiront les composants procéduraux (programmes, méthodes ou services selon la technologie employée).

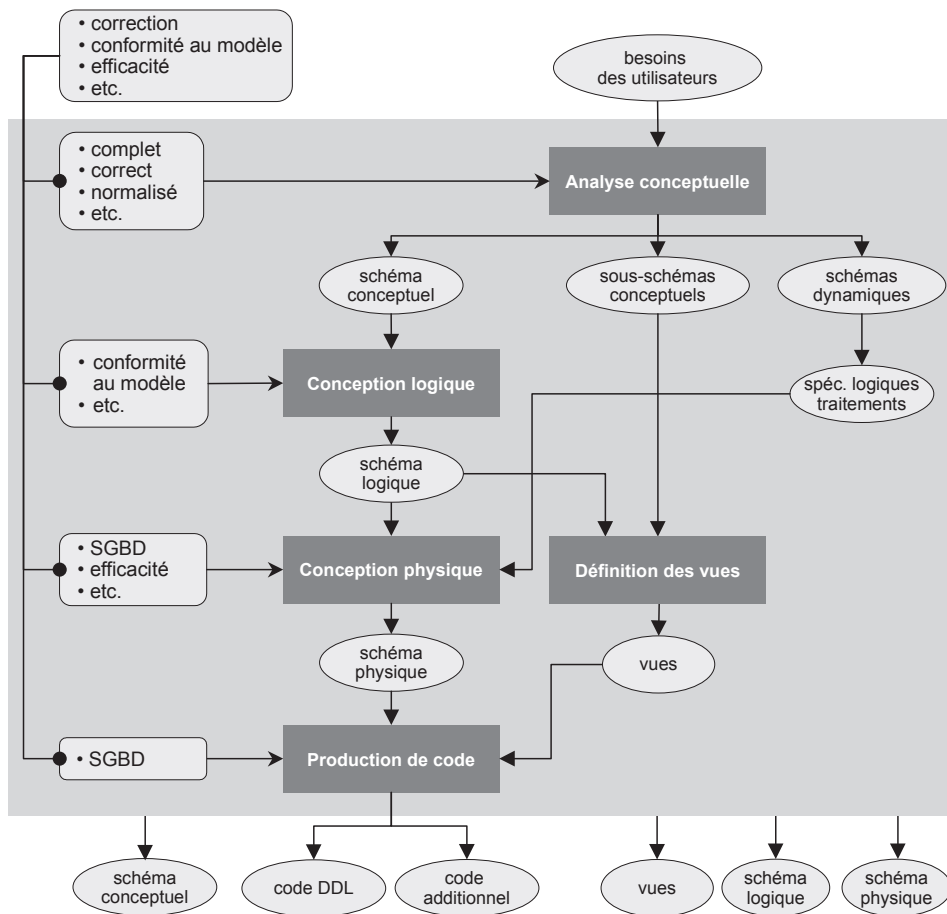


Figure 15.3 - Méthode de développement d'une base de données
- Description interne

Selon UML (chapitre 17), ces documents comprendront notamment les diagrammes d'activité, les diagrammes de séquence, les diagrammes de collaboration et les diagrammes d'état. Ces diagrammes ne présenteront pas pour nous un intérêt immédiat. Cependant, dans la mesure où ils font référence à des données du système d'information, certains diagrammes pourront nous apporter de l'information utile pour construire le schéma physique de la base de données. Le processus doit garantir que les schémas conceptuels sont **corrects**, c'est-à-dire que toute l'information nécessaire, et elle seulement, a été traduite dans les schémas conceptuels. Ceux-ci sont exprimés dans un modèle **Entité-association étendu**, comportant des constructions plus riches et plus expressives que celui que nous avons utilisé jusqu'ici. Ce modèle sera

étudié sous deux variantes aux chapitres 16 (modèle Entité-association proprement dit) et 17 (modèle de classes d'UML). L'analyse conceptuelle elle-même fait l'objet du chapitre 18.

- **Conception logique.** La conception logique a pour objectif de traduire le schéma conceptuel dans le modèle du SGBD choisi pour implémenter la base de données. Le résultat, appelé schéma logique, dépend donc du modèle choisi. Il s'agit d'un processus assez bien défini, qui peut être décrit sous la forme d'un ensemble de règles de transformation, comme nous l'avons vu au chapitre 14. Le processus doit garantir que le schéma logique est **conforme au modèle** du SGBD. Il doit en outre garantir que toute l'information décrite dans le schéma conceptuel, et elle seule, a été traduite correctement dans le schéma logique. Le schéma logique est donc **correct** et **conforme**. Le processus de conception logique sera développé au chapitre 19 pour les bases de données relationnelles et à l'annexe A26 pour les bases de données relationnelles objet.
- **Conception physique.** Le schéma physique est un enrichissement du schéma logique. Celui-ci est complété de constructions techniques qui conféreront à la future base de données de bonnes performances. Conformément aux concepts étudiés au chapitre 4, on assignera les tables à des espaces de stockage, on précisera les modes de stockage, on ajoutera les index et on définira leurs paramètres d'implémentation, pour ne citer que quelques constructions. Pour préciser ces constructions et paramètres, nous avons besoin de connaître certaines exigences de la part des traitements, telles que les requêtes à exécuter et leur fréquence. Ces précisions sont collectivement désignées sous le terme de spec. logiques des traitements. Le processus doit garantir que le schéma physique soit **efficace** pour l'ensemble des applications existantes et à développer. Il doit aussi satisfaire les critères imposés au schéma logique, dont il est une extension. Le schéma physique est donc **correct**, **conforme** et **efficace**. La conception physique est étudiée au chapitre 20.
- **Production de code.** Une partie du schéma physique peut être traduite dans le langage de description des données du SGBD, le plus souvent SQL DDL, sous forme du produit dénommé code DDL. Les constructions non traduites, souvent des contraintes, font l'objet d'une génération de code séparée, le code additionnel, soit sous la forme de composants programmés dans le langage du SGBD (check, triggers, stored procedures), soit dans le corps des programmes d'application. Cette production doit être réalisée avec le plus grand soin, de manière que toute l'information contenue dans le schéma physique (et donc en amont dans les schémas logique et conceptuel) soit traduite correctement. Nous étudierons cette question au chapitre 21.
- **Définition de vues.** Dès que le schéma logique est disponible, il est possible d'en extraire les différentes vues qui traduisent les sous-schémas conceptuels. Dans le cas des SGBD relationnels, chaque vue se traduit en SQL sous la forme d'une collection de *views*. Pour simplifier l'exposé, nous intégrerons l'étude de cette question à la conception logique au chapitre 19.

Nous ajouterons un processus qui s'avère indispensable lorsque la base de données est développée dans un environnement où un système d'information est déjà en activité : la **rétro-ingénierie**. Il s'agit de reconstituer les schémas logique et conceptuel d'une base de données existante. Ce processus sera étudié au chapitre 22.

Ces processus seront décrits dans les chapitres qui suivent selon la répartition de la figure 15.4. On notera que deux chapitres ont été dédoublés de manière à présenter des variantes utiles.

Thèmes	Variantes	Chapitre
Modèle conceptuel	Entité-association	16
	UML	17
Analyse conceptuelle		18
Conception logique	modèle relationnel	19
	modèle relationnel objet	A26
Conception physique		20
Production du code		21
Rétro-ingénierie		22

Figure 15.4 - Processus de la méthode

15.4 LES MODÈLES DE DÉVELOPPEMENT

La méthode que nous venons de décrire obéit à ce qu'on appelle un modèle de développement **en cascade**³.

Ce modèle classique stipule que les activités de développement s'organisent séquentiellement, délivrant des produits qui vont du plus abstrait au plus concret. Dans sa version la plus simple, il ne prévoit pas en principe de retour en arrière ni d'itération. Ce modèle a été très populaire dans les années soixante-dix et quatre-vingts. Il offre l'avantage de décomposer le processus de développement en activités bien identifiées et garantissant la qualité des produits. Chacun de ceux-ci est complété, vérifié et validé avant d'être traité par le processus suivant. Ce modèle est souvent appliqué dans les grands projets faisant appel à de multiples équipes à coordonner et produisant des systèmes stables à moyen terme. En revanche, on reproche au modèle en cascade sa rigidité (les besoins ne peuvent être modifiés facilement), son opacité pour les utilisateurs (qui sont exclus de la plupart des processus) et les trop grands délais de livraison (plusieurs mois à plusieurs années). À ce modèle classique, on oppose souvent les **modèles itératifs**, dont le modèle en *spirale* et les méthodes plus récentes dites *agiles*.

3. En anglais *waterfall*.

Les méthodes itératives sont faites d'itérations courtes, chacune parcourant la totalité des processus des méthodes en cascade. L'objectif d'une itération n'est pas de produire le système d'information complet, mais plutôt de livrer aux utilisateurs une première version partielle mais opérationnelle. En fonction des réactions des utilisateurs et des besoins qui restent à satisfaire, d'autres itérations sont effectuées, produisant des versions successives du système d'information. Ce modèle offre des avantages importants, parmi lesquels nous citerons les suivants :

- Le développement peut commencer alors que la totalité des besoins n'ont pas encore été identifiés.
- Les utilisateurs disposent rapidement d'une première version utilisable.
- Les utilisateurs peuvent évaluer et critiquer le système opérationnel très tôt dans le processus global ; ils jouent un rôle actif dans le processus de développement, ce qui augmente la qualité du produit final et favorise son acceptation par les utilisateurs.
- Les erreurs sont détectées très tôt dans le processus et peuvent être corrigées rapidement et à faible coût.
- Le contexte économique actuel fait que les besoins sont en évolution rapide et sont donc imprévisibles à moyen terme. Il devient de plus en plus difficile d'établir une liste exhaustive et stable de ces besoins. Les méthodes itératives permettent de suivre de près cette évolution et de la traduire dans le système en développement.
- Le développement en cascade s'oppose au principe de plaisir auquel chaque personne raisonnablement équilibrée (et donc la plupart des informaticiens et des utilisateurs) cherche à adhérer, et selon lequel un effort est d'autant plus valorisé que son résultat est rapidement disponible. L'immédiateté du résultat par rapport à l'effort est une exigence qu'on reproche souvent aux jeunes générations. L'avantage des méthodes itératives est qu'elles réussissent à exploiter un défaut de société au profit de la qualité du produit final, et ce, en tout bien tout honneur. Cette conjonction est suffisamment rare pour être soulignée !

Cette description désigne clairement les modèles itératifs comme supérieurs au modèle en cascade. Le lecteur pourrait dès lors s'étonner que la suite de cet ouvrage développe le modèle en cascade décrit à la figure 15.3. Le paradoxe n'est qu'apparent. En effet, d'une part, chaque itération est un parcours complet d'un processus en cascade, qui part des besoins pour aboutir au code, appliqué à une partie seulement des besoins et du système final. D'autre part, le modèle en cascade peut être vu, non pas comme une manière impérative de conduire le processus de développement, mais comme un **modèle de référence** à la fois méthodologique et pédagogique, qui permet d'identifier les critères de conception, de classer de manière homogène les problèmes à résoudre, d'associer à ceux-ci des techniques, des raisonnements et des outils appropriés. C'est donc dans ce sens qu'il faut lire, comprendre et appliquer les chapitres qui suivent.

15.5 TECHNIQUES TRANSFORMATIONNELLES

Les méthodes que nous allons développer s'appuient fortement sur des *techniques transformationnelles*. Dans les chapitres qui suivent, nous les appliquerons de manière intuitive, en fonction des nécessités du moment. Le lecteur intéressé trouvera plus de détails sur ces techniques dans [Hainaut, 2006] et à l'annexe A25.

15.6 RÔLE DES OUTILS

Si l'utilisation d'outils d'aide à la conception de bases de données reste facultative pour des projets de petite taille, il n'en va pas de même pour les projets de taille plus importante. Un schéma conceptuel de taille moyenne compte plusieurs centaines de types d'entités, au moins autant de types d'associations et plusieurs milliers d'attributs. Le code SQL qui lui correspond occupera, tables, contraintes et index compris, plusieurs centaines et même plusieurs milliers de pages qu'il est illusoire de tenter de rédiger à la main, ne fut-ce que pour des raisons de fiabilité. L'évolution synchronisée de ces schémas suite à l'évolution du domaine d'application est un problème particulièrement complexe qui nécessite également le support d'un outil.

En toute généralité, on demandera à un atelier de génie logiciel (AGL) *orienté données* d'assurer, entre autres, les fonctions suivantes, dont nous rencontrerons de multiples applications dans les chapitres qui suivent :

- saisie, modification et documentation de schémas, sous forme graphique ou textuelle ;
- gestion et mémorisation de schémas et de leurs inter-relations ;
- visualisation, navigation et interrogation de schémas, sous forme graphique ou textuelle; marquage et coloriage de constructions ;
- analyse de schémas, recherche de patterns ;
- transformation de schémas selon différentes granularités, traduction d'un schéma d'un modèle dans un autre ;
- aide à la normalisation et à la correction de schémas ;
- aide à l'intégration de schémas ;
- génération de code à partir d'un schéma ;
- extraction de schémas à partir de code ;
- d'interactions et échanges avec d'autres AGL.

Dans le cadre de cet ouvrage, nous continuerons à utiliser l'outil DB-MAIN, dans des fonctions plus avancées liées aux modèles et aux processus que nous allons décrire dans la suite. L'environnement SQLfast sera aussi mis à contribution, notamment dans la génération de code et le développement d'outils de support.

Chapitre 16

Le modèle Entité-association étendu

La modélisation de domaines complexes réclame des concepts plus expressifs et plus riches que ceux qui ont été utilisés jusqu'ici. Ce chapitre complète le chapitre 12 en décrivant certains de ces concepts, notamment sous forme d'extensions de leur version d'origine. Il introduit notamment les attributs composés, les attributs multivalués, les types d'associations complexes (n-aires ou dotés d'attributs), les hiérarchies de généralisation (relations *is-a*), de nouvelles contraintes d'intégrité ainsi que les objets génériques. Ce chapitre introduit la notion de sémantique du modèle Entité-association.

16.1 INTRODUCTION

Le modèle de base présenté au chapitre 12 n'est qu'un sous-ensemble des modèles utilisés par les informaticiens pour concevoir et développer les systèmes d'information, et plus particulièrement leurs bases de données. Nous étudierons dans ce chapitre des extensions qui complètent ce modèle de base.

Pourquoi ces extensions ? Un vocabulaire limité permet sans doute de se tirer, maladroitement, de la plupart des situations, mais un vocabulaire riche, plus expressif, conduit à une meilleure compréhension et permet d'exprimer de manière plus simple, plus claire et plus précise des concepts complexes. Il en est de même du modèle Entité-association, qui est un langage de description de concepts. Ignorer ces extensions nous conduira à faire des choix techniques prématurés qui risquent de dégrader la qualité des schémas conceptuels.

Ni les concepts, ni leur représentation graphique ne sont à proprement parler standardisés, à l'exception de la variante des diagrammes de classes d'UML à laquelle

nous consacrerons le chapitre 17. Nous adopterons ici des définitions et conventions graphiques proches des différents modèles utilisés en pratique mais offrant parfois plus de précision afin de satisfaire des besoins de modélisation très diversifiés.

16.2 LE MODÈLE DE BASE - RAPPEL

La lecture de ce chapitre suppose acquis les concepts du modèle de base, qu'on rappelle par précaution, à partir du schéma de la figure 16.1.

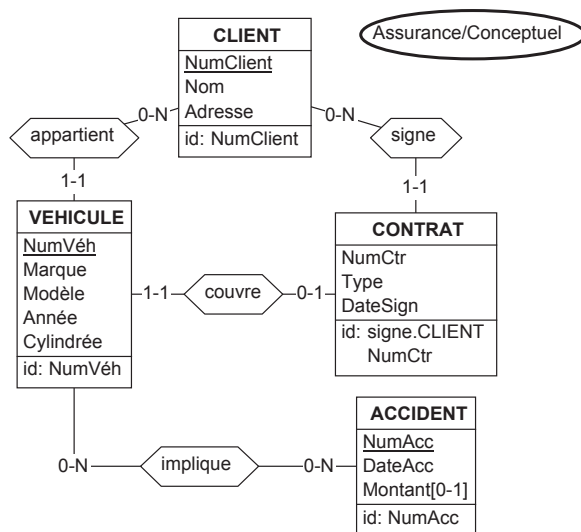


Figure 16.1 - Un schéma conceptuel exprimé dans le modèle de base

Un schéma conceptuel exprimé dans le modèle de base est composé de types d'entités (CLIENT, VEHICULE, CONTRAT, ACCIDENT), de types d'associations binaires (appartient, signe, couvre, implique), d'attributs de type d'entités (NumClient, Nom, Adresse, etc.) et d'identifiants ({NumClient}, {signe.CLIENT, NumCtr}). Ses valeurs appartiennent à un type de base (nombres, chaînes de caractères, booléens, dates, etc.). Les attributs sont atomiques et monovalués. Un type d'associations est composé de deux rôles, définis chacun sur un type d'entités. Un rôle est soumis à une contrainte de cardinalité [i-j] qui précise les valeurs minimum (i) et maximum (j) du nombre d'associations auxquelles une entité participe. Les différentes combinaisons de contraintes de cardinalité définissent les classes fonctionnelles *un-à-plusieurs* (1:N), *plusieurs-à-un*¹ (N:1), *un-à-un* (1:1) et *plusieurs-à-plusieurs* (N:N). Chaque type d'entités possède un ou plusieurs identifiants. Un schéma conceptuel exprimé dans le modèle de base obéit aux règles suivantes :

1. Si on admet qu'un type d'associations *se lit* dans un certain sens : de A vers B ou de B vers A ; alors un type d'associations est *N:1* selon un sens de lecture s'il est *1:N* dans l'autre sens. La question ne se pose évidemment pas pour les classes fonctionnelles *1:1* et *N:N*.

1. une entité appartient à un et un seul type ;
2. un attribut est atomique et monovalué; il est attaché à un type d'entités ;
3. un type d'associations est binaire et ne possède pas d'attributs ; chaque rôle est joué par un type d'entités ;
4. les contraintes de cardinalité admises sont les suivantes : [0-1], [1-1] et [0-N] ; les combinaisons de contraintes de cardinalités autorisées pour les rôles d'un type d'associations sont [1-1, 0-N], [0-1, 0-N], [1-1, 0-1], [0-1, 0-1] et [0-N, 0-N] ;
5. un type d'entités possède un ou plusieurs identifiants ;
6. un identifiant est constitué d'un groupe d'attributs, d'un groupe de rôles opposés ou d'une combinaison d'attributs et de rôles opposés.

Ces règles font du modèle de base un mode d'expression de schémas simple, intuitif et aisé à traduire en structures de tables.

16.3 SÉMANTIQUES

La notion de *sémantique* est omniprésente dans le domaine des bases de données mais n'est pas toujours aisée à définir de façon univoque. Nous distinguerons la sémantique externe et la sémantique interne.

Sémantique externe d'un schéma

La *sémantique externe d'un schéma* est l'ensemble des correspondances entre les objets de ce schéma et les concepts du domaine d'application. Elle précise le plus rigoureusement possible et sans ambiguïté ce que représente dans le monde réel une entité du type E, une valeur de l'attribut A et une association du type R. La sémantique (externe) d'un schéma s'exprime généralement en langage naturel *discipliné*. Elle constitue la documentation du schéma conceptuel de la base de données. Elle pourrait se présenter sous la forme d'un ensemble de propositions telles que les suivantes, empruntées à la section 13.10 :

- 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...

On peut également parler de la *sémantique externe d'un modèle* dans la mesure où il est possible d'indiquer quelle famille de concepts de tout domaine d'application on représente habituellement par chaque objet du modèle. On dira par exemple qu'*une entité représente un objet concret ou abstrait, animé ou inanimé, au sujet duquel on désire enregistrer de l'information*, ajoutant qu'*un type d'entités représente un ensemble d'entités similaires* (voir 12.2). Ces définitions restent cependant trop générales pour présenter un réel intérêt.

Sémantique interne d'un modèle

La *sémantique interne d'un modèle* est l'ensemble des correspondances entre les objets du modèle et une théorie mathématique communément acceptée. On a parfois reproché aux modèles conceptuels leur manque de rigueur, leur caractère empirique et leur définition intuitive. On a parlé à leur égard de *boxologie*². Cette critique est importante car l'absence de définition rigoureuse conduit à l'ambiguïté, aux conflits d'interprétation, à l'impossibilité de raisonner rigoureusement et à la difficulté de construire des outils basés sur ce modèle³. Les deux propositions suivantes, inspirées de la section 16.8, associent une sémantique relationnelle aux concepts de *type d'entités* et de *type d'associations* :

- un type d'entités E est représenté par un **ensemble** d'entités de nom E ;
- un type d'associations R défini sur les types d'entités E1, E2, ..., En est représenté par une **relation** de nom R définie sur les ensembles E1, E2, ..., En.

Ces deux définitions, bien que très partielles, permettent de comprendre certaines propriétés des types d'entités et des types d'associations. La notion de sémantique interne permet de clarifier le rôle de la notation associée à un modèle. Un modèle peut utiliser plusieurs notations, dont certaines sont de nature graphique. La notation graphique utilisée dans cet ouvrage n'est pas unique. Un même schéma peut s'exprimer (*se dessiner*) selon des notations différentes tout en signifiant la même chose. Une notation est définie par une *syntaxe* qui définit les combinaisons graphiques et textuelles valides. Le lecteur intéressé par les rapports entre syntaxe et sémantique lira avec intérêt [Harel, 2004].

La *sémantique interne d'un schéma* est la transposition de la sémantique interne de son modèle à une instance particulière de celui-ci.

La section 16.17 de ce chapitre est consacrée à une sémantique relationnelle du modèle Entité-association utilisant les concepts présentés au chapitre 3.

16.4 MODÈLE, SCHÉMA ET POPULATION

Il règne actuellement une grande confusion autour du terme de *modèle*, due à la rencontre de communautés qui ont défini leurs propres concepts de manière indépendante. Afin de tenter de clarifier les choses, nous aurons besoin de repréciser ce qu'on entend par le terme *instance*. Considérant un objet C représentant une catégorie de *choses*, chacune de ces *choses* est une **instance** de C. Si C représente le type de données entier, alors 14 dénote une instance de C. De même, un *objet* (au sens

2. Boxology: 1) a hierarchical presentation of software architecture using boxes and arrows, 2) pejoratively, reduction of a scientific hypothesis to a series of boxes connected by arrows in lieu of a definitive exposition of the hypothesis referencing careful studies in <http://home.comcast.net/~wwftd/abc.htm#boxology>. Rien à voir donc avec la science des lieux peu recommandables.

3. Les notations UML sont réputées avoir une sémantique mal définie. Les outils basés sur celles-ci implémentent une sémantique qui est celle que leur développeur a choisie. Deux outils différents proposent donc des modèles différents dès lors que leurs développeurs ont fait des hypothèses différentes sur la signification des notations UML.

orienté objet) est une instance d'une *classe*, une *entité* est une instance d'un *type d'entités* et une *association* est une instance d'un *type d'associations*. L'ensemble des instances d'un objet à un moment déterminé constitue la **population** de cet objet. On notera $\text{pop}(E)$ la population actuelle de l'objet E , ou, lorsqu'il n'y a pas de risque d'ambiguïté, simplement E . Par exemple, on écrira, pour les types d'entités PERSONNE et FEMME :

$\text{pop}(\text{FEMME}) \subseteq \text{pop}(\text{PERSONNE})$ ou plus simplement $\text{FEMME} \subseteq \text{PERSONNE}$.

Nous examinerons comment le concept de modèle est interprété dans trois communautés différentes mais fortement inter-indépendantes, souvent à leur corps défendant, celle des bases de données, celle de la programmation orientée objet et celle de la logique.

- Dans le domaine des *bases de données*, un **modèle** est un système formel général destiné à représenter certains aspects de tous les domaines d'application réels et possibles d'un certain type. Un modèle est généralement constitué d'objets, de règles qui définissent les assemblages valides d'instances de ces objets et d'une sémantique précisant l'interprétation interne à donner aux instances de ces objets. Par exemple, le *modèle Entité-association* permet de représenter les concepts statiques constituant ces domaines d'application. Il comporte, entre autres, les objets *type d'entités*, *attribut*, *type d'associations* et *identifiant*. Il existe une règle qui indique qu'un type d'associations est composé d'au moins deux rôles, chacun étant joué par un ou plusieurs types d'entités. Un **schéma** représente les concepts statiques d'un domaine d'application particulier, tel que la compagnie d'assurance ASSURTOUT ou la bibliothèque municipale. Un schéma est exprimé dans un modèle. Il est constitué d'objets qui sont autant d'*instances* d'objets de ce modèle. Par exemple, le schéma de la figure 16.1 est formé de quatre instances de l'objet *type d'entités*, nommées CLIENT, VEHICULE, CONTRAT et ACCIDENT. À chaque objet d'un schéma on peut associer l'ensemble de ses instances, qui constitue sa **population**. La population de CLIENT du schéma cité précédemment est formée de toutes les entités CLIENT observables à un instant donné dans le domaine d'application.

De même, le *modèle relationnel* (chapitre 3) comprend notamment les objets *relation*, *domaine*, *attribut*, *identifiant* (ou *clé candidate*), *clé étrangère* et *dépendance fonctionnelle*. Quant au *modèle SQL2* qui en dérive (chapitre 2), il inclut les objets *table*, *colonne*, *type de données*, *identifiant primaire (clé primaire)*, *clé étrangère*, *prédicat*, *trigger*, etc.

- Dans le domaine de la *programmation et des architectures orientées objet*, un **modèle** est une représentation, à l'aide d'une notation appropriée, de certains aspects d'un domaine d'application particulier, ce que nous appelons dans cet ouvrage un schéma pour ce qui concerne les concepts statiques du domaine d'application. Cette deuxième acception est celle d'UML par exemple. Ce que nous appelons *schéma conceptuel de la compagnie d'assurance ASSURTOUT* s'appellera en UML le *modèle de classes* de cette compagnie. Son expression graphique est appelée *diagramme*.

- Pour compliquer les choses, la *logique* assigne au terme de **modèle** une troisième signification. Étant donné un ensemble de prédicats, un modèle est un ensemble de valeurs qui satisfont ces prédicats⁴.

Très schématiquement, considérant dans le domaine des bases de données un modèle M, un schéma S exprimé dans M et une population D de S (figure 16.2),

1. M est un *modèle* dans le domaine des bases de données ;
2. S est un *modèle* au sens d’UML ;
3. D est un *modèle* au sens de la logique.

Bases de données	UML	Logique
modèle	<i>notation, diagrammes</i>	logique du 1 ^{er} ordre
schéma	modèle	ensemble de prédicats
données	instances	modèle

Figure 16.2 - Pérégrination du concept de *modèle*. Les termes d’une même ligne sont plus ou moins équivalents

Quand on sait que les bases de données, les diagrammes de classes UML et la logique ont de nombreuses inter-relations, on comprend l’importance du choix des termes appropriés à la culture de l’interlocuteur.

16.5 LES STÉRÉOTYPES

Un *stéréotype* est un nom, choisi dans une liste déterminée, qu’on peut attacher à un ou plusieurs objets d’un schéma. Cette construction est empruntée aux diagrammes de classes d’UML, où elle constitue un moyen facile d’étendre la puissance du modèle. Chaque stéréotype permet de définir dans un schéma une *classe d’objets*. Les stéréotypes ne sont généralement pas liés à un domaine d’application particulier. Ils n’ont a priori aucune signification prédéfinie. C’est au concepteur de leur attribuer une sémantique et des règles qu’il estime utiles.

Un stéréotype apparaît entre doubles crochets à proximité du nom de l’objet qu’il caractérise. La figure 16.3 illustre le concept : le type d’entités PRODUIT est qualifié de «signalétique» tandis que l’attribut Montant et le type d’associations achète sont qualifiés de «dérivé». On peut imaginer que le stéréotype «signalétique» désigne les types d’entités décrivant les objets fondamentaux de l’entreprise (le personnel, les clients, les fournisseurs, les produits, les postes budgétaires) et que le stéréotype «dérivé» est utilisé pour qualifier les objets dont les instances sont dérivables et donc redondantes. Dans les extensions du modèle Entité-associations décrites dans ce chapitre, nous utiliserons à plusieurs reprises des stéréotypes prédéfinis pour exprimer des concepts que le modèle standard n’offre pas.

4. Cette interprétation ne semblera sans doute pas tout à fait orthodoxe à certains logiciens. Elle se base sur la dualité algèbre/calcul (et en particulier *relation/prédicat* et *n-uplet/proposition*) à la base de l’interprétation mathématique du modèle relationnel.

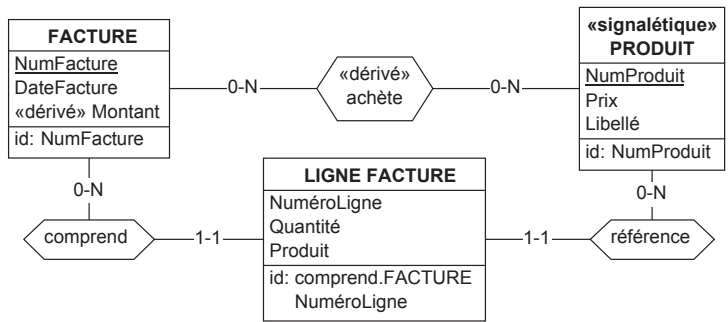


Figure 16.3 - Exemples de stéréotypes

16.6 SCHÉMA DE BASE DE DONNÉES

Un schéma conceptuel est une description d’une partie d’un domaine d’application. Il peut exister plusieurs schémas relatifs à ce même domaine, représentant chacun un aspect ou une partie spécifique à un moment déterminé. Les schémas relatifs à un même domaine d’application portent des noms distincts. Il est courant d’associer à ce nom l’indication de la version. On représentera un schéma par une ellipse contenant le nom et la version de celui-ci (figure 16.1).

16.7 ATTRIBUTS

Dans le modèle de base, les attributs sont définis sur des domaines élémentaires (entiers, chaînes de caractères, booléens, etc.), c’est-à-dire monovalués (une seule valeur) et atomiques (une valeur est indécomposable). Nous avons cependant rencontré des difficultés de modélisation dues à cet excès de simplicité. À la section 13.5, nous aurions voulu par exemple attacher un attribut à un attribut préexistant ou encore représenter une propriété multiple. Le modèle étendu comble ces lacunes en offrant les concepts d’attributs composés et d’attributs multivalués.

16.7.1 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 élémentaires Rue et Ville (figure 16.4). Une valeur de l’attribut Adresse pourrait être : ('rue du Tertre', 'Poitiers'). Le **niveau** d’un attribut est sa distance dans l’arbre des attributs par rapport à son type d’entités ou d’associations. Dans la figure 16.4, Contact est un attribut de niveau 1 de CLIENT tandis que Ville est de niveau 3.

16.7.2 Attributs multivalués

Jusqu’ici, on a considéré que chaque entité d’un type possède au plus **une** valeur de chacun de ses attributs. Rien n’interdit qu’une entité possède plusieurs valeurs

d'un attribut, qu'on caractérise dès lors de *multivalué*. Tel serait le cas de l'attribut MotClé de DOCUMENT et Téléphone d'EMPRUNTEUR (figure 16.4). Par défaut, une valeur d'un attribut multivalué est un *ensemble* d'éléments. On admet aussi d'autres types de collections de valeurs, tels que les *multi-ensembles*, les *listes* et les *tableaux*. Ces structures étant surtout pertinentes au niveau des schémas logiques⁵, elles seront traitées au chapitre 19 et à l'annexe A26.

EMPRUNTEUR	DOCUMENT	CLIENT
NumPers	ID-Doc	NCli
Adresse	MotClé[0-10]	Contact[1-5]
Rue	id: ID-Doc	Statut
Ville		Adresse
Téléphone[1-5]		Rue
id: NumPers		Ville
		Téléphone[0-5]
		CodePays[0-1]
		CodeZone
		NuméroLocal
		id: NCli

Figure 16.4 - Attributs composés et multivalués

16.7.3 Attributs complexes

Un attribut peut être à la fois composé et multivalué. S'il est composé, chacun de ses composants peut, à son tour, être composé et/ou multivalué. L'attribut Contact du type d'entités CLIENT du schéma de la figure 16.4 en est une illustration. Attention cependant, un attribut de structure trop complexe peut être l'indice d'une mauvaise modélisation (voir exercice 16.4). Le tableau de la figure 16.5 explicite les principales caractéristiques des attributs du type d'entités CLIENT de la figure 16.4.

16.7.4 Domaine et type d'un attribut

Le *domaine de valeurs* d'un attribut est l'ensemble dont il tire ses valeurs. Ce domaine peut être générique et défini par un *type de base* indépendants du domaine d'application tels que nombres, chaînes de caractères ou dates. Dans d'autres cas, on définira le domaine de valeurs en précisant les valeurs permises, soit de manière exhaustive, soit en imposant des restrictions sur les valeurs d'un type. Ces restrictions constituent des *contraintes de domaines*, qui seront étudiées à la section 16.11.1). On parlera alors de *types définis par l'utilisateur*.

a) Types de base

Il s'agit des types de données les plus courants, tels qu'on les trouve dans les SGBD. Ils font partie intégrante du modèle et comprennent typiquement les nombres

5. En réalité, ces autres types de collections pourraient également être pertinents dans un schéma conceptuel. Par exemple, une personne ne possède pas un ensemble de prénoms, mais bien une liste de prénoms distincts (ce qu'on appellera unique list), le concept de premier, deuxième, ... prénom étant pertinent dans le domaine d'application.

entiers, les nombres réels, les chaînes de caractères, les dates, les instants, les booléens, les images, les sons, etc. L'utilisateur peut parfois préciser certaines caractéristiques : longueur fixe ou variable, longueur maximum, type de codage, etc.

Attributs	atomique	composé	composant	monoval.	multival.	obligat.	facult.
NCLI	X			X		X	
Contact		X			X	X	
Contact.Statut	X		X	X		X	
Contact.Adresse		X	X	X		X	
Contact.Adresse .Rue	X		X	X		X	
Contact.Adresse .Ville	X		X	X		X	
Contact.Téléphone		X	X		X		X
Contact.Téléphone .CodePays	X		X	X			X
Contact.Téléphone .CodeZone	X		X	X		X	
Contact.Téléphone .NuméroLocal	X		X	X		X	

Figure 16.5 - Classification des attributs du type d'entités CLIENT

b) Types définis par l'utilisateur (TDU)

Les types définis par l'utilisateur sont spécifiques au domaine d'application. Tel est le rôle des *types définis par l'utilisateur* (TDU), concept analogue au *domaine* de SQL2 et au *TDU* de SQL3. La figure 16.6 décrit une collection de cinq types de valeurs spécifiques à un domaine d'application ainsi que le type d'entités CLIENT dont certains attributs sont définis sur ces types. Cet exemple montre comment l'usage de ces types clarifie la définition des types d'entités tout en la simplifiant.

CLIENT	«TDU» Gestion CLIENTS
Code client: matricule Nom: char(32) Adresse expédition: adresse Adresse facturation: adresse Contact[0-3]: téléphone Compte Identité: numéro_de_compte Niveau: montant Fiabilité: boolean	matricule code_région: char(2) numéro_série: num(6) adresse numéro: char(8) rue: char(26) ville: char(32) téléphone_principal: téléphone téléphone code_pays: num(3)[0-1] code_zone: num(4) code_local: num(8) numéro_de_compte code_banque: num(4) code_compte: num(10) check_sum: num(2) montant: num(10,2)

Figure 16.6 - Utilisation de types de valeurs définis par l'utilisateur

16.7.5 Cardinalité d'un attribut

Un attribut est directement attaché à un type d'entités, à un type d'associations (voir ci-après) ou à un attribut composé. Chacun de ces trois derniers objets est appelé le **parent** de l'attribut. Dans le schéma 16.4, CLIENT est le parent de l'attribut Contact tandis que Contact est le parent de l'attribut Téléphone.

La contrainte de cardinalité d'un attribut, plus simplement appelée sa *cardinalité*, précise par un intervalle [min-max] les nombres minimum et maximum de valeurs attachées à chaque instance du parent. Cette contrainte est indiquée explicitement dans le schéma sauf lorsqu'elle vaut [1-1], auquel cas elle est omise.

Soit $[i-j]$ la cardinalité de l'attribut A. Cette contrainte exprime deux propriétés indépendantes : le caractère obligatoire ou facultatif de l'attribut d'une part et sa multiplicité d'autre part. L'attribut A sera dit :

- **obligatoire** si $i > 0$ ou **facultatif** si $i = 0$
- **monovalué** si $j = 1$ ou **multivalué** si $j > 1$

16.8 TYPES D'ASSOCIATIONS

Selon le modèle de base, un type d'associations binaire (sans attribut) traduit l'existence de couples d'entités dans le domaine d'application : les clients et leurs commandes, les véhicules et leurs accidents, les ouvrages et leurs exemplaires. Les limitations de ces structures élémentaires nous ont déjà posé des problèmes de modélisation : comment représenter des triplets d'entités (section 13.3.4) ou encore comment attacher un attribut à un type d'associations (section 13.5.5) ? Le modèle étendu permet de représenter des situations complexes de manière simple et directe.

16.8.1 Types d'associations fonctionnels

On qualifie de *fonctionnel* un type d'associations binaire, sans attributs et qui est le siège d'au moins une fonction. On désigne ainsi les types d'associations dont au moins un rôle est de cardinalité $i-1$, soit les classes fonctionnelles 1:1, N:1 et 1:N.

16.8.2 Types d'associations *n-aires*

Un type d'associations binaire est formé de deux rôles et sa population est constituée d'un ensemble de couples d'entités. On peut généraliser ce nombre à plus de deux. On peut ainsi décrire explicitement le fait que « les *clients* achètent des *produits* chez des *fournisseurs* » ou que « les *emprunteurs* empruntent des *exemplaires* pour le compte de *projets* » (figure 16.7). On qualifie de *n-aires*, où n est le nombre de rôles, de tels types d'associations. Statistiquement, les types d'associations binaires sont les plus fréquents, suivis des types d'associations ternaires (degré 3). Les types d'associations de degré supérieur sont plus rares.

Lorsqu'on sera tenté de les définir, on s'assurera qu'ils ne sont pas décomposables (voir section 18.7).



Figure 16.7 - Deux types d'associations ternaires ou de degré 3

Un type d'associations est dit *cyclique* lorsqu'au moins deux de ses rôles sont définis sur le même type d'entités. Le type d'associations vendu de la figure 16.8 indique qu'un *immeuble* a été vendu par un *client-vendeur* à un *client-acheteur*. Le schéma de droite de la même figure représente des transferts d'un produit d'un dépôt vers un autre par un transporteur⁶.

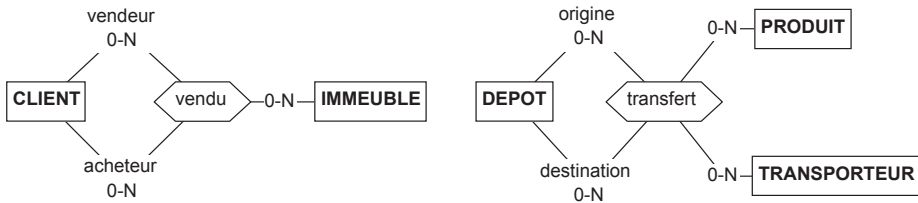


Figure 16.8 - Types d'associations cycliques n-aires

La typologie en classes fonctionnelles {*un-à-plusieurs* ou 1:N, *un-à-un* ou 1:1, *plusieurs-à-plusieurs* ou N:N}⁷ des types d'associations binaires est généralisable aux degrés supérieurs : on parlera par exemple d'un type d'associations ternaire *plusieurs-à-plusieurs-à-plusieurs*, ou, de manière plus concise N:N:N (figure 16.29).

16.8.3 Types d'associations à rôles polymorphiques

En principe, un rôle est joué par **un** type d'entités. Certaines situations suggèrent que les entités jouant ce rôle puissent être tirées d'un type ou d'un autre⁸. La figure 16.9 illustre un de ces cas. Elle indique que tout véhicule a un propriétaire et que celui-ci est soit une personne, soit un service. Un tel rôle est qualifié de *polymorphique* ou *multitypes d'entités*. Malgré la présence de ce rôle polymorphique, possède est bien

6. On trouvera dans [Dullea, 1999] une étude approfondie des types d'associations cycliques.

7. Qu'on se gardera bien de confondre avec les *cardinalités des rôles*, dont l'expression reprend certaines de ces formes.

8. Bien que rarement reprise dans les modèles Entité-association, cette construction est très ancienne. Elle apparaît en effet dans le modèle Data Structure Diagrams [Bachman, 1969], un ancêtre des modèles Entité-association sous la forme de *multi-member set class* et *alternate owner set class*. La fonctionnalité *multi-member* a été reprise par les modèles dits *en réseau* (ou CODASYL) à la base de SGBD toujours en activité [Hainaut, 2009]. Ce concept existe également dans les ORM (convertisseurs relationnel/objet) tels que *Ruby on Rails* et *Hybernate* sous le nom de *polymorphic association*.

un type d'associations **binaire**, dont la population est un **ensemble de couples**.

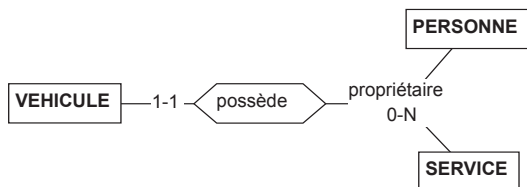


Figure 16.9 - Un rôle polymorphe

16.8.4 Attributs de type d'associations

Un type d'associations peut recevoir des attributs. Par exemple, le type d'associations **réserve** (figure 16.10), qui indique qu'un document est réservé par un emprunteur, est caractérisé par l'attribut **Date Réserve**, qui spécifie la date à laquelle cette réservation a été enregistrée. Cet attribut n'est spécifique ni aux emprunteurs seuls ni aux documents seuls, mais aux couples du type d'associations qui les unit.

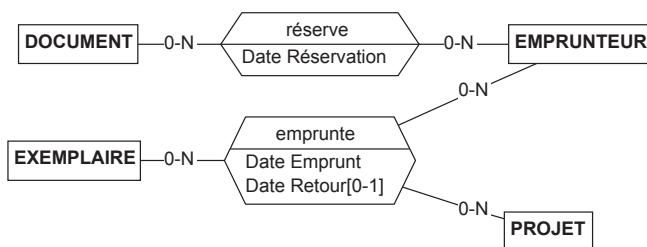


Figure 16.10 - Deux types d'associations munis d'attributs

De même, il est intéressant d'attacher au type d'associations **emprunte** (même figure) des attributs qui renseignent sur les dates d'emprunt et de retour des exemplaires, cette dernière date n'étant pas connue durant la période d'emprunt.

16.8.5 Généralisation des cardinalités des rôles

Les contraintes de cardinalité ne sont pas limitées aux trois combinaisons [1-1], [0-1] et [0-N] proposées dans le modèle de base. Tout couple $[i-j]$ est autorisé pour autant que $0 \leq i, 1 \leq j \leq N$ et $i \leq j$. Seul j peut prendre la valeur N , qui représente un entier arbitrairement grand, soit l'*infini*. Le schéma de la figure 16.11 montre qu'une commande doit spécifier au moins un produit (chez un fournisseur), mais ne peut en spécifier plus de 20. Il indique aussi que tout produit doit être offert par au moins un fournisseur et que les offres d'un fournisseur sont limitées à 10.

16.8.6 Taille de la population d'un type d'associations

L'étude des relations qui existent entre la taille de la population d'un type d'associations et de celles de ses types d'entités participants conduit à des résultats qui nous

seront très utiles dans la suite.

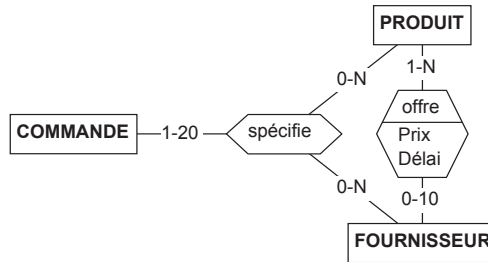


Figure 16.11 - Contraintes de cardinalité de rôles

Soit un type d'associations R entre A et B (figure 16.12). Ses contraintes de cardinalité ont été représentées symboliquement par $[m_A-M_A]$ et $[m_B-M_B]$. Il existe N_A entités A et N_B entités B tandis que la taille de la population de R est de N_R .

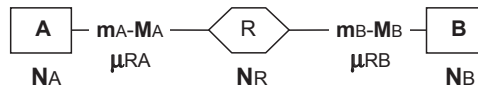


Figure 16.12 - Paramètres principaux d'un type d'associations binaire

Considérons ensuite μ_{RA} la *cardinalité moyenne* du rôle R.A, c'est-à-dire le nombre moyen, à un instant donné, d'associations R par entité A et μ_{RB} la cardinalité moyenne du rôle R.B. On a bien sûr les contraintes :

$$m_A \leq \mu_{RA} \leq M_A$$

$$m_B \leq \mu_{RB} \leq M_B$$

Si $m_A = M_A$ alors $m_A = \mu_{RA} = M_A$.

Le nombre d'associations de R peut être calculé de deux manières :

$$N_R = N_A \times \mu_{RA}$$

$$N_R = N_B \times \mu_{RB}$$

En éliminant N_R on en tire deux expressions remarquables :

$$N_A = N_B \times (\mu_{RB} / \mu_{RA})$$

$$N_B = N_A \times (\mu_{RA} / \mu_{RB})$$

Ces calculs sont valables pour les types d'associations cycliques et se généralisent immédiatement aux types d'associations n-aires⁹. Un modèle plus complet de quantification d'un schéma Entité-association est proposé dans [Hainaut, 1996].

9. Ce qui ne sera pas le cas en UML.

16.9 TYPES D'ENTITÉS

L'extension la plus spectaculaire du modèle de base est celle qui admet qu'une entité puisse **appartenir à plus d'un type**, ce qui est représenté par une nouvelle construction, la relation de *généralisation-spécialisation* ou relation *is-a*.

16.9.1 Relation de généralisation-spécialisation (relation *is-a*)

Le modèle de base fait l'hypothèse que *chaque entité appartient à un et un seul type* (figure 12.2). Étant donné deux types d'entités distincts A et B, on a obligatoirement $\text{pop}(A) \cap \text{pop}(B) = \emptyset$. Dans la réalité, les choses sont plus complexes : une personne est aussi un employé de son entreprise, un contribuable, un client du magasin du coin, un abonné au gaz et un parent d'élève. Comme membre de chacune de ces catégories, cette personne possède des propriétés, participe à des associations et est soumise à des contraintes propres à cette catégorie.

Par exemple, c'est en tant qu'*abonné au gaz* qu'elle dispose d'un *numéro de compteur* et en tant que *parent d'élève* qu'elle est en relation avec une ou plusieurs écoles. Reconnaître ces faits et disposer de moyens de les représenter explicitement dans un schéma doit nous aider à mieux comprendre et décrire les concepts que nous voulons représenter dans une base de données. La figure 16.13 (gauche) représente des ensembles d'entités qui se partagent certains éléments.

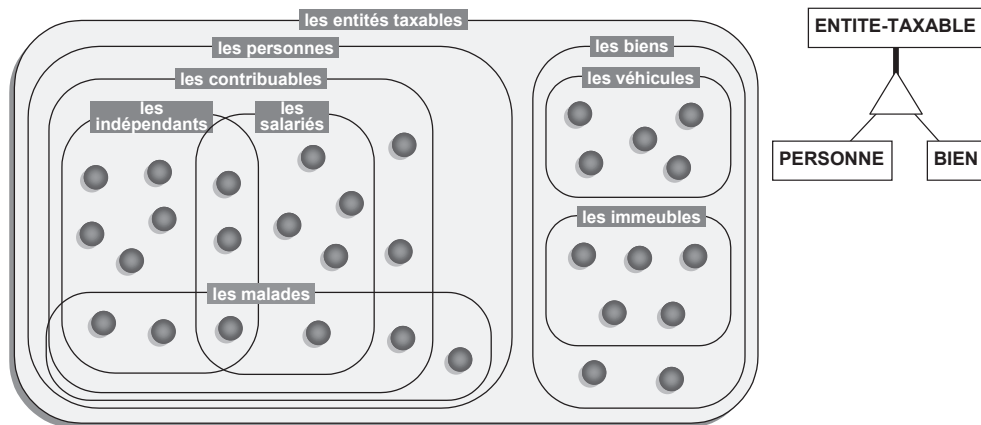


Figure 16.13 - A gauche, répartition d'un ensemble d'entités selon différentes catégories. A droite, représentation graphique de la relation *is-a* entre le surtype ENTITE-TAXABLE et ses sous-types PERSONNE et BIEN

Ce graphe montre que *tout bien est une entité taxable* et que *tout salarié est une personne*. La population d'un type d'entités est donc incluse dans celle d'un autre type. Le premier est dit *sous-type* du second, lequel est un *surtype* du premier. Pour représenter ces relations entre ces ensembles, on utilisera une construction nouvelle, la relation *is-a* (*est un(e)* en anglais), dont la représentation graphique est illustrée à la figure 16.13 (droite). On y indique que le type d'entités ENTITE-TAXABLE possède deux sous-types PERSONNE et BIEN, et qu'il est donc un surtype de ces

derniers. On dit aussi que PERSONNE est une *spécialisation* d'ENTITE-TAXABLE et que ENTITE-TAXABLE est une *généralisation* de PERSONNE et BIEN.

16.9.2 Propriétés et contraintes ensemblistes des sous-types

Il est utile de préciser les relations qui existent, et donc qu'on désire imposer, entre les populations d'un surtype et de ses sous-types. Ces relations constituent les réponses à deux questions :

- 1. les populations des sous-types d'un surtype sont-elles *disjointes* ou peuvent-elles contenir des entités communes (auquel cas elles sont *en recouvrement*) ?
- 2. toutes les entités du surtype appartiennent-elles à un (ou plusieurs) de ses sous-type(s) ou peut-il exister des entités du surtype qui ne sont reprises dans aucun sous-type ?

La première contrainte est celle de *disjonction*, qui sera indiquée graphiquement par le symbole D (disjonction) ou par son absence (non disjonction). La seconde est celle de la *couverture* représentée par le symbole T (couverture totale) ou son absence (couverture partielle).

Configuration	T	D	Exemples de populations	Schéma
couverture partielle + recouvrement = libre	-	-		
couverture totale + recouvrement	T	-		
couverture partielle + disjonction	-	D		
couverture totale + disjonction = partition	T	D		

Figure 16.14 - Les quatre configurations de sous-types

Le tableau de la figure 16.14 décrit les quatre configurations définies par ces contraintes et leur représentation. Les propriétés D et T simultanées définissent une

partition qui se représente par le symbole P.

Ces relations ensemblistes entre les populations de A, B et C induisent d’autres relations entre les tailles de celles-ci. Dans tous les cas, on a $NA \geq NB$. On a également :

Contraintes	Propriétés
-	$NA \geq \max(NB, NC)$
T	$NA \leq NB + NC$
D	$NA \geq NB + NC$
P	$NA = NB + NC$

Le schéma de la figure 16.15 explicite les propriétés ensemblistes des ensembles d’entités de la figure 16.13.

CONTRIBUABLE est un *surtype direct* de SALARIE tandis que PERSONNE et ENTITE-TAXABLE en sont des *surtypes indirects*. À l’inverse, CONTRIBUABLE est un *sous-type direct* de PERSONNE et INDEPENDANT en est un *sous-type indirect*.

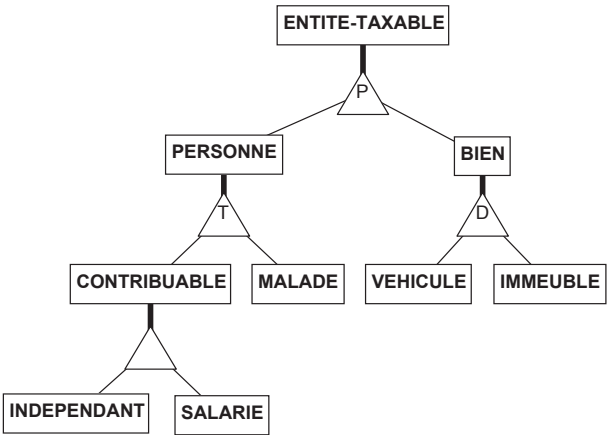


Figure 16.15 - Description des relations entre les ensembles d’entités de la figure 16.13

16.9.3 L’héritage

Une entité peut donc appartenir à plus d’un type. Elle possède les attributs, joue les rôles et est soumise aux contraintes de tous les types auxquels elle appartient, c’est-à-dire son type propre ainsi que tous ses surtypes directs et indirects. On dira qu’elle *hérite* des caractéristiques de tous ses types.

Nous examinerons cette propriété à partir de l’exemple de la figure 16.16. On y relève le surtype SALARIE qui admet deux sous-types EMPLOYE et OUVRIER. Selon la lecture habituelle :

- un salarié est caractérisé par un matricule identifiant et un nom ; un dossier personnel peut lui être associé ;

- un ouvrier est affilié à une caisse d'assurance et est caractérisé par un numéro d'affiliation et un régime salarial.

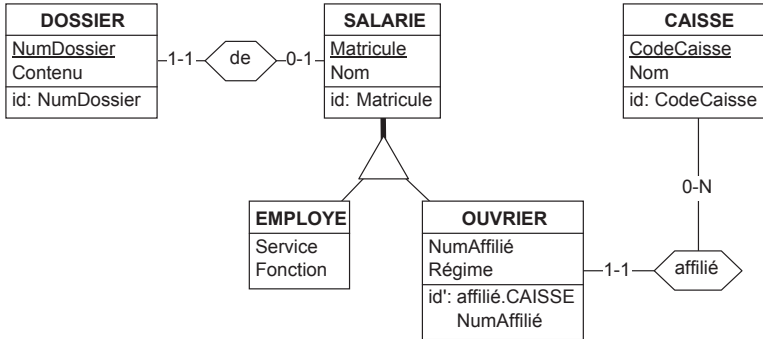


Figure 16.16 - Étude de l'héritage

Cependant, puisque OUVRIER est un sous-type de SALARIE, et donc que tout ouvrier est un salarié, on peut aussi inférer de ce schéma que :

un ouvrier est affilié à une caisse d'assurance et est caractérisé par un matricule identifiant, un nom, un numéro d'affiliation et un régime salarial ; un dossier personnel peut lui être associé ;

Cette dernière interprétation résulte du mécanisme d'**héritage descendant**, qui veut qu'un type d'entités hérite de toutes les caractéristiques (attributs, rôles, contraintes) de ses surtypes directs et indirects (figure 16.17, gauche). Ses caractéristiques sont donc constituées de ses *caractéristiques propres* auxquelles s'ajoutent les *caractéristiques héritées*. Il existe évidemment des contraintes imposées aux types d'entités d'une même branche de la hiérarchie : par exemple, les attributs (propres et hérités) d'un type d'entités doivent avoir des noms distincts et l'ensemble des contraintes d'intégrité d'un type d'entités ne peuvent être contradictoires. Ainsi, un sous-type ne peut posséder d'identifiant primaire si un de ses surtypes en possède déjà un.

Une caractéristique héritée peut être restreinte par rapport à sa définition dans le surtype. Par exemple :

- Le domaine d'un attribut peut être limité à un sous-ensemble du domaine d'origine ; le type d'entités DEPARTEMENT transmet son attribut NOM à son sous-type DOM-ROM¹⁰, mais il est évident que son domaine de valeur y est fortement restreint !
- La contrainte de cardinalité d'un rôle ou d'un attribut peut être plus étroite dans le sous-type : 1-1 au lieu de 0-1, 0-1 au lieu de 0-N.

On considère également l'**héritage ascendant**, selon lequel *certaines entités* d'un surtype sont dotées des caractéristiques d'un sous-type. Ainsi, *certaines salariés* sont affiliés à une caisse d'assurance (figure 16.17, droite)¹¹.

10. Départements et régions d'outre-mer.

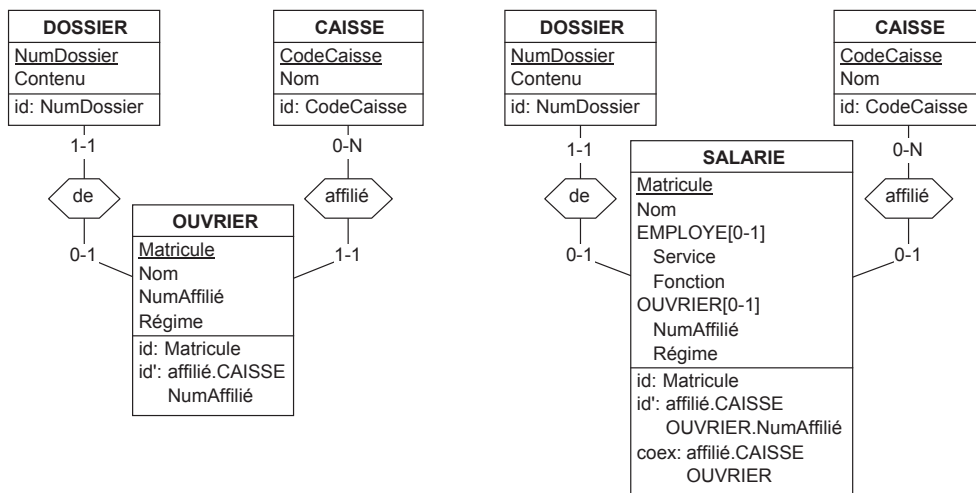


Figure 16.17 - Effet de l'héritage descendant sur le type d'entités OUVRIER (gauche) et de l'héritage ascendant sur SALARIE (droite)

L'*héritage collatéral* est la composition des deux autres mécanismes en cas de *non disjonction*. On attribue les caractéristiques d'un sous-type à chacun des autres sous-types du même surtype. On peut par exemple parler de la caisse d'assurance de certains employés. On notera qu'au plus un sous-type peut être doté d'un identifiant primaire, sinon il existerait des entités qui en posséderaient plusieurs.

16.9.4 Les sous-types non disjoints sont-ils vraiment utiles ?

Aucun SGBD ni langage de programmation n'est capable de traduire **directement** une structure formée d'un surtype et d'un ensemble de **sous-types non disjoints**. On peut par conséquent s'attendre à quelques difficultés lors de sa traduction dans un outil de gestion de données concret. Il est donc légitime de se poser la question de l'utilité de cette structure. Les exemples concrets de ce pattern sont cependant nombreux. Citons par exemple les différents rôles que peuvent jouer les membres d'une communauté : école, parti politique, association caritative, etc. Le schéma de la figure 16.18 en est une illustration tirée du domaine des conférences scientifiques : un membre de la communauté peut être à la fois un membre du comité scientifique (et évaluateur d'articles soumis) et auteur d'un article soumis à évaluation.

Eviter d'utiliser ce pattern dans un schéma revient tout simplement à le traduire prématurément dans une structure logique. Ignorer un mot ne supprime pas le concept qu'il désigne, du moins en démocratie !¹²

11. La contrainte *coex* sera étudiée à la section 16.11.3.

12. Le *novlangue*, langage imaginé par G. Orwell dans son roman *1984*, excluait des mots jugés subversifs par le pouvoir, dans le but d'éliminer les concepts nécessaires à l'esprit critique.

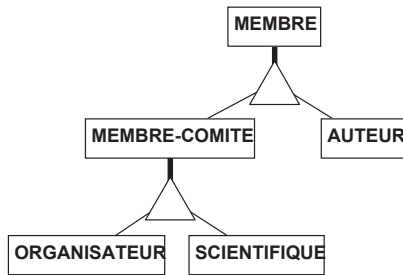


Figure 16.18 - Exemples de sous-types non disjoints

16.9.5 Surtypes et héritage multiples

Rien n'interdit qu'un type d'entités soit sous-type de plus d'un surtype. La figure 16.19 montre que le type d'entités OFFICIER est un sous-type de MILITAIRE et un sous-type de DIPLOME (prononcer *diplômé*). Cette structure traduit l'idée qu'un officier est à la fois militaire et diplômé.

Le schéma indique qu'un *officier*, en tant que *personne*, *militaire* et *diplômé*, possède un numéro national, un nom, un matricule militaire, une date d'incorporation, un numéro d'inscription à un programme d'enseignement et une date d'obtention de son diplôme. C'est en tant qu'officier qu'il possède un rang.

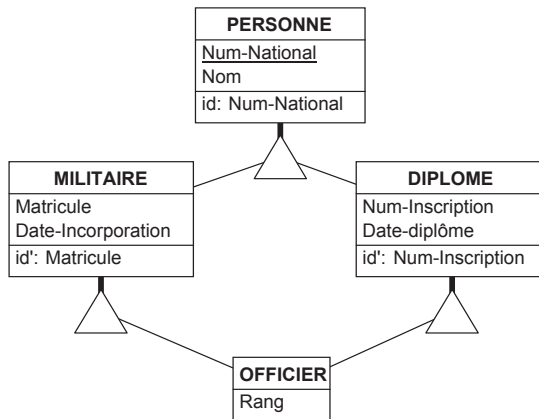


Figure 16.19 - Hiérarchie d'héritage multiple

Attention cependant, il peut exister des *militaires diplômés* qui ne sont pas officiers. Mathématiquement parlant, la population d'OFFICIER est *une partie* de l'intersection de celles de MILITAIRE et de DIPLOME mais ne lui est pas nécessairement égale (figure 16.20).

Il est une question fondamentale à laquelle nous devons répondre avant de poursuivre : les populations de deux types d'entités quelconques qui n'ont pas de surtype (direct ou indirect) commun sont-elles disjointes ou non ?

Nous postulons¹³ qu'elles sont *disjointes*, ce qui nous conduit à formuler les deux règles suivantes :

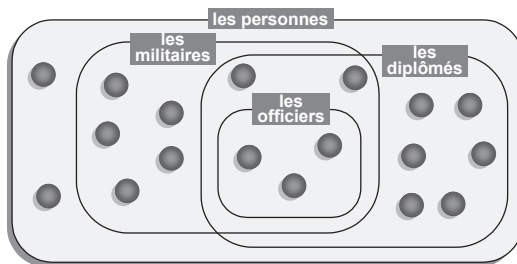


Figure 16.20 - Surtypes et héritage multiples : exemple de populations

1. MILITAIRE et DIPLOME ont un surtype (direct ou indirect) commun. Si ce n'était pas le cas, la population d'OFFICIER serait vide par construction.
Règle : si un type possède plusieurs surtypes directs, ces derniers doivent posséder un surtype direct ou indirect commun.
2. On considère les surtypes directs de OFFICIER, c'est-à-dire MILITAIRE et DIPLOME. On considère ensuite le surtype (direct ou indirect) le plus proche de ces surtypes communs, soit PERSONNE. Les sous-types de ce dernier **ne peuvent** être soumis à une contrainte de **disjonction**. En effet, dans le cas contraire, la population d'OFFICIER serait également vide par construction.
Règle : si un type possède plusieurs surtypes directs, ces derniers doivent posséder (au moins) un surtype direct ou indirect commun. Considérant les plus proches de ces surtypes, leurs sous-types ne peuvent être disjoints.

La figure 16.21 montre deux structures qui ne respectent pas ces règles. Sans être erronées, elle posent des problèmes indésirables de cohérence : pourquoi définir un type d'entités (D) dont on sait d'avance que sa population sera toujours vide ? On dira que le type d'entités D est **non satisfiable** dans chacun des deux schémas. Ce problème sera repris à la section 18.6.2.

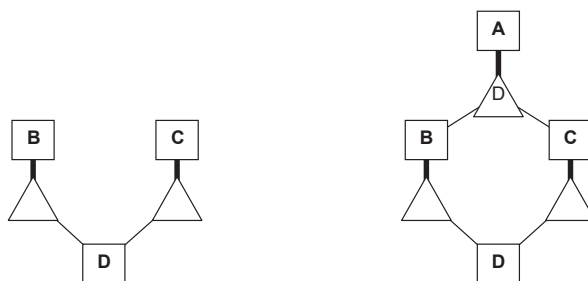


Figure 16.21 - Deux schémas problématiques : la population de D est vide par construction. D est *non satisfiable*.

13. Il s'agit d'un choix arbitraire (une sorte d'*axiome d'Euclide* du modèle Entité-association !). Le choix opposé aurait été moins intuitif et plus contraignant. En effet, dans la plupart des schémas, les (nombreux) types d'entités du plus haut niveau sont naturellement disjoints, de sorte qu'il serait nécessaire de créer des surtypes artificiels pour déclarer cette disjonction.

Remarque

On impose qu'un type d'entités ne puisse être, directement ou indirectement, un sous-type de lui-même. Le graphe des relations *is-a* ne peut donc comporter de circuit. On dira qu'il est *acyclique*.

16.9.6 Contraintes portant sur des composants hérités

Un attribut hérité peut intervenir dans une contrainte, son nom étant préfixé du nom de son type d'entités. Ainsi, dans le schéma de la figure 16.16, le type d'entités OUVRIER peut faire l'objet de contraintes impliquant, outre ses attributs et rôles propres, les attributs SALARIE.Matricule, SALARIE.Nom et le rôle de.DOSSIER. On en trouvera un exemple concret à la figure 18.40.

16.9.7 Répartitions multiples

L'ensemble des sous-types d'un surtype est généralement défini sur la base d'un critère de répartition clairement identifié¹⁴. Dans le cas de la figure 16.16 on répartit les salariés selon leur statut : *employé* et *ouvrier*. Rien n'interdit d'envisager la répartition des entités d'un type selon plus d'un critère. La figure 16.22 (gauche) représente la répartition des entités PERSONNE selon deux critères indépendants : le *statut* (employé/ouvrier) et le *sexe* (femme/homme). Cependant, cette structure de répartition multiple est rarement autorisée tant par les ateliers logiciels que par les SGBD et les langages OO. On en sera souvent réduit à des palliatifs tels que celui qui est illustré dans le schéma de droite basé sur le produit cartésien des deux répartitions, moins expressif mais aisé à mettre en œuvre.

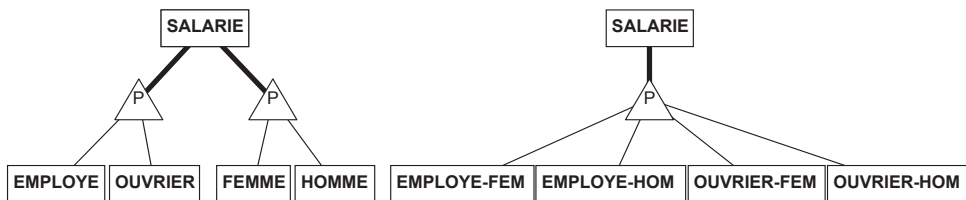


Figure 16.22 - Deux répartitions indépendantes des entités PERSONNE

La figure 16.23 montre un exemple de répartition de populations selon les deux critères. D'autres techniques sont proposées à l'annexe A25.

16.9.8 Règle de répartition en sous-types

Le critère de répartition entre sous-types est, jusqu'ici, implicite ou plus précisément **intrinsèque** : un *salarié* est un *homme* non pas en raison de propriétés particulières mais simplement parce qu'on l'a rangé dans la catégorie HOMME

14. Il faut admettre que les hiérarchies *is-a* des figures 16.15 et 16.19 ne répondent pas entièrement à ce critère. Leur objectif est en effet surtout d'illustrer les concepts.

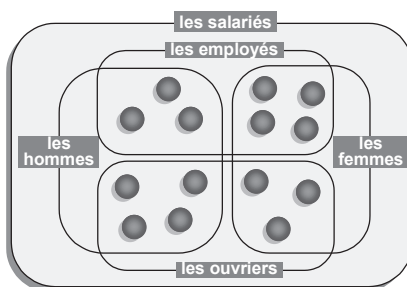


Figure 16.23 - Deux répartitions indépendantes : exemple de populations

Il existe une autre manière de répartir les entités d'un surtype, en fonction de leur état¹⁵. Ainsi, une entité SALARIE se rangera automatiquement dans la catégorie HOMME lorsque sa valeur de SEXE = 'H'. On désigne parfois un tel attribut de *discriminateur*. On exprime donc explicitement le critère de répartition. On appellera cette répartition **conditionnelle**. La figure 16.24 illustre ces deux modes de répartition. On propose deux formes de la répartition conditionnelle. La première (schéma central) est adoptée par UML.

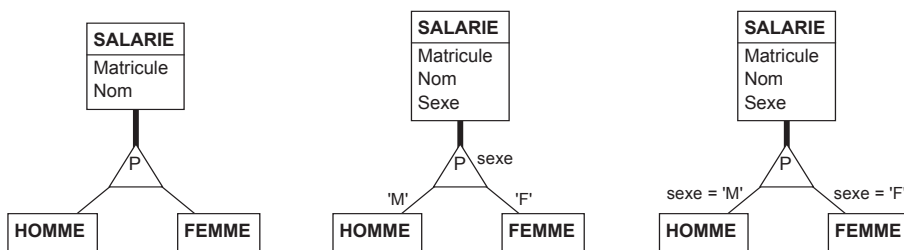


Figure 16.24 - Répartitions intrinsèque et conditionnelles selon les sous-types

Certains concepteurs utilisent la répartition conditionnelle sur la base d'un indicateur de sous-type. Les valeurs de cet indicateur spécifient explicitement le(s) sous-type(s) de chaque entité du surtype. Le schéma de gauche de la figure 16.25 utilise l'attribut Type comme indicateur.

On y indique symboliquement le domaine de valeurs de cet attribut : Type = {'E', 'O'}. En l'absence de contrainte de disjonction, l'attribut Type est *multivalué*. Sa cardinalité dépend des contraintes ensemblistes affectant les sous-types selon les règles de la figure 16.26.

Le schéma de droite de la figure 16.25 utilise autant d'indicateurs booléens que SALARIE compte de sous-types. Cet ensemble d'indicateurs est soumis à des contraintes exprimant les propriétés ensemblistes des sous-types ; pour une partition, un et un seul indicateur doit être à true. Une telle condition peut être difficile à exprimer. On suggère dans ce cas deux autres modes d'expression des indicateurs :

15. J'avais initialement écrit "en fonction de l'état de leurs attributs", mais un collègue m'a prudemment déconseillé cette formulation.

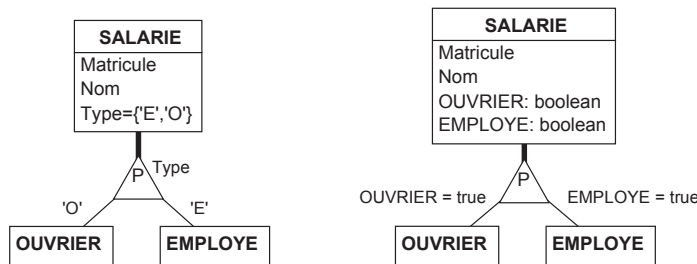


Figure 16.25 - Répartition conditionnelle par indicateur de sous-type(s)

Contrainte de sous-types	Attribut indicateur
P	Type[1-1] = {'E','O'}
D	Type[0-1] = {'E','O'}
T	Type[1-2] = {'E','O'}
-	Type[0-2] = {'E','O'}

Figure 16.26 - Forme de l'attribut indicateur

1. les indicateurs sont des *attributs facultatifs* ; il est alors possible d'utiliser les contraintes d'existence pour exprimer les propriétés des sous-types (pour une partition, exact-1: OUVRIER, EMPLOYE) ;
2. les indicateurs sont des *entiers* qui peuvent prendre les valeurs 0 ou 1 ; les contraintes de sous-types s'expriment simplement par une contrainte sur la somme de leurs valeurs (pour une partition, OUVRIER + EMPLOYE = 1).

16.10 IDENTIFIANTS

Dans le modèle de base, seul un type d'entités est doté d'un ou de plusieurs identifiants. L'extension des structures du modèle entraîne également une extension de la notion d'identifiant.

16.10.1 Identifiants d'un type d'entités

La typologie étudiée dans le modèle de base (12.6) reste d'application. On y ajoutera deux cas, illustrés à la figure 16.27.

- Un type d'entités peut n'avoir *aucun identifiant*¹⁶ (gauche). Bien que ce cas soit plutôt rare, il existe des situations où aucun identifiant certain n'émerge de

16. Il est possible que, lors de la mise en œuvre, la table qui traduit ce type d'entités doive posséder un identifiant, mais celui-ci sera ajouté lors de la conception logique pour des raisons techniques. Cette obligation ne se présenterait pas pour un schéma orienté objet. On évitera donc de l'introduire à ce stade de l'analyse.

- l'analyse. Un répertoire des *visiteurs* d'une exposition ne comporte, a priori, aucun identifiant.
- Un *attribut multivalué* (centre) ou un *composant d'un attribut complexe* (droite) peuvent constituer un identifiant de leur type d'entités. Dans les deux cas, l'attribut NumCompte constitue un identifiant multivalué de CLIENT. En effet, chaque valeur (notée NumCompte[*] au centre et Compte[*].NumCompte à droite), qui représente un numéro de compte bancaire personnel, n'apparaît que chez un seul client. La notation "<nom>[*]" témoigne du fait que c'est chaque valeur qui constitue à elle seule un identifiant et pas seulement l'ensemble des valeurs.

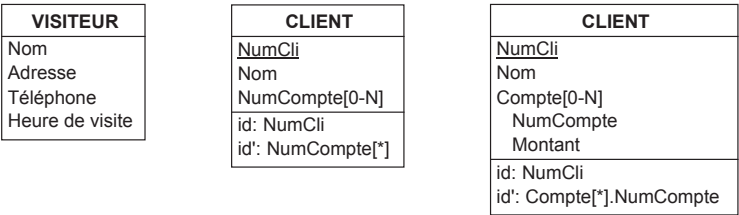


Figure 16.27 - Type d'entités sans identifiant et identifiants multivalués

Il n'est pas nécessaire que les ensembles de composants des deux identifiants soient disjoints. La figure 16.28 représente les relations entre des séminaires de formation (SEMINAIRE) et les instructeurs (INSTRUCTEUR) qui les animent. Un séminaire est décomposé en interventions (INTERVENTION) assurées chacune par un instructeur. Les interventions durant une journée entière, on en déduit les deux identifiants (déclarés secondaires par symétrie), qui se partagent l'attribut Date.

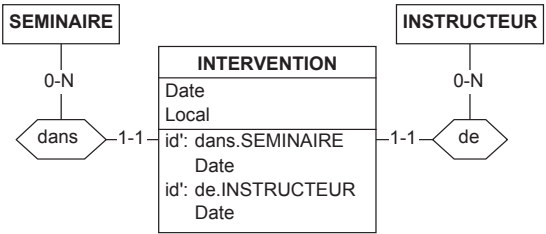


Figure 16.28 - Deux identifiants minimaux non disjoints

Le tableau ci-après reprend les différentes compositions admises des identifiants de types d'entités. D'autres combinaisons pourraient s'avérer utiles, mais nous aurions alors quelques problèmes pour les traduire correctement en structures de tables.

	attributs locaux monovalués	attributs locaux multivalués	rôles opposés
forme 1	1 ou plus	0	0
forme 2	1 ou plus	0	1
forme 3	0, 1 ou plus	0	2 ou plus
forme 3	0	1	0

Rappelons encore deux termes que le lecteur pourra rencontrer au hasard de ses lectures : **type d'entités faible** et **identifiant local** ou **relatif**. Ces concepts ont été détaillés à l'annexe A12.

Identifiants implicites d'un type d'entités

Tout rôle de cardinalité [0-1] ou [1-1] constitue un identifiant pour le(s) type(s) d'entités des autres rôles. Dans le schéma 16.1, {appartient.VEHICULE} et {signe.CONTRAT} sont deux identifiants de CLIENT. En effet, il n'existe qu'un client signataire d'un contrat déterminé. De même, {couvre.CONTRAT} est un identifiant de VEHICULE. Ces identifiants ne doivent pas être déclarés puisqu'ils se déduisent des propriétés des types d'associations. On les qualifie d'*implicites*.

Identifiant global de types d'entités

Il est parfois nécessaire d'indiquer qu'il n'existe pas plus d'une entité possédant une certaine valeur d'un attribut non pas parmi la population d'un type d'entités mais dans l'union des populations de plusieurs types. qu'un attribut commun à deux ou plusieurs types d'entités identifie une entité dans l'union des populations de ces types. Par exemple, considérant les types d'entités CLIENT, FOURNISSEUR et EMPLOYE, tous possédant un attribut ID de type numérique entier, on déclare {ID} **identifiant global**¹⁷ de {CLIENT, FOURNISSEUR, EMPLOYE}, signifiant par là que toutes les entités de ces types réunis ont des valeurs distinctes de ID. On trouvera rarement cette contrainte dans les schémas conceptuels purs, mais elle apparaîtra fréquemment dans les schémas logiques relationnels ou XML par exemple. Si les relations *is-a* sont autorisées, on pourra exprimer cette propriété par un surtype composé exclusivement des composants de cet identifiant, mais au risque de rendre le schéma plus complexe. On trouvera un exemple de ce type d'identifiant et de ses expressions à la section 19.6.3.

16.10.2 Identifiants d'un type d'associations

En principe, il n'existe pas deux associations de même type R entre les mêmes entités. En d'autres termes, *l'identifiant d'un type d'associations est constitué de l'ensemble de ses rôles*. C'est à cette constatation que se limitent en général les ouvrages décrivant le modèle Entité-association. Cependant :

- cet identifiant n'est pas toujours minimal ;
- il peut exister plus d'un identifiant ;
- certaines situations exigent que deux associations soient définies sur les mêmes entités, et qu'elles se distinguent seulement par les valeurs d'attributs de R.

L'interprétation relationnelle des types d'associations (section 16.17.1) nous fournira un cadre simple dans lequel nous pourrions étudier les identifiants des types d'associations. Nous examinerons d'abord les identifiants implicites, qui sont

17. Darwen utilise le terme de *distributed key* [Darwen, 2003].

induits par les cardinalités des rôles puis les identifiants explicitement déclarés. Les types d’associations sont supposés *normalisés* (section 18.7).

Identifiants implicites d’un type d’associations

Un identifiant est dit *implicite* s’il se déduit des cardinalités des rôles du type d’associations. Soit le type d’associations R dans lequel le type d’entités E joue un rôle r de cardinalité [i-j]. Il vient :

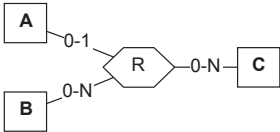
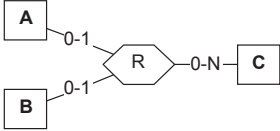
- Si $j = 1$, alors une entité quelconque de type E ne peut jouer le rôle r dans plus d’une association R. Ce rôle constitue donc un identifiant de R.
- Si $j > 1$, alors une entité quelconque de type E peut jouer le rôle r dans un nombre quelconque d’associations R. Ce rôle ne constitue pas à lui seul un identifiant de R.
- Si tous les rôles de R ont une cardinalité maximum supérieure à 1 et en l’absence d’identifiant explicite, alors l’ensemble de ses rôles constitue un identifiant minimal de R.

Les éventuels attributs du type d’associations n’interviennent pas dans les identifiants implicites. La figure 16.29 illustre l’application de ces règles aux principales configurations d’un type d’associations ternaire. Le traitement des types d’associations binaires sera proposé à titre d’exercice (A16.35 de l’annexe A16).

Identifiants explicites d’un type d’associations

Lorsque les identifiants implicites ne décrivent pas correctement les propriétés du domaine d’application, il faudra déclarer explicitement les identifiants appropriés. Ceux-ci seront composés de rôles et d’attributs du type d’associations. Nous analyserons deux exemples avant de formuler des règles générales.

Retournons à la figure 16.10. 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. La gestion de la bibliothèque est telle qu’un exemplaire n’est à nouveau disponible au plus tôt que le lendemain de sa restitution.

Typologie	Schéma	Interprétation	Identifiants
1:N:N		$R(\underline{A}, B, C)$	{A}
1:1:N		$R(\underline{A}, \underline{B}, C)$	{A}, {B}

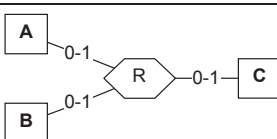
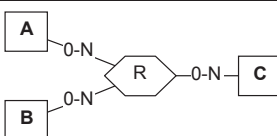
1:1:1		$R(\underline{A}, \underline{B}, \underline{C})$	$\{A\}, \{B\}, \{C\}$
N:N:N		$R(\underline{A}, \underline{B}, \underline{C})$	$\{A, B, C\}$

Figure 16.29 - Identifiants implicites d'un type d'associations ternaire

Il ne peut donc exister deux emprunts du même exemplaire le même jour. On traduit cette règle par un identifiant d'emprunte formé du rôle EXEMPLAIRE et de l'attribut Date Emprunt (figure 16.30).

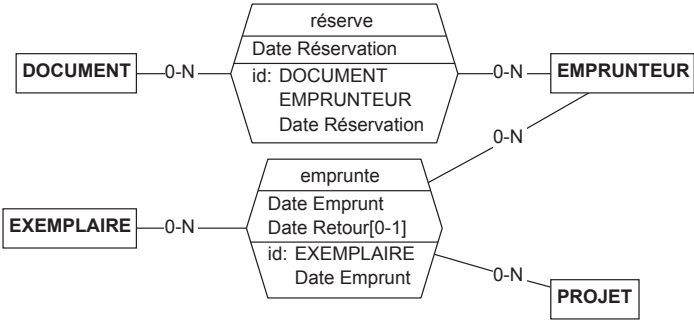


Figure 16.30 - Identifiants explicites de types d'associations

Chaque association réserve représente la réservation d'un document par un emprunteur. La date de réservation est spécifiée par un attribut de ce type d'associations. Quel est l'identifiant de réserve ? À première vue, il devrait être constitué des deux rôles {DOCUMENT, EMPRUNTEUR}. Ce serait pourtant une erreur. En effet, cet identifiant interdirait à un emprunteur de réserver plus d'une fois un même document, imposant ainsi une limitation injustifiée. Un emprunteur peut effectuer plusieurs réservations, mais à des dates différentes. D'où l'identifiant constitué des deux rôles et de l'attribut Date Réservation (figure 16.30).

Règles des identifiants d'un type d'associations

Les identifiants implicites d'un type d'associations R normalisé sont déterminés comme suit :

1. tout rôle de cardinalité [i-1] est un identifiant de R ;
2. si R ne possède pas de rôles de cardinalité [i-1], et en l'absence d'identifiant explicite, l'ensemble des rôles constitue un identifiant de R ;
3. en présence d'identifiant explicite, si l'ensemble des rôles de R constituent également un identifiant, celui-ci devra être explicitement déclaré.

Les identifiants implicites ne sont pas indiqués dans un schéma.

16.10.3 Identifiants d'un attribut complexe

Les valeurs d'un attribut composé multivalué A d'une entité déterminée sont distinctes par définition, puisqu'elles forment un ensemble. Peut-être cet identifiant n'est-il pas minimal. Tel serait le cas si un composant de A à lui seul rendait ces valeurs distinctes. Le type d'entités COMPTE de la figure 16.31 possède un attribut complexe Dépense qui enregistre chaque mois le total des dépenses affectées à ce compte. Les valeurs de cet attribut auront des valeurs de Mois distinctes. C'est ce qu'exprime la clause "id(Dépense): Mois". La présence d'un tel identifiant suggère peut-être une modélisation inadéquate, comme on le montrera dans l'exercice 16.4.

COMPTE
NumCompte
Intitulé
Dépense[0-12]
Mois
Montant
id: NumCompte
id(Dépense): Mois

Figure 16.31 - Identifiant minimal d'un attribut complexe

16.11 CONTRAINTES D'INTÉGRITÉ

Les identifiants, les cardinalités des rôles et les attributs obligatoires du modèle de base ne constituent qu'un échantillon réduit des contraintes dont on voudrait disposer pour modéliser avec précision les propriétés du domaine d'application. Le modèle Entité-association étendu offre des contraintes supplémentaires que nous décrivons ci-après.

Reprécisons d'abord ce qu'on entend par *contrainte d'intégrité* : il s'agit d'une propriété que les instances décrites par un schéma (les entités, les associations et les valeurs d'attributs) doivent respecter de manière à représenter de manière fidèle les situations et le comportement du domaine d'application. Ces contraintes constituent ce qu'on appelle parfois les *règles métier*¹⁸, dans la mesure où elles traduisent des propriétés du domaine d'application (le *métier*). Nous distinguerons les contraintes statiques des contraintes dynamiques.

Les contraintes statiques

Les contraintes d'intégrité statiques décrivent les états ou les situations valides. Les violations de telles contraintes peuvent être observées par un simple examen des instances à un moment donné. Les contraintes vues jusqu'à présent et celles que

18. Business rules

nous allons détailler dans cette section sont statiques. Quelques exemples supplémentaires :

- le prix unitaire d'un produit doit être supérieur à 0 ;
- il existe quatre valeurs de catégorie de clients : B1, B2, C1, C2 ;
- un client ne peut être de catégorie C2 que si son compte est non négatif ;
- toute commande doit comprendre au moins un détail.

Les contraintes dynamiques

Une contrainte d'intégrité dynamique spécifie les changements d'états valides. Elle précise les modifications autorisées. Les violations de contraintes dynamiques sont difficiles à observer. Il faut en effet comparer les états avant et après chaque modification pour y détecter des anomalies. 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 ; néanmoins, il est possible qu'un produit référencé tombe en rupture de stock (la contrainte est bien dynamique et pas statique) ;
- 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" aux statuts "marié" ou "cohabitant" mais pas aux statuts « divorcé », « séparé » ou « veuf ».

16.11.1 Contraintes de domaine

Une contrainte de domaine est associée à un attribut atomique ou à un TDU. Elle spécifie une restriction de l'ensemble de valeurs défini par le type de valeur initial.

- valeurs permises¹⁹ : Sexe char(1); Sexe = {'M','m','F','f'}
- intervalle : DateNaissance date; DateNaissance = [1920-1-1, 2019-12-31]
- règle de formation des valeurs : Code char(10); Code = [A-Z][0-9]*
- valeurs exclues²⁰ : Sigle char(10); Sigle not like '*CON*'

16.11.2 Contraintes entre valeurs

Une contrainte entre valeurs d'attributs spécifie une condition qui doit être satisfaite par les valeurs de certains attributs (figure 16.32).

16.11.3 Contraintes d'existence

Les contraintes d'existence définissent les conditions qui régissent la présence ou l'absence d'instances de composants facultatifs (attributs et rôles) associés à une

19. Contrainte aussi exprimée par un *domaine énuméré*.

20. On a adopté la syntaxe SQL.

entité, à une association ou à un attribut composé. On retient cinq contraintes, la *coexistence*, l'*exclusion*, *au moins un*, *exactement un* et l'*implication*, illustrées à la figure 16.33.

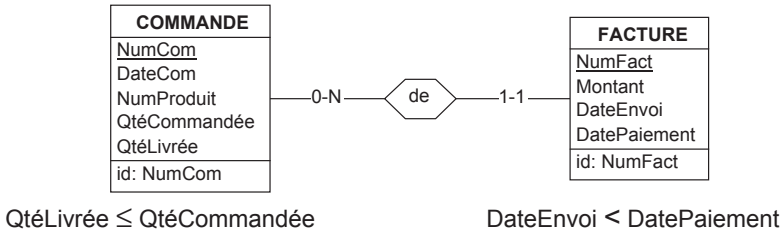


Figure 16.32 - Deux contraintes entre valeurs

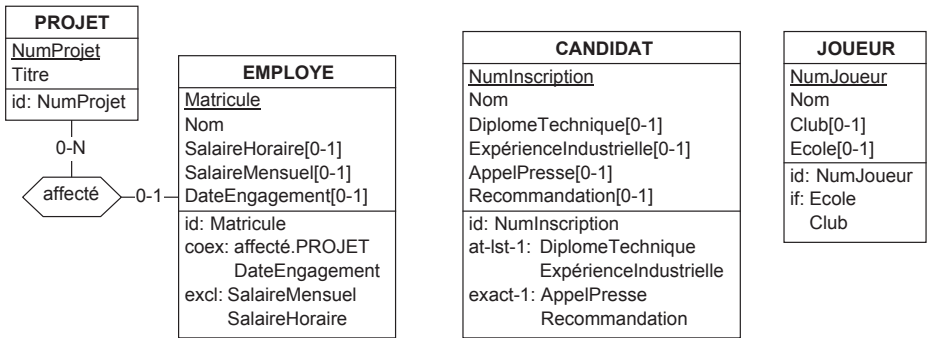


Figure 16.33 - Contraintes de *coexistence* (coex), d'*exclusion* (excl), *au moins un* (at-lst-1), *exactement un* (exact-1) et *implication* (if)

- **Contrainte de coexistence.** Le schéma de gauche représente des employés dont certains peuvent être affectés à un projet à partir d'une certaine date d'engagement. Il existe une règle intuitive : si un employé est affecté à un projet, alors il possède aussi une date d'engagement, et inversement. On dira que ces deux informations sont coexistantes, ce qu'on traduira par la *contrainte de coexistence* : coex: affecté.PROJET, DateEngagement.
- **Contrainte d'exclusion.** Les employés sont en outre soumis à une contrainte sur leur régime salarial : un employé reçoit un salaire selon une modalité au plus : *salaire mensuel* ou *salaire horaire* ou aucune mais pas les deux. D'où la *contrainte d'exclusion* : excl: SalaireMensuel, SalaireHoraire (schéma de gauche).
- **Contrainte au moins un (ou inclusif).** Le schéma central représente des candidats à un poste pour lesquels on a enregistré des informations de profil. Chaque candidat peut avoir un diplôme et une expérience industrielle. Une candidature est valide si le candidat peut justifier au moins d'un diplôme ou d'une expérience industrielle, ce qu'on traduit par la *contrainte au moins un (at least one)* : at-lst-1: DiplomeTechnique, ExpérienceIndustrielle.

- **Contrainte exactement un** (ou *exclusif*). En outre le candidat doit avoir répondu à un appel dans la presse ou être recommandé, mais pas les deux. On indique cette règle par une contrainte *exactement un* (*exactly one*) : `exact-1: AppelPresse, Recommandation` (schéma central). Cette contrainte est la conjonction des contraintes *exclusion* et *au moins un*.
- **Contrainte d'implication**. Le schéma de droite représente des joueurs qui peuvent chacun être affiliés à un club sportif et inscrits à une école de sport. Seuls les joueurs affiliés à un club peuvent s'inscrire à une école. Il s'agit d'une *contrainte d'implication*, qui stipule que si une valeur d'Ecole existe, alors une valeur de Club doit aussi exister, ce qu'on traduit par : `if: Ecole, Club` ²¹. La contrainte spécifie le composant dépendant (Ecole) suivi de la liste des composants nécessaires (Club).

Le tableau ci-dessous précise les conditions dans lesquelles chacune des cinq contraintes est satisfaite (1) ou non satisfaite (0) en termes de présence (1) ou d'absence (0) de chacun des deux composants A et B (deux premières colonnes). On se limite aux contraintes portant sur des groupes de deux composants, la généralisation à des groupes de plusieurs composants est aisée.

présence		contrainte				
A	B	coex: A, B	excl: A, B	at-lst-1: A, B	exact-1: A, B	if: A, B
0	0	1	1	0	0	1
0	1	0	1	1	1	1
1	0	0	1	1	1	0
1	1	1	0	1	0	1

L'exercice A16.30 étudie plus en détail les propriétés de ce tableau. Nous verrons dans la suite (section 18.7) que certaines de ces contraintes peuvent s'exprimer de manière plus explicite sous la forme de structures classiques.

16.11.4 Dépendances fonctionnelles

Considérant qu'un type d'entités regroupe un ensemble d'attributs et qu'un type d'associations est constitué d'un ensemble de rôles et d'attributs, il semble naturel d'admettre les *dépendances fonctionnelles* (DF) au titre de contraintes d'intégrité du modèle Entité-association. Nous verrons d'ailleurs à la section 16.17 que ce sentiment est parfaitement justifié.

Dans un *schéma normalisé* (nous reviendrons sur ce concept plus loin), il n'existe pas d'autres DF que celles qui ont pour déterminant un identifiant du type d'entités ou du type d'associations. Il n'est donc pas nécessaire de les mentionner. Dans certaines circonstances cependant, il conviendra de spécifier explicitement des DF

21. Selon le formalisme décrit dans la section suivante, et en admettant que la fonction `existe(E)` évalue l'existence d'une valeur pour l'expression E, on pourrait écrire : $\forall j \in \text{JOUVEUR}, \text{existe}(j.\text{Ecole}) \Rightarrow \text{existe}(j.\text{Club})$.

qui ne satisfont pas à cette propriété. Tel serait le cas d'un schéma conceptuel produit par un concepteur débutant, ou résultant de l'intégration de plusieurs schémas indépendants, ou encore obtenu par rétro-ingénierie d'une base de données existante (chapitre 22). Le schéma de la figure 16.34 comporte deux DF explicites. Celles-ci sont anormales et rendent donc le schéma non normalisé. La question de la normalisation d'un schéma Entité-association sera étudiée à la section 18.7.

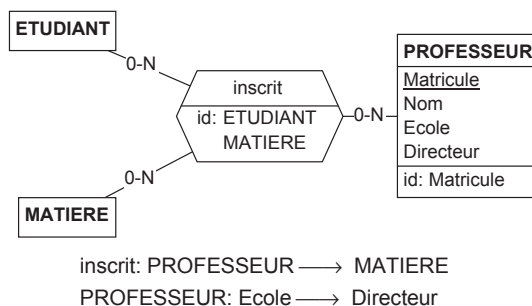


Figure 16.34 - Dépendances fonctionnelles explicites dans un schéma

16.12 CONTRAINTES D'INTÉGRITÉ ADDITIONNELLES

Il est évidemment impossible de répertorier toutes les contraintes possibles et de leur attribuer une dénomination et une représentation graphique comme nous avons pu le faire pour les identifiants, les cardinalités et les contraintes d'existence. Pour exprimer les autres contraintes, on recourra à un mode descriptif général, informel (en langage courant) ou formel (sous forme de prédicat).

16.12.1 Expression des contraintes

Les contraintes entre valeurs (16.11.2) constituent un exemple de langage formel élémentaire. Une formulation quelque peu précise se base sur un langage de contraintes plus ou moins puissant. Celui que nous avons utilisé pour les contraintes entre valeurs est simple et intuitif, mais très peu puissant. La première expression par exemple est à comprendre *pour chaque entité* **COMMANDE**. Dès qu'une contrainte est relative à plusieurs entités ou à des associations, un langage plus étendu sera nécessaire.

Afin d'éviter l'introduction de langages nouveaux, nous proposerons une formulation proche de la logique du premier ordre et décrivant les chemins selon une notation en cascade, à la manière de Pascal, XML XPath ou OCL d'UML (chapitre 17). Selon ce langage, l'expression simplifiée de la première contrainte du schéma 16.32 :

$$QtéLivree \leq QtéCommandée$$

s'écrira de manière plus explicite sous l'expression étendue :

$$\forall c \in \text{COMMANDE}, c.QtéLivree \leq c.QtéCommandée$$

La variable *c* désigne une entité COMMANDE quelconque, dite *entité courante*. Pour illustrer la nécessité d'un langage plus puissant, considérons la contrainte qui veut que *la date d'envoi d'une facture soit postérieure à la date de sa commande*. La désignation de cette dernière date nécessite une expression de navigation de l'entité FACTURE courante vers l'entité COMMANDE qui lui est associée via de :

$$\forall f \in \text{FACTURE}, f.\text{DateEnvoi} > f.\text{de.COMMANDE.DateCom}$$

L'expression *f.de.COMMANDE.DateCom* constitue un chemin qui désigne la *date de commande de la facture f*. On utilisera les formulations simplifiée ou étendue selon les besoins. La formation d'un chemin doit être affinée lorsqu'il emprunte un type d'associations cyclique. On utilisera dans ce cas, au lieu du nom du type d'entités cible, le nom du rôle qu'il joue. Considérant le schéma de la figure 16.36, on désignera le nom du responsable qui supervise la personne *p* par l'expression :

$$p.\text{supervise.responsable.Nom}$$

Bien que les expressions de chemins soient conçues pour les types d'associations binaires, il est possible de les généraliser aux types d'associations n-aires. On désignera par exemple l'emprunteur d'un exemplaire *e* dans le schéma de la figure 16.7 comme suit :

$$e.\text{emprunte.EMPRUNTEUR}$$

Le cas des types d'associations n-aires cycliques nécessite une clause supplémentaire, qui spécifie le rôle que joue l'entité source. On suggère la forme suivante, qui désigne les immeubles vendus par le client *c*, désigné en tant que vendeur :

$$c.\text{vendeur.vendu.IMMEUBLE}$$

16.12.2 Remarques sur les langages d'expression de contraintes

L'élaboration de langages de définition de contraintes portant sur des graphes est une question délicate. Les éléments que nous avons suggérés dans cette section n'ont pour but que d'illustrer le concept et non de proposer un langage complet et rigoureux. Ce dernier devrait notamment permettre d'imposer des conditions aux différents nœuds d'un chemin et de désigner les attributs de types d'associations. Un langage de contraintes comporte essentiellement deux fonctionnalités :

1. la désignation de collections d'instances d'objets du schéma ;
2. et la formulation de conditions sur ces collections.

Ces langages sont donc en général hybrides. Les mécanismes de construction de collections d'instances, ordonnées ou non, ensemblistes ou non, se retrouvent par exemple dans Z, Alloy, OCL que nous étudierons brièvement à la section 17.13, ou encore XPath. Ces mécanismes peuvent être de nature navigationnelle (notation en chaîne), mais aussi algébrique. Par exemple, l'expression navigationnelle :

$$c.\text{signe.CONTRAT.couvre.VEHICULE},$$

qui spécifie les véhicules pour lesquels le client *c* a signé un contrat de couverture (figure 16.35), pourrait s'écrire de manière algébrique²² selon la section 3.2.2.

signe*couvre(CLIENT=c)[VEHICULE]

La notation en chaîne d'un chemin convient bien aux schémas binaires, pour lesquels chacun définit une collection homogène (des entités VEHICULE dans les dernières expressions) tandis que la notation algébrique, plus générale mais moins intuitive, permet en outre de définir aisément des collections de n-uplets comme les couples (CLIENT, VEHICULE) par exemple :

signe*couvre[CLIENT,VEHICULE]

16.12.3 Cycles et contraintes cycliques

Un cycle est une chaîne de types d'associations binaires dont les deux extrémités sont un seul et même type d'entités. Plus précisément, il est constitué d'une suite $R_1(A,B)$, $R_2(B,C)$, $R_3(C,D)$, ..., $R_n(F,A)$ de types d'associations binaires²³ distincts²⁴ tels que le 2^e rôle de l'un et le 1^{er} rôle du suivant sont joués par le même type d'entités. Toute permutation circulaire d'une suite constitue évidemment le même cycle : la suite $R_2(B,C)$, ..., $R_n(F,A)$, $R_1(A,B)$ représente le même cycle que la première citée. Un schéma de taille normale comporte en général un grand nombre de cycles. Notre premier (et tout petit) schéma de figure 16.1 comporte le cycle :

- VEHICULE.appartient.CLIENT.signe.CONTRAT.couvre.VEHICULE

De même, celui de la figure 12.27 en comporte trois :

- SECTION.départ.STATION.arrivée.SECTION
- VOYAGE.conduit.AGENT.dirige.STATION.départ.SECTION.forme.LIGNE.suivant.VOYAGE
- VOYAGE.conduit.AGENT.dirige.STATION.arrivée.SECTION.forme.LIGNE.suivant.VOYAGE

La raison pour laquelle l'étude des cycles est intéressante est qu'ils sont souvent le siège de contraintes d'égalité ou d'inclusion. Le schéma de la figure 12.27 en est un exemple. En effet, une des propriétés des lignes ferroviaires est qu'une section relie deux stations différentes, propriété qui s'exprime comme suit :

$\forall s \in \text{SECTION}, s.\text{départ.STATION} \neq s.\text{arrivée.STATION}$

Considérons à présent le schéma de gauche de la figure 16.35. Sans autre information, il n'y a aucune raison de penser que le client qui signe le contrat couvrant un véhicule en est le propriétaire. Si on devait imposer qu'il le soit, on exprimerait cette contrainte comme suit :

$\forall v \in \text{VEHICULE}, v.\text{couvre.CONTRAT.signe.CLIENT} = v.\text{appartient.CLIENT}$

S'il n'en résulte aucune ambiguïté, on admettra les expressions simplifiées :

22. Ceci fait l'hypothèse d'une interprétation relationnelles qui sera décrite à la section 16.17.

23. Le concept est aisément généralisable à des types d'associations n-aires dont on considère la projection sur deux rôles.

24. On se limite aux *circuits élémentaires*, ne passant pas deux fois par le même type d'entités.

SECTION.départ.STATION $\neq$ SECTION.arrivée.STATION

VEHICULE.couvre.CONTRAT.signé.CLIENT = VEHICULE.appartient.CLIENT

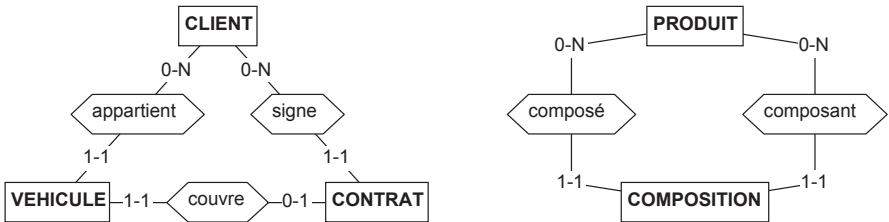


Figure 16.35 - Schémas comportant un cycle

16.12.4 Propriétés des graphes d'instances

Un type d'associations cyclique peut couvrir des **assemblages d'instances** dont certains ne sont pas nécessairement conformes au domaine d'application. D'une manière générale, les associations de ce type forment un graphe orienté²⁵, comme l'illustre la figure 16.36, où chaque pastille représente une entité PERSONNE et chaque arc est tiré d'une entité PERSONNE vers son responsable.

Ce graphe est par nature hiérarchique, et même un ensemble d'arbres. En effet, il est de règle qu'une personne ne puisse être, directement ou indirectement, le supérieur de son supérieur. Un arc de p1 vers p6 serait conforme au schéma, mais serait tout à fait illégal. On dira que le graphe des instances *ne peut comporter de circuits* (ce qu'on appelle communément, mais improprement, *acyclique*). Cette contrainte est très importante, mais ne peut être exprimée dans le modèle Entité-association ni d'ailleurs, comme on l'a déjà constaté, en SQL (voir section 8.7).

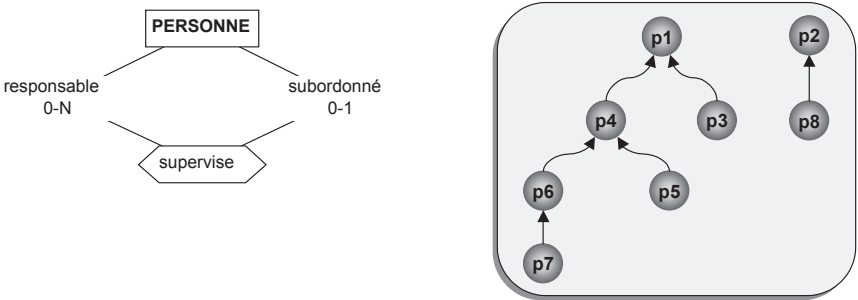


Figure 16.36 - Un type d'associations cyclique et des instances représentatives

Suggestion : définir un stéréotype «acyclic» et l'attribuer aux types d'associations soumis à une telle contrainte. Cette solution ne permet cependant pas de couvrir

25. La caractéristique orienté dérive du fait que les deux rôles du type d'associations sont distincts. Il n'est pas facile de représenter de manière élégante un graphe non orienté à l'aide du modèle Entité-association en raison du caractère symétrique des arcs du graphe.

tous les cas. Considérons par exemple le cycle PRODUIT.composé.COMPOSITION.composant.PRODUIT de la figure 16.35.

Les instances de ce fragment ne peuvent comporter de circuits (voir figure 8.6, par exemple), mais cette contrainte, qui affecte les instances de la *composition* de deux types d'associations, ne peut être spécifiée par un simple stéréotype.

16.12.5 Contraintes de redondance

En principe, un schéma conceptuel ne comporte pas de redondance. Il n'y existe aucun objet dont les instances seraient à tout instant entièrement calculables à partir d'instances d'autres objets du même schéma.

Dans la pratique, les choses sont parfois plus nuancées. Un objet redondant peut subsister si, dans le chef de l'utilisateur ou de l'analyste, sa présence rend le schéma plus lisible et plus naturel ou si elle permet d'exprimer plus simplement une contrainte d'intégrité. Un tel objet est acceptable pour autant qu'une contrainte d'intégrité exprime la redondance en précisant la formule de calcul des instances.

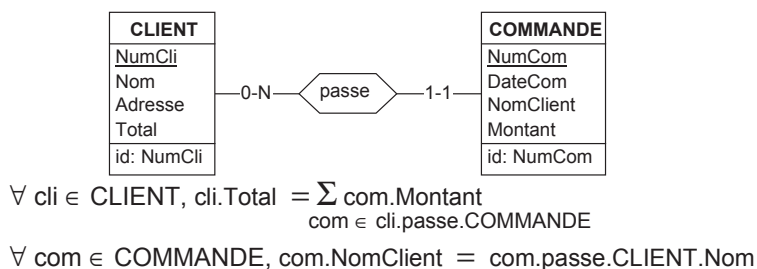


Figure 16.37 - Schéma comportant des attributs redondants

Le schéma de la figure 16.37 comporte deux attributs redondants. La valeur de l'attribut Total d'une entité CLIENT se calcule comme la somme des valeurs de Montant des entités COMMANDE qui lui sont associées. L'attribut NomClient d'une entité COMMANDE est une copie de l'attribut Nom de l'entité CLIENT qui lui est associée. Cette dernière contrainte admet une formulation simplifiée :

$$\text{COMMANDE.NomClient} = \text{COMMANDE.passe.COMMANDE.Nom}$$

16.13 LES NOMS

Revenons sur la question des noms qu'on donne aux objets d'un schéma, question qui a été laissée dans l'ombre au chapitre 13.

16.13.1 Règles de dénomination

On rappelle et complète les règles relatives aux noms des objets d'un schéma.

- Les types d'entités ont des noms distincts.
- Les types d'associations ont des noms distincts.

- Les attributs d'un même parent ont des noms distincts. Un attribut peut être désigné dans son schéma par son *nom absolu*, constitué de son nom préfixé du nom absolu de son parent : CLIENT.Localité, EMPRUNTEUR.Adresse.Ville, emprunte.Date Emprunt.
- Le nom d'un rôle peut être déclaré explicitement ; s'il ne l'est pas, ce nom est celui du type d'entités qui joue ce rôle. Les rôles d'un type d'associations ont des noms (locaux) distincts. Un rôle est désigné univoquement dans un schéma par son *nom absolu* R.r, où R est de nom du type d'associations et r celui du rôle. Le type d'associations de la figure 16.8 possède trois rôles dont les noms locaux sont vendeur, acheteur et IMMEUBLE. Au niveau du schéma, ils seront désignés par leurs noms absolus, respectivement vendu.vendeur, vendu.acheteur et vendu.IMMEUBLE.

Dans un schéma conceptuel, on n'impose généralement pas de syntaxe particulière. Un point délicat cependant : celui de la règle de formation des **noms composés**. Quel nom assigner à un attribut représentant un taux de TVA : Taux de TVA, Taux-de-TVA, Taux_de_TVA TauxTVA, TauxDeTVA ? Le choix peut être guidé par les règles du SGBD cible (SQL n'admet pas les espaces ni les tirets) ou d'un langage de programmation mais en suivant un tel argument, on lierait le schéma conceptuel à une technologie particulière. Certains outils peuvent imposer des contraintes qu'on sera tenu de respecter. Exemple fréquent : *un nom ne peut comporter d'espaces*.

Dans cet ouvrage, sauf exception, on adopte les conventions suivantes :

- le nom d'un type d'entités est en majuscules (CLIENT), le nom d'un type d'associations est en minuscules (signe), le nom (explicite) d'un rôle est en minuscules (supérieur) et le nom d'un attribut est capitalisé (Adresse).
- Le nom d'un type d'entités est au *singulier*, traduisant le concept plus que la population.

Noms multiples, noms identiques et noms absents

On peut parfois admettre qu'un objet porte plusieurs noms, que plusieurs objets portent le même nom, ou qu'ils n'en portent aucun. Ces questions sont discutées à l'annexe A16.

16.13.2 Dénomination des attributs

Il est fréquent que plusieurs types d'entités et d'associations possèdent des attributs de même nom et de signification similaire : Prénom d'EMPLOYE et Prénom de CLIENT par exemple, ces deux types d'entités n'ayant pas de surtype commun pour l'instant. Comment doit-on les nommer ? Deux approches sont fréquentes.

Selon la première, ces attributs **reçoivent le même nom**, Prénom, la distinction se faisant par préfixage : CLIENT.Prénom. Elle favorise la lecture naturelle et l'analyse du schéma, par exemple lorsqu'on recherche les composants similaires (pour identifier des surtypes) ou lors de l'intégration de plusieurs schémas.

La seconde approche assigne des **noms distincts** aux attributs d'un schéma : PrénomClient et PrénomEmployé, ou encore CLI-Prénom et EMP-Prénom par

exemple. On réduit ainsi la distance avec le schéma logique, dans lequel l'unicité des noms de colonnes dans une table oblige parfois à renommer les colonnes des clés étrangères. Elle garantit aussi l'unicité des attributs hérités.

16.13.3 Dénomination des types d'associations

Dans de nombreuses situations, le verbe d'une proposition élémentaire se traduit par un type d'associations, dont le nom est souvent construit à partir de ce verbe²⁶. Puisque le schéma conceptuel est un moyen de communiquer entre concepteurs et utilisateurs, il est normal qu'on cherche à attribuer aux objets de la traduction d'une proposition des noms qui permettent de reconstituer cette proposition.

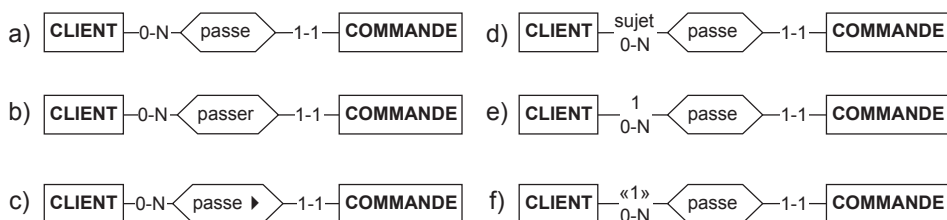


Figure 16.38 - Dénomination d'un type d'associations : six approches

- a) À la phrase « *un client passe des commandes* » on associe généralement le schéma de la figure 16.38/a. Ce schéma se lit naturellement de gauche à droite et, en tenant compte de la cardinalité [0-N], conduit à la reformulation « *un CLIENT passe des COMMANDEs* ». Malheureusement, cette convention n'est pas toujours praticable. Si nous inversons les positions des deux types d'entités, nous obtenons une relecture invalide : « *chaque COMMANDE passe un CLIENT* ». Une disposition verticale n'imposerait en outre aucun sens privilégié de lecture.
- b) Une deuxième approche consiste à neutraliser le verbe et utiliser sa forme infinitive comme nom du type d'associations : *passer* au lieu de *passe* ou *est passée* (schéma 16.38/b). Au lecteur de former une phrase correcte à partir de cette forme. L'infinitif « *passer* » sera à conjuguer selon le sens de lecture choisi : « *le CLIENT passe des COMMANDEs* » ou « *toute COMMANDE est passée par un CLIENT* ». Le résultat est sans doute moins élégant mais il est plus régulier.
- c) Les diagrammes de classes d'UML proposent d'indiquer le sens de lecture d'un type d'associations par un symbole directionnel attaché à son nom. Le schéma 16.38/c indique le sens de lecture de CLIENT vers COMMANDE. Cette solution n'est intéressante que si l'AGL UML ajuste automatiquement ce symbole lors d'un repositionnement graphique des objets d'un schéma.
- d) Le sens de lecture peut aussi être spécifié non pas pour le type d'associations mais via ses rôles. Lorsqu'une proposition du type « *sujet verbe objet* » est traduite par un type d'associations R, chaque substantif jouant un rôle (*sujet*,

26. On utilisera également, selon les circonstances, une préposition : *par*, *pour*, *via*, etc.

objet) dans la phrase est traduit par un rôle de R. Il suffit donc d'indiquer, par le nom « *sujet* » le rôle qui correspond au sujet de la proposition, l'autre s'en déduisant automatiquement. C'est ce que propose le schéma 16.38/d, dans lequel on indique que CLIENT est le *sujet* de la proposition. Dans cette approche, le verbe est évidemment conjugué en conséquence (*passé*).

- e) Le schéma 16.38/e est similaire, mais adopte un nom plus abstrait, en l'occurrence le numéro d'ordre dans la phrase. Il peut également être utilisé lorsque le nom du type d'associations n'est pas un verbe mais une préposition telle que « *de* », « *dans* », « *par* », « *pour* », etc.
- f) Les deux techniques d) et e) ne conviennent pas lorsque le rôle possède déjà un nom significatif. On adoptera alors une solution plus générale via un stéréotype «sujet» ou «1». Les rôles du schéma 16.38/f ne possède pas de nom mais l'un d'eux possède le stéréotype «1».

16.14 CLASSES D'OBJETS GÉNÉRIQUES

Dans certains schémas, on observe souvent la présence de constructions très générales, pas ou peu liées au domaine d'application. Certains concepts ont d'ailleurs été considérés à ce point universels qu'ils ont été intégrés aux modèles de données. Tel est le cas des relations *is-a* par exemple ou des types d'associations d'*agrégation* et de *composition* en UML. On étudiera ci-après le cas des types d'associations génériques. Celui des types d'entités génériques est étudié dans l'annexe de ce chapitre.

16.14.1 Type d'associations de composition

Un type d'associations de composition exprime le fait qu'une entité *est composée* d'autres entités ou est constituée d'un *agrégat* de ces entités. On parle aussi d'un type d'associations *tout-partie* (*whole-part*). Les diagrammes de classes UML offrent deux variantes de types d'associations pour exprimer ce concept : l'*agrégation* et la *composition* (voir section 17.11). La première exprime une composition faible (dite *agrégation*), qui conserve aux composants leur indépendance alors que la seconde définit la composition forte (dite *composition*) dans laquelle un composant n'existe qu'en tant que tel. Le concept de composition est cependant très riche et assurément plus complexe que ce que les modèles actuels en présentent. On lira par exemple [Henderson, 1999] et [Barbier, 2003] pour une étude plus détaillée.

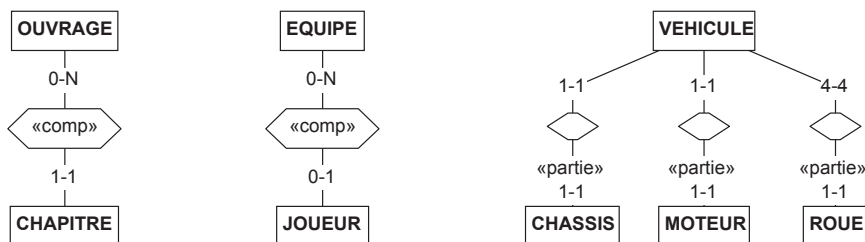


Figure 16.39 - Exemples de types d'associations de composition

La figure 16.39 exprime qu'un ouvrage est composé de chapitres, qu'une équipe est composée de joueurs et qu'un véhicule est composé d'un châssis, d'un moteur et de quatre roues. Dans le premier cas, un chapitre n'existe que comme composant d'un ouvrage mais, en revanche, un joueur peut exister sans appartenir à une équipe. Nous avons utilisé le stéréotype «comp» pour exprimer la composition, ou encore (schéma de droite) le stéréotype «partie» pour indiquer les parties d'un assemblage en cas d'ambiguïté (par exemple, des pièces mécaniques composées d'autres pièces mécaniques). Il est à noter que la sémantique externe du schéma de droite est loin d'être évidente. L'exercice A16. abordera ce point.

Remarque

L'expression d'une composition par un type d'associations de composition n'est possible que pour représenter des structures simples. Le schéma de droite de la figure 16.35, par exemple, représente une structure de composition plus complexe qui nécessite un type d'entités et deux types d'associations. L'utilisation du stéréotype «comp» n'est guère envisageable (voir exercice A16.21).

16.14.2 Type d'associations de matérialisation

Ce type d'associations binaire connecte un type d'entité B à un type d'entités A de sorte que B apparaît comme une *matérialisation*, une *réalisation*, une *concrétisation*, une *incarnation* ou un *avatar* de A, lequel à l'inverse apparaît comme une *abstraction*, un *modèle* ou une *classe générale* de B. Dans la figure 16.40, l'*exemplaire* (concept concret) est décrit comme une matérialisation d'un *ouvrage* (concept abstrait), tout comme un *cours* est une réalisation concrète, durant une certaine année, d'un *cours type*. On propose de représenter un type d'associations de matérialisation par un simple type d'associations, avec ou sans nom, mais caractérisé par le stéréotype «de». Ce type d'associations est le plus souvent *un-à-plusieurs*, de sorte qu'il n'est pas nécessaire de distinguer explicitement le types d'entités abstrait du type d'entités concret.

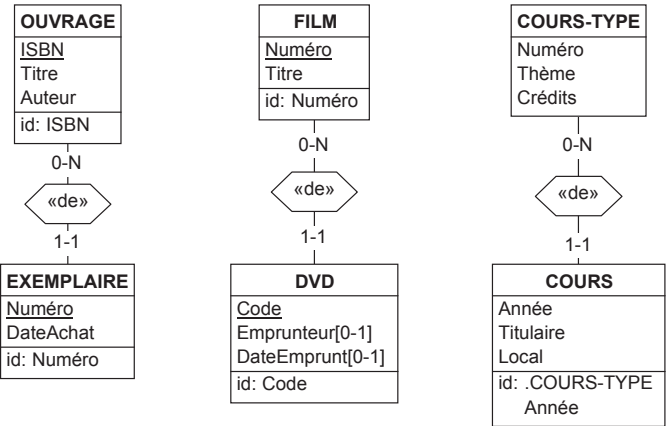


Figure 16.40 - Exemples de types d'associations de matérialisation

Très fréquent et en apparence anodin, ce type d'associations possède des propriétés remarquables qui le rendent relativement inclassable, d'où la proposition d'en faire un objet générique.

1. Il se comporte comme une relation *is-a* puisqu'il s'accompagne d'un mécanisme d'héritage : le DVD hérite du numéro et du titre de son film. Il s'en distingue cependant par le fait que plusieurs entités concrètes peuvent être associées à une même entité abstraite.
2. Il ressemble à une relation d'*instantiation*, qui associe une entité à son type (l'entité identifiée par "C400" est une instance de CLIENT). Le DVD de code "F-7831007" est une instance du film numéro "F-78310". Malheureusement, une telle relation n'existe pas dans le modèle Entité-association : on ne peut représenter simultanément un type et ses instances dans le même schéma, d'autant plus que ces instances sont également des types d'autres instances !
3. En faire un simple type d'associations reviendrait à ignorer ses propriétés et donc à appauvrir la puissance d'expression du modèle.

Le schéma de la figure 16.41, représentant des vols commerciaux, est une application intéressante combinant des types d'associations de composition et de matérialisation. Le type d'entités LIGNE constitue un modèle abstrait pour lequel on enregistre les différentes matérialisations VOL à différentes dates. Une LIGNE est composée d'une suite de SECTIONS entre deux aéroports. Le VOL qui matérialise une LIGNE est lui-même composé de matérialisations (SECTION VOL) des SECTIONS qui composent cette LIGNE. On retrouve des variantes de ce fragment dans de nombreux schémas. Pour être complet, le schéma devrait inclure une contrainte cyclique exprimant le fait que *la ligne du vol d'une section de vol est la ligne de la section de cette section de vol*.

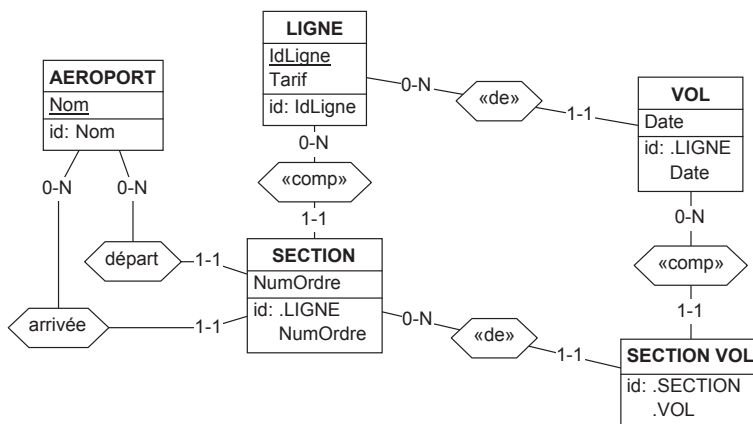
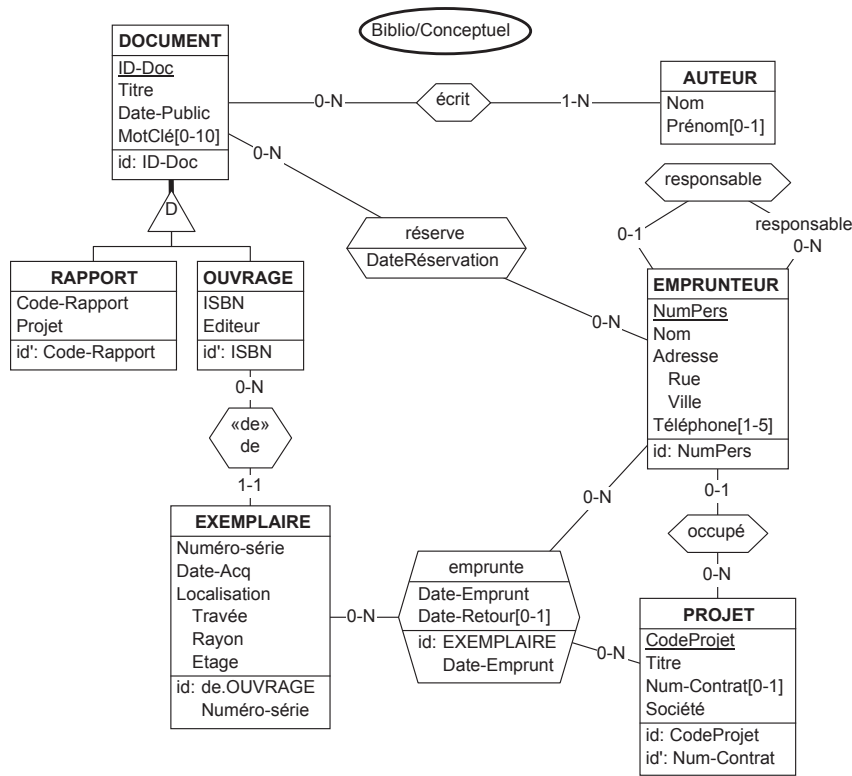


Figure 16.41 - Composition et matérialisation

16.15 EXEMPLE DE SCHÉMA ENTITÉ-ASSOCIATION

Le schéma de la figure 16.42 reprend certaines constructions spécifiques au modèle Entité-association étendu. Son interprétation et sa critique sont laissées à l’initiative du lecteur. Que penser par exemple de la contrainte additionnelle ? Quelle est son intention ? Celle-ci est-elle correctement traduite ? Un emprunteur peut-il emprunter pour le compte d’un projet sur lequel il n’est pas occupé ? Un emprunteur peut-il réserver plus d’une fois un même document ? De quelles informations retient-on l’historique ? Est-ce cohérent ?



$(de * emprunte)[DOCUMENT, EMPRUNTEUR] \cap réserve[DOCUMENT, EMPRUNTEUR] = \emptyset$

Figure 16.42 - Un exemple de schéma Entité-association étendu

16.16 MÉTADONNÉES ET MÉTASCHÉMAS

Les schémas que nous avons considérés jusqu’ici, qu’ils soient relationnels, SQL, Entité-association (et au chapitre suivant, UML) sont des constructions complexes que nous aurons à manipuler, à l’occasion, de manière automatique. Il sera donc nécessaire de les représenter dans un format approprié à ce type de manipulation. Cette représentation se fera sous la forme de données obéissant à un schéma particu-

lier. C’est ce phénomène de représentation de schémas par des données que nous étudions dans cette section. Le concept étant relativement délicat, nous l’aborderons par le biais d’un exemple connu : le catalogue d’une base de données SQL.

16.16.1 Une première approche : le catalogue d’une base de données

Jetons un coup d’œil en arrière pour réanalyser les figures 9.4 et 9.5. Elle représentent quatre tables du catalogue d’une base de données SQL. Ces tables ont un schéma et un contenu. Ce dernier est constitué de données un peu particulières, puisqu’elles décrivent le schéma d’une base de données. Ces tables sont donc relatives à un domaine d’application qui n’est pas une partie du monde réel au sens strict, mais un monde virtuel peuplé de schémas, constitués de tables, de colonnes et de clés. Puisque ces données décrivent des données, on les appelle **métadonnées**²⁷.

Le schéma de ces tables constitue alors un **métaschéma**. En y regardant de plus près, on fait une observation plutôt surprenante : les tables SYS_TABLE et SYS_COLUMN contiennent aussi des lignes qui décrivent les objets du métaschéma lui-même. Ces données, qui décrivent des métadonnées, sont donc des **méta-méta-données**, et leur schéma est par conséquent un **méta-métaschéma** !²⁸

Il nous reste une observation à formuler : les métadonnées et les méta-métadonnées ont été stockées dans une base de données relationnelle. Les données, les métadonnées et les méta-métadonnées sont donc structurées selon le même modèle, le modèle SQL. A priori, cette unicité de modèles n’est pas obligatoire. On aurait pu implémenter les métadonnées dans un autre modèle, jouant le rôle de **métamodèle** (tel est le cas de XML et les DTD). De même, les méta-métadonnées auraient pu être représentées selon un méta-métamodèle différent du modèle SQL. Cette unicité des modèles résulte d’un choix de la part des concepteurs des SGBD. Elle permet en effet, comme nous l’avons montré au chapitre 9, d’utiliser toute la puissance du SGBD pour la consultation des données des trois niveaux.

Le tableau ci-dessous synthétise cette organisation. Il identifie, pour chacun des trois niveaux, les *instances*, le *schéma* et le *modèle* dans lequel celui-ci est exprimé. La cinquième colonne précise le modèle adopté dans les bases de données SQL.

niveau	instances	schema	modèle	BD SQL
1	données	schéma	modèle	SQL
2	schéma = métadonnées	métaschéma	métamodèle	SQL

27. Le préfixe méta, emprunté au vocabulaire de la philosophie, est d’usage courant en informatique. Pour faire court, associé au concept C, le terme méta-C désigne l’application de C à lui-même. Par exemple, un *métalangage* est un langage qui permet de définir des langages (notation BNF par exemple). Un *métacompileur* (ou compilateur de compilateur) est un compilateur qui traduit un programme rédigé dans un métalangage en un compilateur.

28. La chaîne s’arrête ici, non pas de manière naturelle, mais parce que les concepteurs de SGBD ont pris la décision de fusionner métadonnées et méta-métadonnées et donc d’adopter le métaschéma comme méta-métaschéma. On retrouve le même principe d’unicité des modèles en XML, dont les *XML Schemas*, sont des métadonnées exprimées en XML (contrairement aux DTD).

3	métaschéma = méta-métadonnées	méta-métaschéma	méta-métamodèle	SQL

16.16.2 Métaschéma du modèle Entité-association

Pour aborder cette question, nous choisirons de représenter les objets qui constituent les schémas conceptuels exprimés dans le modèle Entité-association de base. Cette représentation constitue donc un métaschéma de ce modèle.

La figure 16.43 montre un métaschéma²⁹ du modèle présenté et utilisé au chapitre 12. Il reprend les concepts de *schéma*, de *type d'entités*, de *type d'associations*, d'*attribut*, de *rôle* et d'*identifiant*. Chacun de ces concepts est représenté par un type d'entités. Les types d'associations expriment qu'un schéma est composé de types d'entités et de types d'associations, qu'un type d'associations est constitué de deux rôles, qu'un rôle³⁰ est joué par un type d'entités, que des attributs et des identifiants peuvent être associés à chaque type d'entités et qu'un identifiant est composé de rôles et/ou d'attributs.

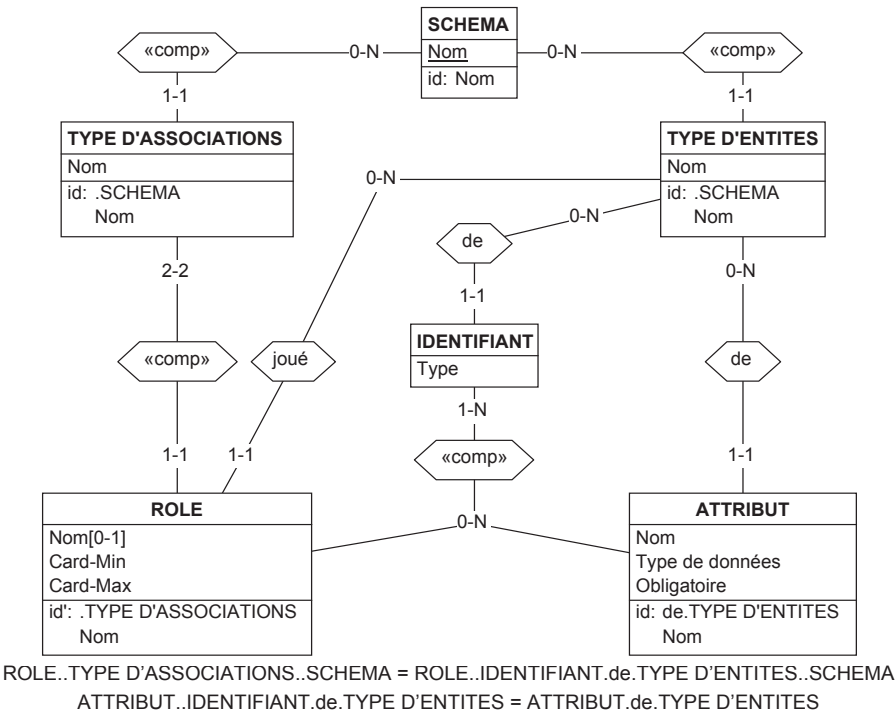


Figure 16.43 - Un métaschéma partiel du modèle Entité-association de base

On ajoute deux contraintes cycliques précisant que (1) les composants rôles d'un identifiant d'un type d'entités E appartiennent à un type d'associations du schéma de

29. ... UN métaschéma : pourquoi pas LE métaschéma du modèle de base ?

30. L'identifiant de ROLE n'est pas entièrement satisfaisant. Voir exercice A16.29.

E et que (2) les composants attributs d'un identifiant d'un type d'entités E sont des attributs de E. Il reste d'autres contraintes, cycliques ou non, dont l'expression est confiée au lecteur.

On observera que ce schéma utilise des constructions du modèle étendu. Le modèle de base est en effet trop pauvre pour exprimer avec précision ses propres propriétés. La construction des schémas des métamodèles du modèle SQL2, du modèle SQL3, du modèle relationnel et du modèle Entité-association étendu est laissée, elle aussi, à l'initiative du lecteur à titre d'exercice.

Le lecteur intéressé par la problématique des métadonnées trouvera dans [Hay, 2006] une analyse approfondie des concepts relatifs aux systèmes d'information d'entreprise et une collection de métaschémas qui les décrivent.

16.17 SÉMANTIQUE INTERNE DU MODÈLE ENTITÉ-ASSOCIATION

Nous avons évoqué à la section 16.3 deux interprétations du concept de *sémantique* d'un modèle et de *sémantique* d'un schéma. Dans cette section, nous évoquons une **sémantique interne** pour le modèle Entité-association basée sur le modèle relationnel du chapitre 3. L'élaboration d'une sémantique complète pour un modèle aussi riche serait une tâche complexe sans intérêt dans cet ouvrage. Nous limiterons donc sa définition aux éléments importants du modèle, et en particulier aux contraintes d'intégrité. Nous établirons cette sémantique sous forme d'une correspondance entre les concepts du modèle Entité-association et une variante du modèle relationnel³¹ à partir d'exemples représentatifs. On en trouvera une description plus détaillée à l'annexe A16.

16.17.1 Types d'entités et relations *is-a*

On admet l'existence a priori de l'ensemble **entités** comprenant toutes les entités passées, présentes, futures ou simplement possibles. On définira le type d'entités SALARIE comme un sous-ensemble d'**entités** :

$SALARIE \subseteq \text{entités}$

Afin d'éviter un excès de symboles mathématiques, on écrira plus simplement :

SALARIE : entités

Une relations *is-a* s'exprime de la même manière :

EMPLOYE : SALARIE

16.17.2 Types d'associations

Un type d'associations correspond une relation (cf. schémas 16.11 et 16.8) :

spécifie(COMMANDE, PRODUIT, FOURNISSEUR)

offre(FOURNISSEUR, PRODUIT)

31. Cette section s'inspire de [Hainaut, 1989] et [Hainaut, 2006].

vendu(vendeur: CLIENT, acheteur: CLIENT, IMMEUBLE)

16.17.3 Attributs

Les attributs d'un type d'entités E sont décrits par une relation spéciale, dite *relation-entité*, qu'on nommera « descr-E » (*description de E*), et qui associe au type d'entités la liste de ses attributs (cf. schéma 16.16) :

descr-SALARIE(**SALARIE**, Matricule, Nom)

L'attribut représentant le type d'entités est l'identifiant primaire de cette relation mais celle-ci peut faire l'objet d'autres identifiants :

descr-SALARIE(SALARIE, Matricule, Nom)

Les attributs d'un type d'associations s'ajoutent à la relation du type d'associations (schéma 16.10) :

réserve(DOCUMENT, EMPRUNTEUR, Date Réservation)

emprunte(EXEMPLAIRE, EMPRUNTEUR, PROJET, Date Emprunt, Date Retour)

16.17.4 Attributs et rôles obligatoires

On déclare les attributs Matricule et Nom *obligatoires* pour SALARIE, par la contrainte selon laquelle toute entité SALARIE apparaît dans descr-SALARIE :

descr-SALARIE[SALARIE] = SALARIE

Un rôle obligatoire se décrit de manière analogue :

appartient(VEHICULE, CLIENT)

appartient[VEHICULE] = VEHICULE

16.17.5 Identifiants

Les *identifiants locaux* sont traduits par des identifiants relationnels, tant pour les types d'entités que pour les types d'associations. Les exemples suivants sont tirés de schémas des chapitres 12 et 16 :

descr-CLIENT(CLIENT, NumClient, Nom, Adresse)

desc-VOL(VOL, Ligne, Date, NumAppareil)

desc-PROJET(PROJET, CodeProjet, Titre, Budget)

appartient(VEHICULE, CLIENT)

implique(VEHICULE, ACCIDENT)

offre(PRODUIT, FOURNISSEUR, Prix, Délai)

reserve(DOCUMENT, EMPRUNTEUR, Date Réservation)

emprunte(EMPRUNTEUR, PROJET, EXEMPLAIRE, Date Emprunt, Date Retour)

La représentation des *identifiants hybrides* est moins immédiate et nécessite un traitement spécifique qui développé à l'annexe A16.

16.17.6 Autres contraintes

L'expression de la plupart des contraintes rencontrées dans ce chapitre ne pose pas de problèmes particuliers. Traitons l'exemple des dépendances fonctionnelles, à partir du schéma de la figure 16.34 :

ETUDIANT, MATIERE, PROFESSEUR: entités
 descr-PROFESSEUR(PROFESSEUR, Matricule, Nom, Ecole, Directeur)
 inscrit(ETUDIANT, MATIERE, PROFESSEUR)
 descr-PROFESSEUR: Ecole $\longrightarrow$ Directeur
 inscrit: PROFESSEUR $\longrightarrow$ MATIERE

Selon les principes de la normalisation étudiés au chapitre 3, il apparaît que les deux relations présentent des anomalies de redondance : aucun des deux déterminants n'est un identifiant de sa relation.

16.17.7 Application de la théorie relationnelle au modèle Entité-association

L'expression relationnelle du modèle Entité-association permet de donner à chacun des objets et contraintes une signification mathématique précise incontestable. Elle conduit aussi à des applications particulièrement utiles.

a) Calcul des identifiants d'un type d'associations

Le type d'associations inscrit rappelé ci-dessus peut se réécrire :

inscrit(ETUDIANT, MATIERE, PROFESSEUR)
 inscrit: ETUDIANT, MATIERE $\longrightarrow$ PROFESSEUR
 inscrit: PROFESSEUR $\longrightarrow$ MATIERE

On en déduit, selon la procédure Proc2 de la section 3.6.3, les deux identifiants {ETUDIANT, MATIERE} et {ETUDIANT, PROFESSEUR}.

b) Décomposition d'un type d'associations

Le schéma du type 1:N:N de la figure 16.29 possède un rôle de cardinalité [1-1]. Son expression relationnelle est très simple : $R(\underline{A}, B, C)$. L'existence des DF $A \longrightarrow B, C$ nous autorise à décomposer R en $R1(\underline{A}, B)$ et $R2(\underline{A}, C)$, avec $R1[A] = R2[A]$. Ce qui, réexprimé dans le modèle Entité-associations, donne le schéma 16.44.



Figure 16.44 - Normalisation d'un type d'associations

c) Normalisation d'un type d'associations

Le type d'associations inscrit (dont on n'arrive décidément pas à se débarrasser !) se traduit en une relation ternaire largement discutée au chapitre 3, et dont on sait qu'il existe trois transformations possibles dénommées métaphoriquement *la peste*, *le choléra* et *la peste ET le choléra*. L'exercice 16.8 leur est consacré.

16.18 QUE RETENIR ?

Le modèle Entité-association étendu comporte des extensions offrant une plus grande expressivité et une meilleure précision dans la description des concepts d'un domaine d'application.

Un attribut peut être **multivalué** et/ou **composé**. Il peut être attaché à un type d'entités ou à un type d'associations.

Un type d'associations peut comporter plus de deux rôles (on parle d'un **type d'associations n-aire**) et posséder des attributs tout comme un type d'entités. Un rôle peut être défini sur l'union de plusieurs types d'entités (rôle **polymorphique**).

On admet désormais qu'une entité puisse appartenir à plus d'un type. On définit ainsi entre les types d'entités une structure hiérarchique basée sur la **relation is-a**, selon laquelle B is-a A indique que B est un sous-type de A, c'est-à-dire que les entités du type B sont aussi des entités du type A. Le principe d'**héritage** stipule que le type d'entités B possède, outre ses caractéristiques propres (attributs, rôles, contraintes), toutes celles du type A.

Tout comme les types d'entités, les attributs complexes (simultanément multivalués et composés) et les types d'associations peuvent être dotés d'**identifiants**.

Le jeu de contraintes d'intégrité est étendu. On y compte notamment les contraintes de **domaine**, les contraintes **entre valeurs** d'attribut, les contraintes d'**existence** et les **dépendances fonctionnelles**. Les contraintes sont trop nombreuses pour qu'on puisse leur affecter une représentation graphique spécifique. D'où la nécessité d'un **langage général d'expression de contraintes**.

La mise en évidence d'**objets génériques** (types d'associations de **composition** et de **matérialisation**) permet de mieux modéliser le domaine d'application.

Le modèle Entité-association reçoit une **sémantique mathématique** qui permet une définition précise de ses composants et de leurs propriétés selon le modèle relationnel. Tous les résultats développés dans ce dernier modèle sont ainsi applicables aux schémas Entité-associations.

16.19 POUR EN SAVOIR PLUS

Le modèle Entité-associations a fait l'objet d'innombrables propositions, extensions, interprétations et suggestions d'usage, publiées dans plusieurs milliers d'ouvrages et d'articles de journaux et de conférences depuis près de 40 ans. La majorité des contributions ont été publiées dans les actes des conférences internationales *Entity-relationship* (renommées *International Conference on Conceptual Modeling*) et *Conference on Advanced Information System Engineering* (CAiSE)

ainsi que dans les journaux *Journal of Data Semantics* (Springer) et *Data and Knowledge Engineering* (Elsevier). On peut considérer [Deheneffe, 1974] et [Chen, 1976] comme les premières descriptions de ce modèle respectivement dans leurs versions européenne et américaine³².

Le *Modèle Individuel* ou *Entité-relation* de la méthode Merise est l'une des variantes les plus populaires du modèle Entité-association. Développé dans les années soixante-dix à l'initiative d'Hubert Tardieu, il s'inspire du modèle de Bachman [Bachman, 1969], des travaux de J.-R. Abrial sur le SGBD Socrate et d'une première formalisation des modèles binaires [Abrial, 1974]. Largement répandue dans les entreprises et les administrations françaises dans les années quatre-vingts et quatre-vingt-dix, la méthode Merise tend à être supplantée par la modélisation UML, malgré l'immaturité et les faiblesses de celle-ci. La dernière version de Merise est décrite dans l'ouvrage [Tardieu, 2000] et plus récemment, dans [Baptiste, 2012].³³ La méthode IDA [Bodart, 1993], de l'université de Namur, est proche de la méthode Merise, dont elle partage l'origine. Le modèle présenté dans ce chapitre est l'héritier des modèles Entité-association de Merise et de IDA.

ORM, *Object-role model* [Halpin, 2008], constitue un formalisme d'expression de schémas conceptuels proche de leur interprétation linguistique. L'accent est mis plus sur le rôle que jouent les types d'entités que sur les types d'associations eux-mêmes. Il est pris en charge depuis les années quatre-vingts par une communauté scientifique et industrielle très active. Un journal électronique lui est consacré³⁴.

Les diagrammes de classes d'UML dérivent du modèle de Chen [Chen, 1976] mais seules les constructions utiles pour la programmation et les architectures orientées objet ont été reprises. Comme on le verra au chapitre 17, ces diagrammes sont mal adaptés à la modélisation conceptuelle et logique des bases de données.

Le modèle Entité-association a fait l'objet d'extensions vers des domaines spécifiques tels que les bases de données temporelles [Theodoulidis, 1991], [Detienne, 2001], [Parent, 2006], les bases de données spatiales [Parent, 2006], les bases de données actives [Ceri, 1997] et les entrepôts de données [Vaisman, 2014].

L'origine des relations *is-a* décrites dans ce chapitre est très ancienne puisqu'on trouve les notions de classe, d'héritage et de polymorphisme dans le premier langage orienté objet *Simula 67* (1967). Leur utilisation comme mécanisme d'abstraction dans la modélisation des données a été présentée dans [Smith, 1977]. On trouvera dans [Brachman, 1983] une étude de cette structure dans les domaines de l'intelligence artificielle, de la programmation et des bases de données.

[Goldstein, 1993] et [Pirotte, 1994] offrent une présentation détaillée des types d'associations de matérialisation et de leurs propriétés.

Diverses sémantiques du modèle Entité-association ont été proposées, notamment, parmi les plus récentes, dans [Talheim, 2000], [Hainaut, 2006], [Gogolla,

32. Ces versions se distinguent essentiellement, outre par le graphisme, par l'interprétation des contraintes de cardinalité des rôles : conforme au présent ouvrage pour les versions européennes et à la façon d'UML pour les versions américaines.

33. On recommandera une visite du site <http://merise.developpez.com/faq/?page=GENE>, notamment pour l'historique relativement complet de la méthode par un de ses auteurs.

34. <http://www.orm.net> (janvier 2015)

1991], et [Doberkat, 2004]. La définition d'une sémantique pour les différents types de diagrammes UML est une thématique largement explorée aujourd'hui sans qu'un consensus se dégage de manière claire. On en trouvera une tentative au chapitre 17.

Les notions de sémantique et de syntaxe dans les langages de programmation et dans les langages et formalismes de modélisation sont souvent mal comprises. L'une des meilleures références sur la question est incontestablement [Harel, 2004].

Signalons enfin une initiative intéressante de Microsoft, ADO.NET, qui offre au programmeur .NET une interface Entité-association, y compris un langage de requête approprié au modèle, à des bases de données relationnelles, sans qu'il soit nécessaire de traduire les requêtes en SQL³⁵.

16.20 CRÉER UN SCHÉMA CONCEPTUEL À L'AIDE D'UN OUTIL (COMPLÉMENTS)

Un tutoriel relatif aux concepts décrits dans ce chapitre est disponible sur le site de l'ouvrage. Ce tutoriel décrit les manipulations suivantes dans l'atelier DB-MAIN:

- définition et utilisation de stéréotypes ;
- création d'attributs composés et multivalués ;
- création et utilisation de types définis par l'utilisateur ;
- création de types d'associations n-aires, de rôles multitypes d'entités et d'attributs ;
- création de hiérarchies *is-a* ;
- création d'identifiants de types d'associations et d'attributs ;
- création de contraintes ;
- création de méthodes.

16.21 EXERCICES

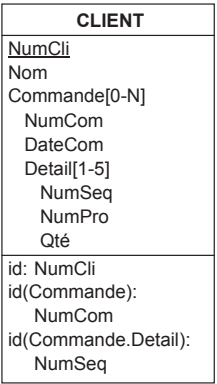
- 16.1 Le type d'associations cyclique vendu de la figure 16.8 n'est pas satisfaisant, car il impose certaines contraintes aux possibilités de vente. On suggère d'ajouter un attribut *Date Vente*. Compléter le schéma en introduisant cet attribut et en définissant l'identifiant qui semble le plus approprié.
- 16.2 On considère le schéma ci-dessous, exprimant qu'un auteur apparaît dans une certaine position dans la liste des auteurs d'un ouvrage. Compléter ce schéma de manière à rendre explicite les deux règles métier : *un auteur n'apparaît qu'une seule fois pour un ouvrage*, et *il n'y a qu'un seul auteur à une position déterminée pour un ouvrage*. Ce schéma est-il normalisé ?

35. [http://msdn2.microsoft.com/en-us/library/aa730866\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/aa730866(VS.80).aspx) (janvier 2015)



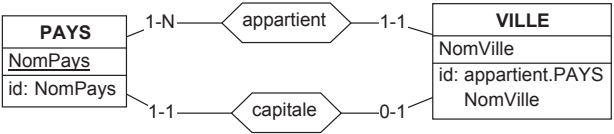
16.3 Proposer deux formes équivalentes du rôle polymorphique de la figure 16.9.

16.4



Un attribut de structure trop complexe peut être l'indice d'une mauvaise modélisation. Montrer que l'exemple ci-contre en est une illustration. Proposer une variante équivalente plus appropriée.

16.5 Compléter le schéma ci-dessous des contraintes qui paraissent évidentes.



16.6 Compléter le schéma de la figure 16.45 en calculant les grandeurs inconnues, notées « ? ».

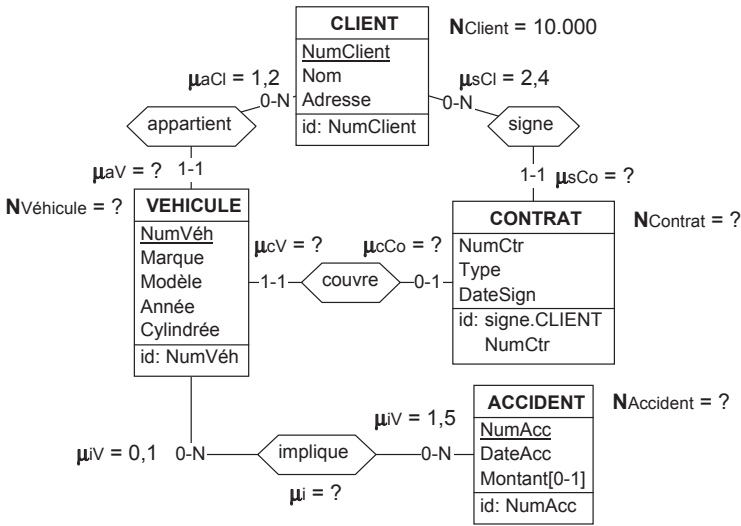


Figure 16.45 - Calcul des tailles des populations

- 16.7 Indiquer pour chaque type d'entités du schéma 16.46 les attributs pour lesquels on doit fournir une valeur de manière à identifier une entité de ce type. Par exemple, pour identifier une entité CLIENT, il faut fournir une valeur de CLIENT.NumCli, tandis que pour une entité FACTURE, il faut fournir une valeur de de.CLIENT.NumCli et une valeur de FACTURE.NumFact³⁶.

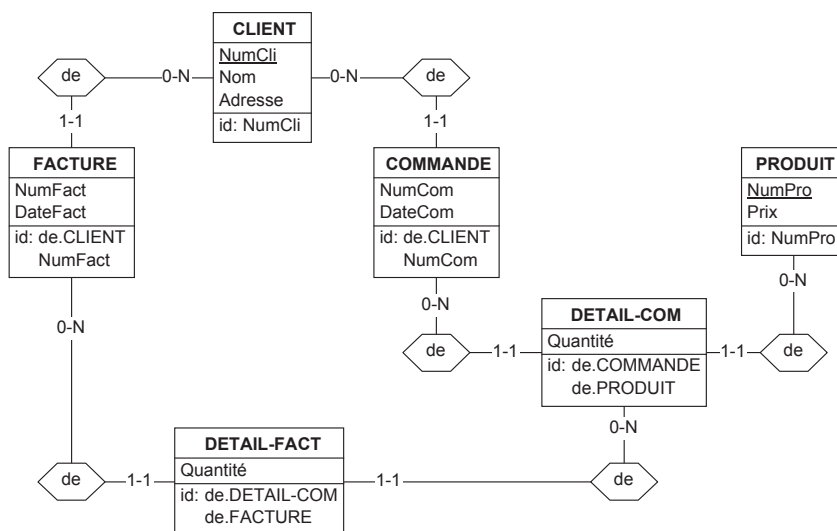


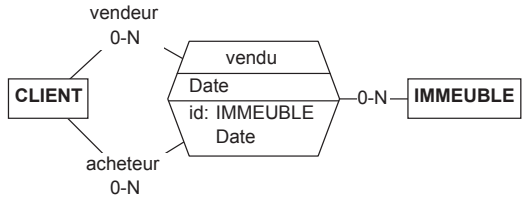
Figure 16.46 - Identification de types d'entités

- 16.8 On a montré dans la section 16.17.7, point (c), que le type d'associations inscrit de la figure 16.34 n'était pas normalisé et qu'il pouvait se transformer en trois schémas équivalents. Dessiner ces trois schémas dans le modèle Entité-association en indiquant soigneusement toutes les contraintes d'intégrité. Quelle est le schéma qui semble le plus naturel ?

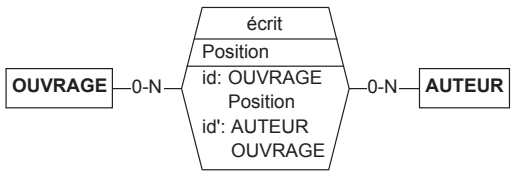
Exercices corrigés

- **Exercice 16.1.** Ce schéma interdit à un vendeur de vendre plus d'une fois un même immeuble à un même acheteur. Situation rare mais possible ! En introduisant la *date de vente*, et en admettant qu'un immeuble ne puisse être vendu deux fois le même jour, il vient :

36. Attention, ceci ne veut pas dire que le groupe {NumCli, NumFact} constitue un identifiant de FACTURE au sens du modèle Entité-association.



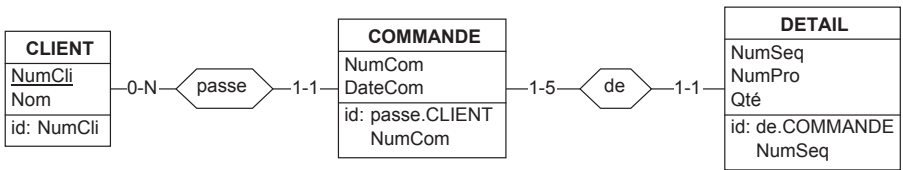
- **Exercice 16.2.** Cette question a été abordée dans l'exercice 3.5, où on a conclu que le schéma était normalisé.



• **Exercice 16.3**



- **Exercice 16.4.** L'attribut complexe **Commande** est extrait sous la forme du type d'entités **COMMANDE**. L'attribut complexe **Détail** de ce dernier est traité de la même manière. On obtient dès lors le schéma ci-dessous :



- **Exercice 16.5.** Si on admet que la capitale d'un pays appartient à ce dernier, alors il faut imposer la contrainte $\text{capitale} \subseteq \text{appartient}$.
- **Exercice 16.6.** En exploitant les relations de la section 16.8.6, il vient :

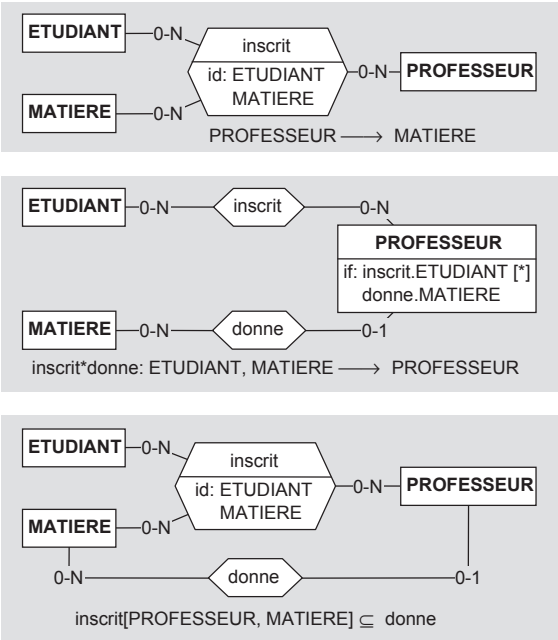
appartient :	$\mu_{aV} = 1$	ACCIDENT :	$N_{\text{Accident}} = 800$
signe :	$\mu_{sCo} = 1$	couvre :	$\mu_{cV} = 1$
VEHICULE :	$N_{\text{Véhicule}} = 12.000$	couvre :	$\mu_{cCo} = 0,5$
CONTRAT :	$N_{\text{Contrat}} = 24.000$	implique :	$\mu_i = 1.200$

• Exercice 16.7

Pour identifier ...	il faut connaître les valeurs de ...
CLIENT	CLIENT.NumCli
FACTURE	de.CLIENT.NumCli, FACTURE.NumFact
COMMANDE	de.CLIENT.NumCli, COMMANDE.NumCom
PRODUIT	PRODUIT.NumPro
DETAIL-COM	de.COMMANDE.de.CLIENT.NumCli, de.COMMANDE.NumCom, de.PRODUIT.NumPro
DETAIL-FACT	de.FACTURE.de.CLIENT.NumCli, de.FACTURE.NumFact, de.DETAIL-COM.de.COMMANDE.de.CLIENT.NumCli, de.DETAIL-COM.de.COMMANDE.NumCom, de.DETAIL-COM.de.PRODUIT.NumPro

Remarque : l’identification de DETAIL-FACT nécessite la connaissance de deux valeurs de NumCli. Sont-elles identiques ?

- **Exercice 16.8.** En s’inspirant des solutions obtenues à la fin de la section 3.8.5, on obtient :



Des compléments et des exercices supplémentaires sont disponibles à l’annexe A16.

Chapitre 17

Les diagrammes de classes UML

UML est souvent proposé comme formalisme universel de spécification de systèmes d'information. Dans le domaine de la modélisation des bases de données, majoritairement basée sur l'une ou l'autre variante du modèle Entité-association, le diagramme de classes d'UML est parfois utilisé comme un langage d'expression de schémas conceptuels, logiques ou physiques de bases de données. L'objectif de ce chapitre est de présenter et d'analyser les principaux concepts des diagrammes de classes d'UML, par comparaison au modèle Entité-association. Nous en dériverons DB-UML, une version qui satisfait aux exigences de la modélisation conceptuelle des données.

17.1 INTRODUCTION

Unified Modeling Language, ou UML, est un ensemble de *notations* destinées à représenter divers aspects d'une application orientée-objet (dite *le système*). Par extension, ces notations peuvent décrire certaines applications non OO et même certains aspects d'un domaine d'application. Pour chaque domaine de modélisation, UML se présente comme un langage, et plus particulièrement une notation, à partir de laquelle plusieurs méthodes peuvent être développées. Les textes de base insistent sur les aspects syntaxiques plus que sur la signification des symboles. La sémantique d'UML est intuitive et informelle, ce qui présente l'avantage d'une grande liberté d'interprétation. L'inconvénient majeur est qu'il n'existe pas de consensus quant à la sémantique exacte (au sens de la *sémantique interne*) de chacun des concepts définis¹. En pratique, la sémantique la plus fiable est celle de l'outil

qu'on utilise. Dans certains cas, on ne peut découvrir l'interprétation d'une construction d'un diagramme qu'à partir du code SQL ou Java généré par l'outil.

Intégrant les concepts des méthodes OMT [Rumbaugh, 1991], OOSE/Objectory [Jacobson, 1992] et OOAD [Booch, 1994], UML permet de représenter graphiquement les aspects du développement d'une application orientée-objet, depuis les scénarios d'utilisation jusqu'à l'implémentation et le déploiement. Les spécifications UML 2 proposent 13 types de diagrammes (ou modèles) regroupés en cinq vues. Chaque vue et chaque type de diagrammes apporte un éclairage particulier sur le système ou sur une phase de la construction du système.

Vue utilisateur	diagramme de cas d'utilisation (<i>use case</i>)
Vue structurelle	diagramme de classes diagramme d'objets diagramme de structures composites diagramme de paquetage
Vue comportementale	diagramme de séquence diagramme d'interaction diagramme d'états (<i>statechart</i>) diagramme d'activités diagramme de communication diagramme temporel
Vue d'implémentation	diagramme de composants
Vue environnementale	diagramme de déploiement

Figure 17.1 - Les types de diagrammes (ou modèles) d'UML 2

Le présent exposé est limité au sous-ensemble des **diagrammes de classes** relatif à la *modélisation conceptuelle des bases de données*. Il suppose acquis les concepts du modèle Entité-association des chapitres 12 et 16. Le **diagramme d'objets**, destiné à représenter des instances de classes, est à comparer au graphisme de la section 12.4.7.

17.2 UML ET LES BASES DE DONNÉES

Conformément au vocabulaire spécifique d'UML, et au risque de perturber le lecteur, nous utiliserons le terme *diagrammes de classes* ou *modèles de classes* pour parler de *schémas conceptuels* et de *métamodèle de classes* pour désigner la variante UML du *modèle Entité-association* (voir figure 16.2).

Historiquement, le métamodèle de classes est un emprunt à OMT [Object Modeling Technique; Rumbaugh, 1991], laquelle s'est inspirée du modèle Entité-association de [Loomis, 1988], lui-même héritier du modèle de Chen [Chen, 1976].

1. Certains définissent UML comme une *syntaxe à la recherche d'une sémantique* [Harel, 2004].

Certaines constructions pourtant essentielles n'ont pas été reprises, tels les identifiants. Les raisons sont historiques et culturelles. OMT étant prioritairement destiné à la description abstraite et au développement d'applications orientées objet, ce sont avant tout les constructions directement exprimables dans les langages OO qui ont été développées. En outre, les données persistantes étaient abordées dans une perspective orientée objet, laquelle stipule que les objets sont dotés d'un identifiant intrinsèque (oid). Nul besoin donc, pensait-on, d'identifiants définis par l'utilisateur. Ceux-ci relevaient de contraintes d'intégrité, concept étranger au monde de la programmation. Cette vision des données a été reprise et amplifiée par UML.

Malgré les lacunes et les défauts congénitaux d'UML dans le domaine de la modélisation des bases de données, il apparaît qu'il est possible à peu de frais de tirer du métamodèle de classes un sous-modèle conceptuel tout à fait convenable se situant entre les modèles Entité-association de base et étendu.

La figure 17.2 montre un diagramme de classes élémentaire dans lequel on identifie trois *classes d'objets*, comportant chacune des *attributs*, faisant l'objet d'*opérations* et formant des *associations*. Nous étudierons chacun de ces concepts après une courte description des mécanismes d'extension.

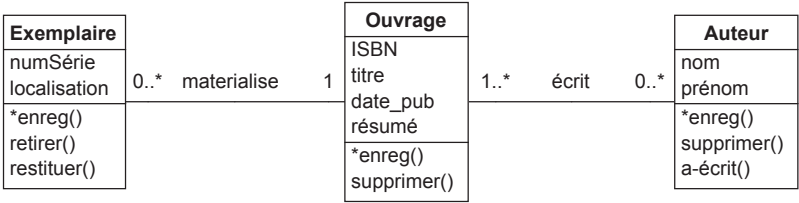


Figure 17.2 - Un diagramme de classes UML

17.3 LES MÉCANISMES D'EXTENSION

Tout métamodèle propose un ensemble cohérent de concepts qui doit satisfaire l'essentiel des besoins prévisibles mais qui présentera tôt ou tard des lacunes pour modéliser certains aspects des domaines d'application auxquels il est destiné. UML propose trois mécanismes d'extension des notations : tags, stéréotypes et profils.

- Un **tag** (ou *tagged value*) définit une étiquette attachée à un objet d'un diagramme et destinée à représenter une propriété de celui-ci. La signification de cette propriété n'est pas précisée dans le métamodèle, mais dépend de l'utilisateur du diagramme. Sa forme générale est : étiquette = valeur, ou simplement étiquette. Par exemple, si l'attribut numClient est une *primary key*, on prefixera son nom de l'étiquette **PK**, comme dans "**PK** numClient"².
- Un **stéréotype** définit une catégorie remarquable dans une classe ou tout autre objet (section 16.5). Ici encore, l'interprétation est laissée à l'utilisateur du méta-

2. Cet exemple n'infirme pas l'inexistence d'identifiants en UML. Le tag illustré est un simple (et insuffisant) palliatif souvent utilisé mais qui reste externe à UML.

modèle. Les conventions UML suggèrent d'associer le nom du stéréotype à celui de l'objet. Par exemple, si la classe `i_compte` est du type *virtuel*, on associera au nom de la classe le mot-clé **virtuel** comme suit : "**«virtuel»** `i_Compte`".

- Un **profil** définit un métamodèle spécialisé, réduit à un sous-ensemble de concepts, augmenté de tags et stéréotypes, destiné à représenter un domaine de modélisation spécialisé. Il existe par exemple un profil adapté à la représentation de schémas relationnels (IBM Rational Software).

17.4 LES CLASSES D'OBJETS

Une **classe** UML correspond à un type d'entités. À une classe peuvent être associées des sous-classes tout comme dans le modèle Entité-association.

Les propriétés de *couverture partielle et totale* (section 16.9.2) s'appellent en UML respectivement **incomplete** et **complete**. Les propriétés de *disjonction* et de *recouvrement* correspondent aux contraintes UML **disjoint** et **overlapping** (schéma 17.11).

Les sous-classes d'une classe C peuvent être définies par un **discriminateur**, qui est généralement un attribut de C dont la valeur indique à quelle sous-classe chaque entité C appartient (voir section 16.9.8). Une classe *Personne* qui admettrait les deux sous-classes *Femme* et *Homme* se verrait associer le discriminateur *sexe* (un attribut de *Personne*) dont la valeur "F" forcerait l'entité *Personne* à appartenir à la sous-classe *Femme* et la valeur "H" à la sous-classe *Homme*.

17.5 LES ASSOCIATIONS BINAIRES

Une **association** UML correspond à un type d'associations du modèle Entité-association. Elle peut être nommée. Une instance d'association s'appelle un **lien** (*link*). UML distingue les associations binaires et les associations n-aires.

Une *association binaire* possède deux extrémités (rôles), éventuellement nommées, caractérisées par leur multiplicité. Les conventions de dénotation des multiplicités sont différentes de celles des cardinalités³ : min-max se note min..max, N se note *, k-k peut se noter k..k ou k et 0-N peut s'écrire 0..* ou simplement *. Il faut être attentif au fait que la multiplicité UML attachée à un rôle est en fait **une propriété du rôle opposé**, et se note donc sur l'extrémité opposée à celle à laquelle elle s'applique. Dans le cas de la figure 17.2, la multiplicité 1 (soit 1..1) de l'extrémité *Ouvrage* de matérialise signifie qu'à partir d'un objet *Exemplaire*, on voit un et un seul objet *Ouvrage*. On retient donc que la position *des multiplicités UML est inversée par rapport aux cardinalités Entité-association*.

3. En toute généralité, la multiplicité prend la forme d'une suite d'entiers définie par une liste d'entiers et/ou d'intervalles. On se limitera ici à la version communément utilisée : un intervalle.

Le nom d'une association peut être accompagné d'un triangle qui précise le sens de lecture du schéma ("*un client passe ▶ des commandes*"). Cette convention a été décrite dans la figure 16.38/c.

17.6 LES ASSOCIATIONS N-AIRES

Une *association n-aire* possède plus de deux rôles. Elle se représente par un losange auquel un nom peut être attaché. Un rôle d'une association n-aire peut être caractérisé par une multiplicité, mais l'interprétation de ce concept et son intérêt sont sujet à caution comme nous allons le voir. Le schéma de la figure 17.3 indique que *des usines exportent des produits vers des pays*.

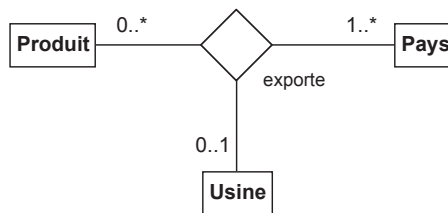


Figure 17.3 - Une association ternaire

La multiplicité d'un rôle joué par la classe C indique combien d'objets C (au minimum et au maximum) sont vus à partir de toute combinaison de n-1 objets correspondant aux n-1 autres rôles. Ainsi, les multiplicités des rôles de l'association *exporte* s'interprètent comme suit :

- *exporte.Produit* [0..*] : étant donné un Pays p et une Usine u, on voit à partir du couple (p,u) de 0 à * (= infini) instances de Produit à travers *exporte* ;
- *exporte.Pays* [1..*] : étant donné un Produit r et une Usine u, on voit à partir du couple (r,u) de 1 à * (= infini) instances de Pays à travers *exporte* ;
- *exporte.Usine* [0..1] : étant donné un Produit r et un Pays p, on voit à partir du couple (r,p) de 0 à 1 instance de Usine à travers *exporte*.

Pour un rôle d'une association binaire, la multiplicité représente une propriété du *rôle d'en face*. La logique est la même pour les associations n-aires, à la différence qu'ici il y a *plusieurs rôles en face*. La multiplicité impose donc une contrainte non pas sur la classe sur laquelle elle est définie mais sur les combinaisons d'objets qu'on peut former à partir des n-1 classes opposées, ce qui complique considérablement la compréhension du concept de multiplicité. Si cette interprétation peut avoir un intérêt pour la multiplicité maximale, elle s'avère particulièrement ésotérique et fort éloignée de l'intuition pour ce qui concerne la multiplicité minimale.

Pour y voir plus clair, tentons de donner une interprétation plus précise au concept de multiplicité. Étant donné une association R entre les classes A, B et C, la signification communément acceptée de la multiplicité min..max du rôle R.A joué par A est la suivante⁴ :

$$\forall b \in B, \forall c \in C, \min \leq |R(B=b \wedge C=c)[A]| \leq \max$$

Ainsi, on peut expliquer l'association exporte comme suit :

$$\begin{aligned} \forall p \in \text{Pays}, \forall u \in \text{Usine}, \quad 0 \leq |\text{exporte}(\text{Pays}=p \wedge \text{Usine}=u)[\text{Produit}]| &\leq \infty \\ \forall r \in \text{Produit}, \forall u \in \text{Usine}, \quad 1 \leq |\text{exporte}(\text{Produit}=r \wedge \text{Usine}=u)[\text{Pays}]| &\leq \infty \\ \forall r \in \text{Produit}, \forall p \in \text{Pays}, \quad 0 \leq |\text{exporte}(\text{Produit}=r \wedge \text{Pays}=p)[\text{Usine}]| &\leq 1 \end{aligned}$$

17.6.1 Multiplicité maximale (max)

On distinguera généralement deux cas de figure : $\text{max} = 1$ et $\text{max} > 1$ (le plus souvent $\text{max} = *$). Si $\text{max} = 1$, les $n-1$ autres rôles constituent un **identifiant** de l'association. Si $\text{max} > 1$, ces autres rôles ne forment **pas un identifiant**. Le schéma de la figure 17.3 indique donc que $\{\text{Produit}, \text{Pays}\}$ est un identifiant (et d'ailleurs le seul) de l'association exporte. En revanche, une Usine peut exporter dans un Pays plusieurs Produits ($\text{exporte.Produit}[0..*]$) et une Usine peut exporter un Produit dans plusieurs Pays ($\text{exporte.Pays}[1..*]$).

On conclut que, pour les valeurs les plus fréquentes 1 et *, la multiplicité maximale indique l'existence (1) ou l'absence (*) d'un **identifiant** formé des $n-1$ rôles opposés. Cette conclusion vaut également pour les associations binaires. Il est à noter que si plus d'un rôle d'une association n -aire est de multiplicité maximale 1, alors les identifiants induits possèdent des rôles en commun. En revanche, il n'est pas possible de spécifier, comme dans le modèle Entité-association, qu'une instance d'une classe ne peut appartenir à plus d'une seule instance d'association.

17.6.2 Multiplicité minimale (min)

Ici aussi, on distinguera deux cas de figure : $\text{min} = 0$ et $\text{min} > 0$ (le plus souvent $\text{min} = 1$). Si $\text{min} = 0$, il n'existe aucune contrainte sur la présence dans des instances de l'association des combinaisons d'objets des $n-1$ autres classes. Si $\text{min} > 0$, toute combinaison doit apparaître dans au moins une instance de l'association. Le schéma de la figure 17.3 s'interprète comme suit :

- $\text{exporte.Produit}[0..*]$: étant donné une Usine **u** et un Pays **p**, il est possible que **u** n'exporte aucun Produit vers **p** ; conséquences : il est possible que certaines Usines n'exporte rien et qu'on n'exporte rien vers certains Pays ;
- $\text{exporte.Pays}[1..*]$: pour chaque Usine **u** et chaque Produit **r**, il doit exister au moins un Pays vers lequel **u** exporte **r** ; il est cependant possible qu'on n'exporte rien vers certains pays ;
- $\text{exporte.Usine}[0..1]$: étant donné un Produit **r** et un Pays **p**, il est possible qu'aucune Usine n'exporte **r** vers **p** ; conséquences : il est possible qu'on n'exporte rien vers certains pays et que certains produits ne soient pas exportés ;

4. On adopte ici la sémantique interne d'une association similaire à celle d'un type d'associations. On fait donc l'hypothèse qu'une association est représentée par une relation, qui peut être soumise à des opérateurs algébriques de sélection et de projection. L'expression $|A|$ représente la taille de l'ensemble A.

La multiplicité [1..*] du rôle `exporte.Pays` a des effets qui malheureusement ne se limitent pas à cette interprétation.⁵ Leur analyse est développée à l'annexe A17.

Il en ressort que les associations n-aires doivent être évitées. Lorsque de telles structures s'avèrent nécessaires, on recommandera de les traduire sous la forme d'une classe et de n associations binaires, comme illustré à la figure 17.4. On observe que cette forme permet d'exprimer simplement les rôles obligatoires et facultatifs, les cardinalités à l'europpéenne (par exemple, un produit fait l'objet de 1 à 20 exportations) et que les multiplicités $\max = 1$ sont exprimées sous la forme d'un identifiant, concept nouveau mais indispensable sur lequel nous reviendrons⁶.

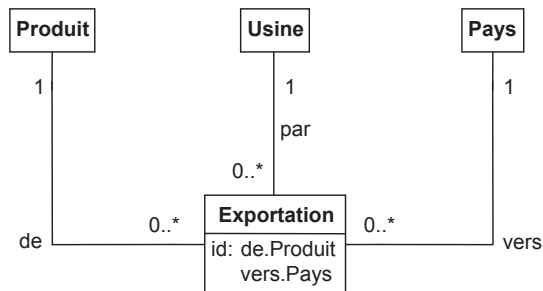


Figure 17.4 - Substitut recommandé d'une association ternaire en DB-UML

17.7 LES ATTRIBUTS

Les attributs UML obéissent aux mêmes règles que ceux du modèle Entité-association : ils sont obligatoires ou facultatifs, monovalués ou multivalués. L'utilisateur peut définir des types de données, simples ou composés, qui lui sont propres (TDU). Un attribut est composé s'il est défini sur un TDU composé.

17.8 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

5. Ajoutons aux problèmes logiques que pose l'interprétation de `min`, les interprétations très personnelles de certains auteurs. Citons en deux, trouvées dans des ouvrages largement répandus consacrés à la conception de bases de données. La première consiste à interpréter `min` selon le modèle Entité-association : `min = 1` indique que le rôle est obligatoire, notion qu'UML ne permet malheureusement pas de représenter pour les associations n-aires. La seconde est plus surprenante car elle viole la définition mathématique de la relation au sens du chapitre 3 : s'il existe dans `exporte` un couple (`Produit`, `Usine`), alors ce couple est obligatoirement associé à un objet `Pays`. À suivre cette interprétation, il existerait donc aussi des instances de `exporte` dans lesquelles `Usine` ou `Produit` seraient inconnus, puisque pour ces deux rôles, `min = 0` !

6. Cette forme ne permet pas d'exprimer directement la multiplicité `min = 1`, mais nous avons vu que cette faiblesse est plutôt un avantage. Si nécessaire, on utilisera une contrainte additionnelle.

diagramme 17.2). Ce concept, très général, est notamment utilisé pour définir les méthodes des classes d'objets.

17.9 LES ASSOCIATIONS QUALIFIÉES

Il s'agit d'une des curiosités les plus pittoresques d'UML, héritée d'OMT [Rumbaugh, 1991]. Le *qualifieur* d'une association R entre les classes A et B est un attribut (ou un groupe d'attributs) T attaché au rôle de 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 (figure 17.5). Un qualifieur peut induire une modification de la multiplicité du rôle de B dans R. En effet, en présence d'un qualifieur, cette multiplicité perd son sens d'origine : elle représente les nombres minimum et maximum d'objets B *dans chaque sous-ensemble défini par la partition*. Le cas le plus fréquent, et d'ailleurs le plus intéressant, est celui dans lequel cette multiplicité se réduit à 1 ou 0..1 (il s'agit d'un cas élémentaire d'identifiant hybride).



Figure 17.5 - Association R qualifiée par l'attribut T

Considérons les trois schémas de la figure 17.6. Les deux premiers sont extraits des documents de l'OMG où ils sont destinés à illustrer les associations qualifiées tandis que le troisième est repris d'un schéma bien connu du lecteur.

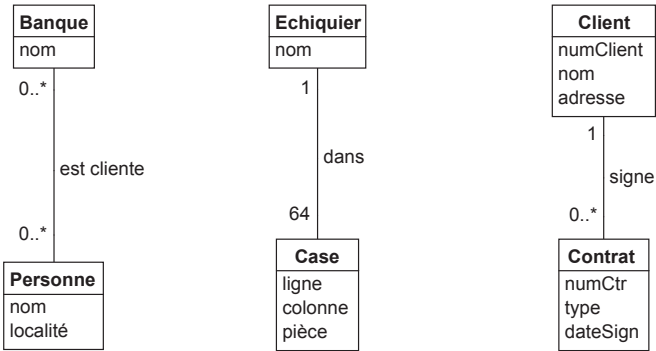


Figure 17.6 - Trois schémas binaires

La sémantique externe de ces schémas est considérée comme évidente. Nous ajoutons à présent des contraintes sur ceux-ci :

- dans une Banque, les clients ont des numéros de compte (numCpt) distincts ;
- dans un Echiquier, les cases ont des coordonnées ligne et colonne distinctes ;
- les contrats signés par un Client ont des numéros de contrat (numCtr) distincts.

Le modèle de classes d'UML permet d'exprimer ces contraintes en *qualifiant* les associations d'attributs ajoutés ou extraits d'une classe participante (figure 17.7).

- Dans le schéma de gauche, un nouvel attribut numCpt est ajouté à l'association et attaché au rôle Banque. On le lit ainsi : à un objet Banque et à une valeur de numCpt correspond un seul objet Personne.
- Dans le schéma central, les attributs ligne et colonne sont extraits de Case pour être ajoutés à l'association dans et attachés au rôle Echiquier. Ce schéma indique qu'à un objet Echiquier, une valeur de ligne et une valeur de colonne correspond un seul objet Case.
- Dans le schéma de droite, l'attribut numCtr est extrait de Contrat, ajouté à l'association signe et attaché au rôle Client. Ce schéma indique qu'à un objet Client et une valeur de numCtr correspond un seul objet Contrat.

On observe que l'introduction d'un qualifieur conduit à une réduction de la multiplicité du rôle opposé à ce qualifieur. En toute généralité, cette réduction est quelconque et peut être différente de 1 ou 0..1 comme illustré ici. Cependant, ce cas de figure est rarement illustré dans la littérature (voir [Blaha, 1998] par exemple).

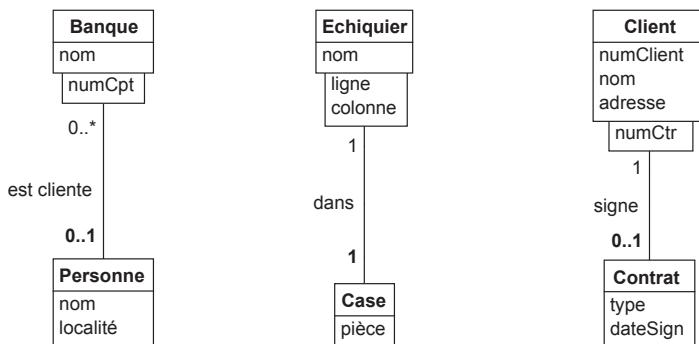


Figure 17.7 - Trois exemples d'associations qualifiées

Malgré l'intérêt apparent du concept d'association qualifiée, celui-ci se révèle à l'examen à la fois mal défini, irrégulier, incomplet et souvent inutile. Dans certains cas particuliers assez fréquents, il conduit à des schémas confus, erronés ou non normalisés. Mentionnons quelques problèmes que pose cette construction.

- La multiplicité réduite supprime la multiplicité d'origine ; il y a donc perte d'une information essentielle. L'exemple de l'Echiquier est particulièrement démonstratif : nous ne savons plus qu'un échiquier possède 64 cases.
- Le qualifieur exprime le plus souvent la forme la plus simple d'identifiant hybride. Aucune des autres formes, bien que très fréquentes (identifiant constitué de deux rôles par exemple, comme dans le schéma 12.19), n'est exprimable en UML.
- Le qualifieur est défini sur le rôle A et est présumé être attaché à l'association, mais il apparaît dans la quasi totalité des exemples comme un attribut de la classe

B. Dans la figure 17.7, nul ne contestera que numCtr est bien un attribut de Contrat et ligne et colonne des attributs de Case.

- On ne peut définir qu'un seul qualifieur par association ; la capacité de modélisation de l'association qualifiée est donc limitée.
- Si un même attribut d'une classe est utilisé pour qualifier plusieurs associations, il est nécessaire de le dupliquer. Le schéma comporte alors des constructions redondantes. Tel serait le cas du schéma 16.28 traduit en UML, où l'attribut Date qualifierait deux associations.
- Il existe deux modes totalement indépendants d'affectation d'un attribut à une association : l'association qualifiée et la classe association, décrite ci-après.
- Tout attribut monovalué obligatoire de B définit une partition des objets B associés à chaque objet A. Par exemple, les Personnes d'une Banque sont naturellement partitionnées selon les valeurs de localité. Il n'est nul besoin d'utiliser un qualifieur d'association pour le préciser.
- Le qualifieur attaché à une association *plusieurs-à-plusieurs* apparaît souvent comme une maladresse de conception, qui ignore une classe implicite. Le schéma de gauche cache de toute évidence le concept de Compte, dont la mise en évidence clarifierait la sémantique externe du schéma. Il est à noter que ce cas de figure, autorisé en UML 1, n'est plus repris en UML 2.

On suggère donc d'ignorer le concept d'association qualifiée.

17.10 LES CLASSES ASSOCIATIONS

Une association ne peut posséder d'attributs, à l'exception du cas très particulier des associations qualifiées. En revanche, on peut définir une *classe association*, c'est-à-dire une association qui est également une classe (cf. emprunte et réserve dans le schéma 17.11). Cette classe peut être impliquée dans d'autres associations ou relations *is-a*. Cette construction peut avoir une certaine utilité mais elle n'est pas indispensable (on peut sans réelle perte d'expressivité la remplacer par une classe) et s'avère complexe à utiliser et à interpréter pour certains analystes. En outre, ses attributs ne peuvent être utilisés pour la qualifier ni pour définir des identifiants d'association dans DB-UML.

17.11 LES ASSOCIATIONS D'AGRÉGATION ET DE COMPOSITION

UML inclut le type d'associations générique de *composition*, pour lequel il propose deux variantes : l'agrégation et la composition. Une association d'*agrégation* entre A et B associe à chaque objet composite du type A l'ensemble de ses composants du type B. On peut ainsi indiquer qu'une équipe *inclut* des personnes ou que les actes d'un colloque *sont composés d'*articles (figure 17.8).

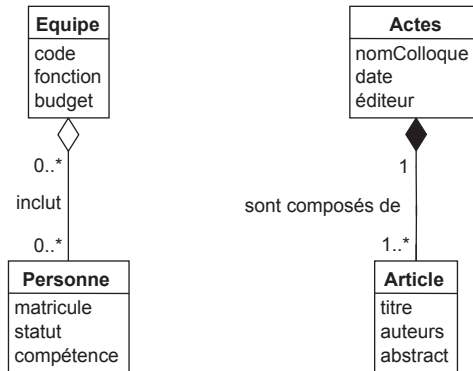


Figure 17.8 - Associations d'agrégation (*includ*) et de composition (*sont composés de*)

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*⁷.

17.12 LES CONTRAINTES

À de rares exception près, UML ne propose pas de contrainte d'intégrité explicite, laissant à l'analyste le soin de les exprimer en OCL ou via des *tags* ou encore d'annoter son schéma avec des commentaires en texte libre. La principale contrainte que UML offre est l'*exclusion* d'associations, notée *xor* (figure 17.11).

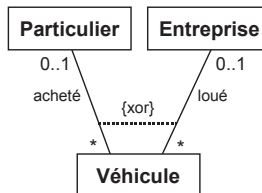


Figure 17.9 - Contrainte d'exclusion de rôles : un véhicule est soit loué soit acheté

On notera qu'UML ignore la plus importante des contraintes d'intégrité du domaine des bases de données : l'**identifiant**⁸. Cette option est conforme à la philosophie de la programmation et des bases de données orientées objet selon laquelle l'identification d'un objet est réalisée par des identifiants internes implicites. On recourra à des

7. La distinction entre ces deux structures est parfois justifiée par le mode d'implémentation dans un langage OO : dans le cas de l'agrégation, l'association est réalisée par des pointeurs vers les composants tandis que pour la composition, les composants apparaissent dans l'objet composé comme attributs complexes. Une telle explication ne relève évidemment pas du niveau conceptuel.

8. 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 hybride.

contraintes rédigées en OCL, comme illustré ci-après, pour exprimer chaque identifiant significatif, ce qui constitue un palliatif extrêmement lourd. Certains outils utilisent un *tag* PK pour marquer l'attribut (ou les attributs) jouant le rôle d'identifiant primaire. Il s'agit d'une astuce simpliste qui ne permet pas de représenter des situations telles que (1) une classe dotée de plusieurs identifiants, (2) des identifiants ayant des attributs en commun ou (3) des identifiants hybrides.

17.13 LE LANGAGE DE CONTRAINTES OCL

OCL (*Object Constraint Language*, versions 1.4 et 2.0) est un langage très puissant qui permet d'exprimer des contraintes dans certains types de diagrammes UML, et en particulier dans les diagrammes de classes. Nous ne décrivons ici que les aspects qui concernent les contraintes d'intégrité, appelées *invariants* en UML. Le schéma de la figure 17.10 servira de support aux exemples d'invariants. OCL utilisant le nom des rôles des associations, nous avons attribué aux rôles un nom explicite (le nom implicite est celui de la classe en minuscules).

Un invariant (inv) est attaché à une classe (son *contexte*) et est exprimé pour une instance courante de cette classe, dénommée *self* par défaut ou qui peut recevoir un surnom explicite (à la manière d'un alias de table en SQL).

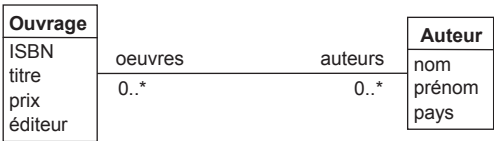


Figure 17.10 - Étude des contraintes d'intégrité ou invariants

Dans le premier exemple d'invariant ci-après, nommé *intervallePrix* et qui exprime une contrainte sur le prix des ouvrages, on repère la spécification du contexte *Ouvrage* et la définition de la condition, formée de deux conditions élémentaires connectées par l'opérateur logique *and*.

```
context Ouvrage
inv IntervallePrix: self.prix > 1 and self.prix < 100
```

On peut adopter une forme plus régulière, basée sur une variable qui désigne l'instance courante :

```
context o: Ouvrage
inv: o.prix > 1 and o.prix < 100
```

ou encore une forme implicite plus concise :

```
context Ouvrage
inv: prix > 1 and prix < 100
```

Le composant de base d'une condition est l'*expression*. Une expression est une sorte de requête qui désigne, soit une constante (1, 'Mercier', 123,5), soit un élément ou une collection d'éléments (ensemble, séquence ou multi-ensemble) parmi les instances

des objets du schéma (classes et attributs notamment), soit encore une valeur calculée à partir de ces éléments.

a) Les expressions

Désignation d'instances

On désigne des instances par une expression en cascade qui décrit un chemin ou une navigation dans le schéma. Le point de départ est l'instance courante du contexte (self, variable ou implicite), les étapes intermédiaires sont des associations traversées, nommées par le rôle vers lequel on navigue et la destination est un attribut ou une classe. Quelques exemples relatifs à une instance de la classe *Ouvrage*, dite *ouvrage courant* :

<code>self.prix</code>	<i>le prix de l'ouvrage courant</i>
<code>self.auteurs</code>	<i>les auteurs de l'ouvrage courant</i>
<code>self.auteurs.pays</code>	<i>les pays des auteurs de l'ouvrage courant</i>
<code>self.auteurs.oeuvres</code>	<i>les ouvrages écrits par les auteurs de l'ouvrage courant</i>
<code>self.auteurs.oeuvres.titre</code>	<i>les titres des ouvrages écrits par les auteurs de l'ouvrage courant</i>

Fonction de filtrage

On peut appliquer une fonction de filtrage `select` à une collection de manière à ne conserver que les éléments qui satisfont une condition. Nous en illustrons la syntaxe dans le contexte d'un objet *Auteur* courant :

```
self.oeuvres->select (prix<10) les oeuvres à moins de 10 € de l'auteur courant
```

Fonctions agrégatives

Les fonctions `size`, `sum` et `count` renvoient une valeur calculée sur un ensemble d'éléments. Il n'en existe malheureusement pas d'autres (telles que `max`, `min` ou `avg`). Soient `o` une instance courante d'*Ouvrage* et `a` une instance courante d'*Auteur*.

<code>o.auteurs->size()</code>	<i>nombre d'auteurs de l'ouvrage o</i>
<code>a.oeuvres.prix->sum()</code>	<i>somme des prix des ouvrages de l'auteur a</i>

Autres

Des fonctions permettent de changer la nature d'une collection. La fonction `asSet()`, appliquée à une collection, renvoie l'ensemble des éléments de celle-ci. Par exemple, si `o` désigne une instance d'*Ouvrage*, `o.auteurs.oeuvres` désigne les instances d'*Ouvrage* des Auteurs de `o`, qui constituent un multi-ensemble (si un ouvrage a deux auteurs, cet ouvrage apparaîtra deux fois), alors que `o.auteurs.oeuvres->asSet()` transforme ce multi-ensemble en un ensemble.

b) Les conditions unaires

Une condition unaire est obtenue par l'application d'une fonction logique à une collection d'éléments. On en décrit quatre : `isEmpty` (et son inverse `notEmpty`), `isUnique`, `forAll` et `exists`.

- `isEmpty()` et `notEmpty()` : renvoient vrai si la collection est respectivement vide et non vide. La condition suivante est vraie si l'ouvrage `o` a au moins un auteur :

```
o.auteurs->notEmpty()
```

- `isUnique(attribut)` : renvoie vrai si les valeurs de *attribut* de la collection sont uniques. La condition suivante est vraie si les auteurs de l'ouvrage `o` ont des noms distincts :

```
o.auteurs->isUnique(nom)
```

- `forAll` : renvoie vrai si tous les éléments de la collection satisfont une condition déterminée. La condition suivante est vraie si tous les auteurs de l'ouvrage `o` sont français :

```
o.auteurs->forAll(pays = 'France')
```

- `exists` : renvoie vrai si la collection contient au moins un élément qui satisfait une condition déterminée. La condition suivante est vraie si parmi les auteurs de l'ouvrage `o`, l'un au moins n'est pas français :

```
o.auteurs->exists(pays <> 'France')
```

La fonction `isUnique` peut être appliquée à l'ensemble des instances d'une classe, indépendamment de l'instance courante. L'invariant suivant impose l'unicité des codes ISBN des ouvrages, ce qui revient à déclarer l'attribut ISBN identifiant de la classe `OUVRAGE`⁹. La fonction `allInstance` renvoie la population d'une classe.

```
context OUVRAGE
```

```
inv IdOuvrage: OUVRAGE.allInstance()->isUnique(ISBN)
```

c) Les conditions binaires

La condition binaire exprime une comparaison entre les résultats de l'évaluation de deux expressions. La condition suivante indique que le prix d'un ouvrage est d'au moins 5 € par auteur :

```
self.prix > 5*self.auteurs->size()
```

d) Les opérateurs logiques

Les opérateurs logiques permettent de construire des conditions composées de conditions plus simples. OCL propose les opérateurs `and`, `or`, `xor`, `not` et `implies` (implication). Les parenthèses permettent de construire des conditions complexes.

9. On entend parfois l'argument selon lequel UML n'a pas besoin du concept d'identifiant puisqu'on peut l'exprimer par un invariant OCL. Selon le même raisonnement, on n'aurait pas besoin en SQL ni des *primary keys*, ni des *foreign keys* ni des *colonnes obligatoires*, puisqu'il est possible de les simuler à l'aide de prédicats *check* et de *triggers*.

Dans les diagrammes de classes, OCL est également utilisé pour préciser la valeur initiale d'un attribut et les règles de calcul d'un attribut ou d'une association dérivés. Le lecteur trouvera une définition complète d'OCL 2.0 dans [OMG, 2006].

17.14 UNE QUESTION DE VOCABULAIRE

Les concepts du modèle de classes UML et du modèle Entité-association sont proches mais cependant distincts, notamment quant à leur dénomination. Le tableau suivant résume les correspondances.

Classes UML		Entité-association	
Type	Instance	Type	Instance
classe	objet	type d'entité	entité
association	lien	type d'associations	association
multiplicité		cardinalité (<i>inversée</i>)	
attribut	valeur d'attribut	attribut	valeur d'attribut

17.15 EXEMPLE DE SCHÉMA DE CLASSES EN UML

Dans la figure 17.11, nous exprimons en UML l'essentiel des constructions du schéma de la figure 16.42. En raison des limitations du métamodèle de classes d'UML, certains éléments importants sont manquants, tels les identifiants, à l'exception de celui d'Exemplaire.

17.16 LE MODÈLE DE CLASSES D'UML REVISITÉ (DB-UML)

En raison de son origine, la modélisation des classes dans les programmes orientés-objet, le métamodèle de classes UML est mal adapté à l'expression de schémas conceptuels professionnels. La présence de constructions d'utilité douteuse (les *associations qualifiées*) ou dont l'interprétation est complexe et non intuitive (les *associations n-aires*), l'absence de définition précise pour la plupart des concepts (comment se comporte une composition lors de modifications ?) et surtout de graves lacunes telles que celle des identifiants de classes suffiraient à l'écarter des modes d'expression de schémas de bases de données. En revanche, sa standardisation, son universalité ainsi que des apports très intéressants (OCL) poussent beaucoup de développeurs à adopter, parfois à leur corps défendant, UML pour exprimer des schémas conceptuels.

Pour l'heure, l'approche UML en matière de modélisation conceptuelle de bases de données apparaît trop faible pour se substituer sans dommages à des modèles éprouvés comme celui de Merise 2 [Tardieu, 2000] : quelles que soient les qualités des moissonneuses-batteuses, elles ne sont, pour l'instant du moins, pas conçues pour le transport du courrier express.

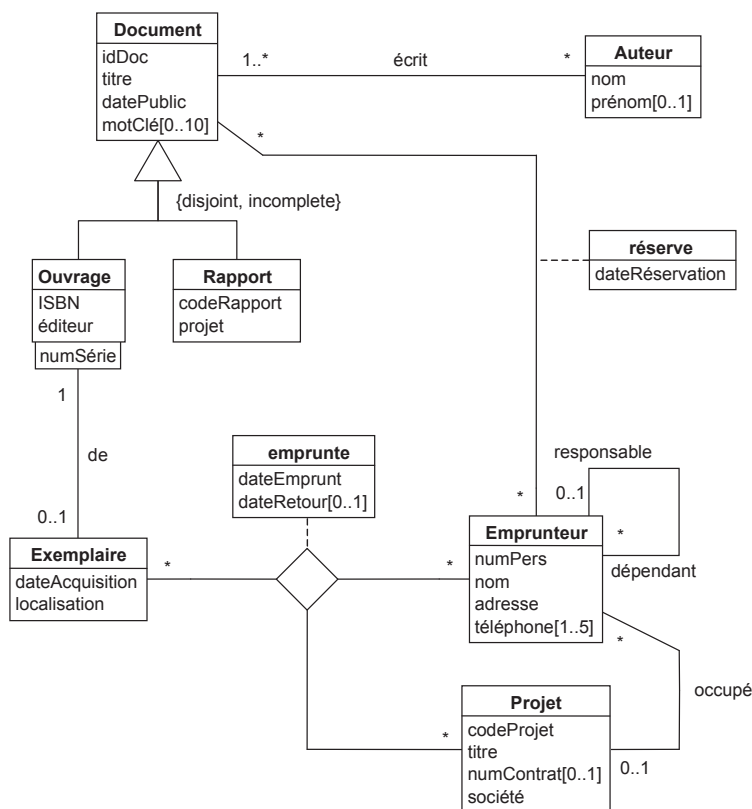


Figure 17.11 - Traduction approximative du schéma 16.42 en UML. Les identifiants n'ont pas été exprimés, à l'exception de celui d'Exempleire qui correspond à une association qualifiée. Les attributs composés n'apparaissent pas en tant que tels.

L'OMG a publié en décembre 2005 un *Request for proposal* général couvrant différents aspects de la modélisation des (bases de) données [OMG, 2005]. Des propositions sont sollicitées notamment pour la définition d'un métamodèle pour la gestion de l'information et pour des profils décrivant le modèle Entité-association, le modèle SQL, le modèle XML et le modèle de fichiers classiques. L'état des travaux dix ans plus tard, tels que décrits sur le site de l'OMG¹⁰, ne semble cependant pas très encourageant. Seules des initiatives industrielles ont conduit à des propositions utilisables, telles que *IBM Rational Software* et [Ambler, 2006].

Le tableau n'est heureusement pas aussi noir qu'on pourrait le penser. Si on accepte quelques compromis, il est possible de définir une variante du métamodèle de classes d'UML parfaitement adaptée à la représentation de schémas conceptuels simples. En écartant les constructions qui posent problème et en ajoutant les

10. [http://www.omgwiki.org/imm/doku.php?id=Welcome to IMM](http://www.omgwiki.org/imm/doku.php?id=Welcome+to+IMM) (juin 2015)

concepts indispensables mais manquants, on obtient un métamodèle convenable, que nous appellerons DB-UML, constitué notamment des composants suivants :

- les stéréotypes ;
- les classes d'objets ;
- les relations de généralisation-spécialisation et leurs propriétés ;
- les types définis par l'utilisateur ;
- les attributs; obligatoires ou facultatifs, mono-valués ou multivalués ;
- les opérations associées aux classes ;
- les associations binaires ;
- les associations de composition (éventuellement d'agrégation) ;
- le langage OCL.

On y ajouterait

- les attributs composés ;
- les identifiants de classe ;
- les contraintes d'intégrité de base.

Cette variante ne reprendrait pas les constructions suivantes, mal définies dangereuses et/ou inutiles car exprimables plus simplement.

- les associations n-aires ;
- les associations qualifiées ;
- les classes associations.

Le schéma de la figure 17.11 pourrait alors s'exprimer en DB-UML sans perte d'information sous la forme du schéma 17.12.

17.17 SÉMANTIQUE DU MODÈLE DE CLASSES DB-UML

Le métamodèle de classe d'UML tel qu'il est proposé par l'OMG ne dispose pas encore d'une sémantique interne communément acceptée, malgré les nombreuses propositions qui ont été publiées à ce jour. La complexité et les lacunes de ce modèle, la présence d'idiosyncrasies telles que les associations qualifiées et les associations n-aires et surtout les divergences entre les communautés (bases de données et programmation notamment) qui utilisent UML comme base d'expression de modèles rendent particulièrement ardu tout effort de clarification.

En revanche, le sous-ensemble du métamodèle de classes DB-UML que nous venons de définir est suffisamment simple et dépouillé pour être doté d'une sémantique relationnelle. En effet, apparaissant comme un sous-ensemble du modèle Entité-association du chapitre 16, il hérite des principes de correspondance que nous avons établis à la section 16.17.

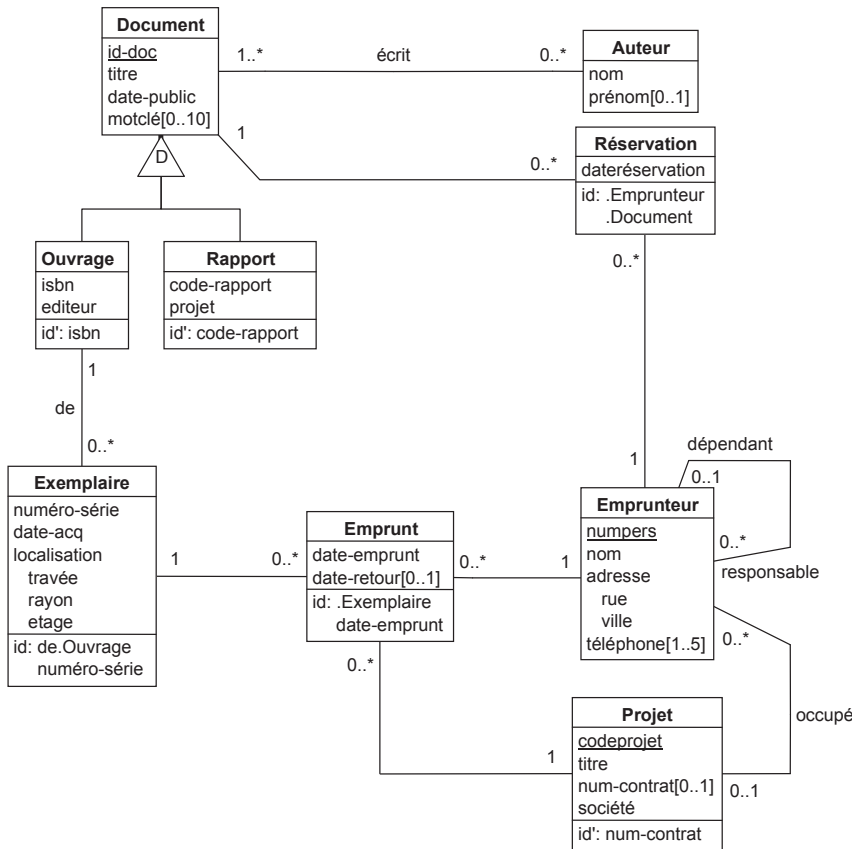


Figure 17.12 - DB-UML permet de représenter la plupart des structures d'information au niveau conceptuel

Les propositions UML de l'OMG comportent un important volet, dénommé *Meta-Object Facility* ou *MOF*, proposant un métamodèle des notations dont les instances sont des *métaschémas* : chacun des types de diagrammes de la figure 17.1 peut être décrit par un de ces méta-schémas. Cependant, le niveau très élevé de généralité du *MOF* le destine clairement à un public de développeurs d'AGL.

UML définit des notations pour des types de documents importants dans le développement d'un système d'information. Il ne propose pas de sémantique interne pour celles-ci, ce qui en fait sa souplesse mais aussi son ambiguïté. En outre, il ne préconise pas de manière particulière d'utiliser ces notations pour construire et faire évoluer des systèmes, c'est-à-dire qu'il n'impose aucune *méthode*. Certaines propositions existent telles que RUP (*Rational Unified Process*) et 2TUP (*2 Tracks Unified Process*).

17.18 QUE RETENIR ?

Parmi les différentes notations d'UML, celle des **diagrammes de classes** est parfois proposée comme un substitut au modèle Entité-association¹¹. Cette notation (que nous appellerons ici *métamodèle de classes*) permet de définir des **classes** d'objets, des hiérarchies **is-a**, des **associations** binaires et n-aires, des **attributs**, des **opérations** sur les classes (et leurs instances), ainsi que, via le langage OCL, des **contraintes** sur les instances de ces objets. Chaque membre d'une association constitue un **rôle**, caractérisé par une propriété de **multiplicité**. Celle-ci, du moins pour les associations binaires, est identique à la contrainte de cardinalité, mais est associée au rôle opposé. Les associations d'**agrégation** et de **composition** sont destinées à représenter les relations de composition entre objets. Une **association qualifiée** permet d'affecter un groupe d'attributs à une association de sorte que les valeurs de ces attributs définissent une partition des objets d'une classe participante. Une **classe association** est une association qui est également une classe.

Indépendamment de son utilisation, il apparaît que certaines constructions mal définies, inutiles et dangereuses doivent être écartées : tel est le cas des *associations n-aires* et des *associations qualifiées*. Pour ce qui concerne la modélisation conceptuelle dans le domaine des bases de données, on note l'absence quasi totale de contraintes d'intégrité prédéfinies, et en particulier des identifiants. Comme tel, le modèle de classes UML ne convient donc pas pour modéliser les bases de données.

Si on accepte de compléter ce modèle des constructions manquantes et d'écarter les constructions déficientes, on peut dériver un sous-modèle convenant à l'expression de schémas conceptuels. Ce modèle, appelé **DB-UML**, comprend les constructions suivantes : *classes*, *relations is-a*, *attributs*, *opérations*, *associations binaires*, *identifiants d'entités* et *contraintes prédéfinies* du modèle Entité-association étendu, langage de contraintes OCL. On a écarté les *associations n-aires*, les *associations qualifiées* et les *classes associations*.

17.19 POUR EN SAVOIR PLUS

La première référence est bien sûr le site de l'OMG <http://www.omg.org/>, sur lequel on trouvera les dernières versions des différentes recommandations relatives à UML. Ces textes, par nature, ne brillant pas par leur lisibilité, on leur préférera le plus souvent les ouvrages qu'on en a tirés. On recommandera [Booch, 2004], traduction française de la référence rédigée par les trois auteurs d'UML, et [Larman, 2005] introduisant des aspects méthodologique. Les actes des conférences consacrées à UML, en particulier la série très populaire *MoDELS* ainsi que les conférences CAiSE constituent des sources de choix pour le lecteur désireux de compléter ses

11. Cette formulation est abusive : une notation permet d'exprimer des modèles relevant d'un métamodèle, mais n'est pas comparable à ce dernier. On peut associer (et d'ailleurs, on ne s'en prive pas) plusieurs notations (graphiques, textuelles, XML) à un même métamodèle. Mais, comme l'exprime la sagesse populaire, *c'est mal formulé mais on comprend bien ce qu'on veut dire !*

connaissances. Le lecteur trouvera dans [Soutou, 2012] une présentation plus enthousiaste du modèle de classe d'UML pour les bases de données. L'analyse des associations n-aires du présent chapitre s'inspire de la référence [Hainaut, 2002b] disponible sur le site de l'ouvrage.

17.20 CRÉER UN SCHÉMA CONCEPTUEL UML À L'AIDE D'UN OUTIL

Il existe de nombreux outils, dont certains sont gratuits, permettant de dessiner des diagrammes de classes UML et de générer du code SQL à partir de ces derniers. L'atelier DB-MAIN offre deux modèles de représentation de schémas conceptuels : Entité-association étendu et DB-UML. Le choix du modèle se fait lors de la création du schéma. L'atelier offre en outre deux fonctions de conversion automatiques de schémas :

- génération d'un schéma Entité-association équivalent à un schéma DB-UML ;
- génération d'un schéma DB-UML équivalent à un schéma Entité-association.

Des compléments et des exercices sont disponibles à l'annexe A17.

Chapitre 18

Analyse conceptuelle du domaine d'application

Ce chapitre reprend, complète et approfondit la démarche de construction d'un schéma conceptuel développée au chapitre 13. Il exploite en particulier les extensions du modèle Entité-association et introduit de nouveaux processus indispensables au traitement de grands schémas complexes. Il décrit notamment l'analyse de sources non textuelles, la correction, la normalisation, l'évaluation et l'intégration de schémas.

18.1 INTRODUCTION

Le chapitre 13 nous a appris à construire le schéma conceptuel d'un domaine d'application peu complexe. Les moyens d'expression dont nous disposions pour construire de tels schémas étaient en effet relativement rudimentaires : types d'entités, attributs simples, types d'associations binaires. Le chapitre 16 nous a apporté de nouvelles constructions qui offrent des possibilités d'expression beaucoup plus étendues. Ainsi, les *relations is-a*, les *attributs composés* et *multivalués*, les *types d'associations n-aires*, la généralisation des *cardinalités* des attributs et des rôles et de nouvelles formes de *contraintes* nous permettent de traduire de manière plus simple et plus expressive les concepts statiques du domaine d'application dans un schéma conceptuel.

L'utilisation d'un modèle plus riche n'est pas la seule raison pour laquelle il est nécessaire d'étendre la démarche du chapitre 13. Notre ambition est aussi d'aborder des projets plus complexes et de plus grande taille. Au cours de ceux-ci, nous rencontrerons de nouveaux problèmes que nous devrons analyser plus en profon-

deur et pour lesquels nous devrons élaborer des techniques de résolution plus puissantes. Nous devrons également définir une organisation des tâches plus professionnelle permettant de maîtriser cette taille et cette complexité en toute sécurité.

Nous extrayons de la figure 15.3 la description externe de l'analyse conceptuelle (figure 18.1). Parmi les critères dont nous devons tenir compte, nous retiendrons ceux-ci : le schéma doit être **complet**, **correct** et **normalisé**, ce dernier critère regroupant notamment des propriétés d'expressivité, de minimalité, de lisibilité que nous aurons l'occasion d'étudier plus en détail dans ce chapitre.

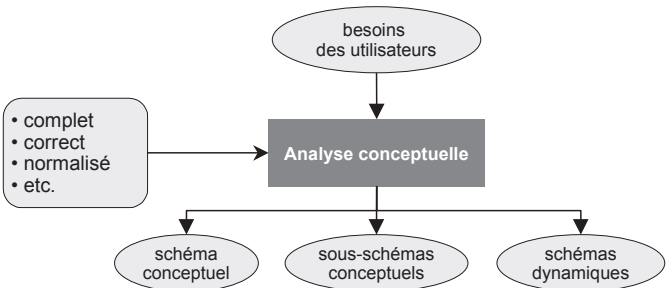


Figure 18.1 - Analyse conceptuelle - Description externe

La stratégie d'analyse conceptuelle que nous allons décrire se base sur le principe de décomposition du domaine d'application en sous-ensembles homogènes, ou sous-systèmes, plus aisés à modéliser que le domaine tout entier. On déduit de chaque sous-système un schéma conceptuel, puis on fusionne les sous-schémas ainsi obtenus en un schéma conceptuel global (figure 18.2).

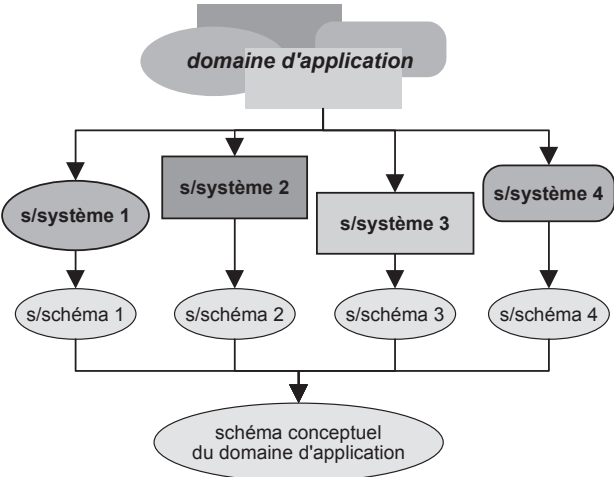


Figure 18.2 - Construction du schéma conceptuel par décomposition du domaine d'application

Nous n'abordons pas ici la question de la décomposition d'un domaine en sous-système. La gestion de tout organisme complexe se base en effet sur une décompo-

sition naturelle selon une ou plusieurs dimensions fonctionnelles (finance, personnel, production, marketing) ou spatiales (département, branche, filiales, sites) par exemple. C'est bien sûr cette décomposition que nous adopterons. Observons que deux sous-systèmes se partagent en général des concepts communs. Ainsi, le concept d'*employé* se retrouvera dans la *gestion du personnel*, la *gestion financière* et la *gestion des locaux*.

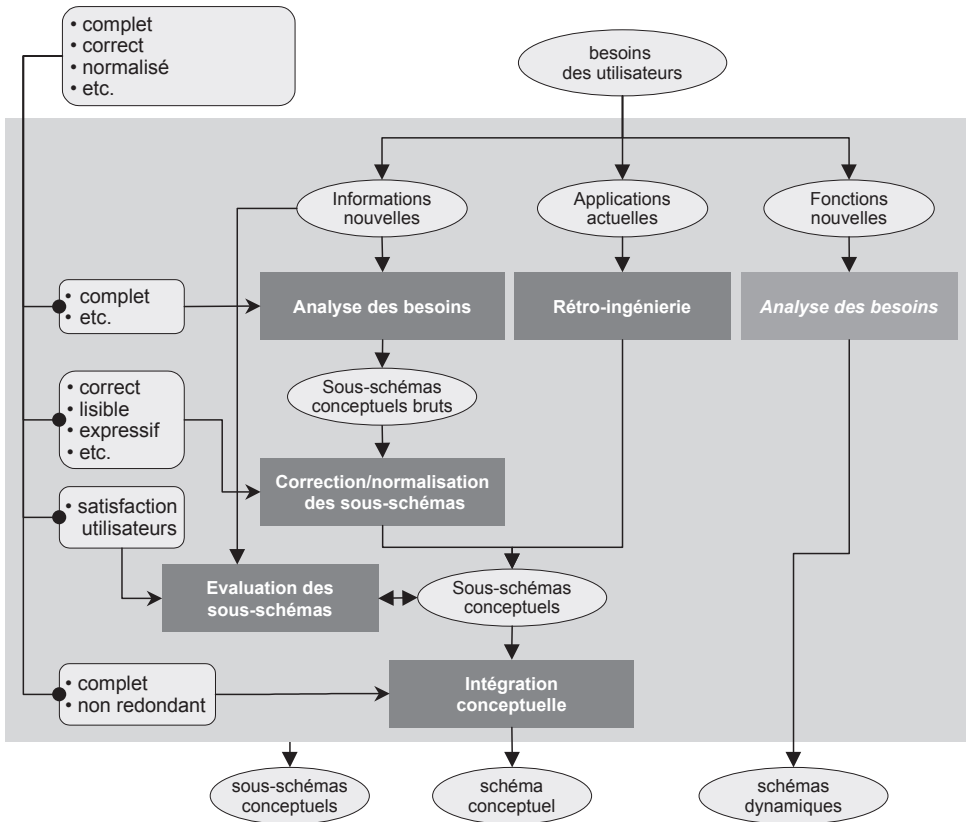


Figure 18.3 - Stratégie d'analyse conceptuelle orientée données. L'analyse des aspects dynamiques (branche de droite) est indiquée pour mémoire.

La figure 18.3 décrit cette stratégie. Elle met en évidence des caractéristiques nouvelles par rapport au processus simplifié du chapitre 13. Avant tout, on se situe dans un scénario dans lequel un système d'information, y compris sa base de données, **existe déjà** (Applications actuelles) et satisfait une partie des besoins des utilisateurs. D'autre part, une partie de ces besoins (Besoins fonctionnels) s'exprimeront sous la forme de schémas dynamiques, qui se traduiront physiquement en programmes, méthodes ou services. Nous ignorerons cette dernière branche dans cet ouvrage, bien que son résultat intervienne dans la conception physique de la base de données. Examinons de plus près les processus composant cette stratégie.

- **Analyse des besoins.** Ce processus exprime les concepts sous-jacents à une collection de documents décrivant un sous-système sous la forme d'un (sous-) schéma conceptuel (sous-schémas conceptuels bruts). L'analyse des besoins est décrite dans la section 18.2 pour les documents textuels et dans la section 18.4 pour les autres types de sources, tels que les documents structurés.
- **Correction/normalisation des sous-schémas.** Le schéma résultant de l'analyse des besoins sera parfois de qualité médiocre et contiendra des erreurs techniques et des structures maladroites. Ce processus a pour objectif l'analyse fine du schéma, sa correction (section 18.6) et sa normalisation (section 18.7). Il produit des sous-schémas conceptuels corrects et normalisés.
- **Évaluation des sous-schémas.** Il faut ensuite vérifier que chaque sous-schéma conceptuel correspond parfaitement aux besoins exprimés par les utilisateurs (section 18.8). Si nécessaire, ces schémas sont modifiés.
- **Rétro-ingénierie.** Si la base de données existante ne dispose pas d'une documentation à jour (dont un schéma conceptuel), il est nécessaire de la reconstruire ou de l'actualiser afin d'identifier les objets qui satisfont déjà certains des besoins exprimés. Ce processus sera étudié au chapitre 22.
- **Intégration conceptuelle.** Les schémas relatifs aux sous-systèmes sont ensuite intégrés pour former le schéma conceptuel du domaine d'application. L'intégration de schémas sera étudiée dans la section 18.9.

18.2 ANALYSE DES ÉNONCÉS TEXTUELS

Le processus de traduction d'un énoncé textuel en un schéma conceptuel a été introduit à la section 13.3 sous l'intitulé *Décomposition de l'énoncé*. La puissance accrue du modèle Entité-association va nous permettre d'analyser de manière moins rigide les textes dans lesquels nous cherchons à identifier les concepts du domaine d'application. Le principe de l'analyse de textes reste inchangé : on décompose un texte complexe en propositions (phrases) élémentaires qui se traduisent de manière directe en constructions du schéma conceptuel. On identifie trois types de propositions : *existentielles*, *relationnelles binaires* et *relationnelles n-aires*.

18.2.1 Proposition existentielle

Il s'agit d'une proposition *unaire* (on y trouve **un** terme pertinent) qui exprime l'**existence d'un concept**. En règle générale, cette proposition s'interprète par la création d'un type d'entités.

Quelques exemples

On considère les **clients**
de l'entreprise



CLIENT



Objection : une proposition unaire pourrait mentionner un concept mineur qui se verrait ainsi illégalement promu au rang de type d'entités ! Ainsi, la phrase « *c'est sur le **taux de remise** que tout va se jouer* » est une proposition existentielle qui tendrait à introduire le type d'entités TAUX-de-REMISE, ce qui risque d'être excessif. Pas d'inquiétude, s'il s'avère que ce type d'entités ne se justifie pas à la fin de l'analyse, le processus de normalisation lui rendra un statut plus approprié.

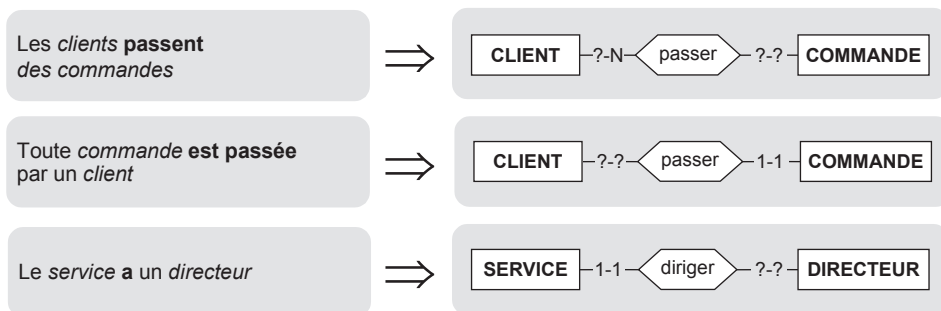
18.2.2 Proposition relationnelle binaire

Une proposition binaire, à la base de la procédure simplifiée, mentionne **deux** termes, désignant des concepts déjà connus ou nouveaux, et exprime une relation binaire entre ceux-ci. La phrase « *les **clients** passent des **commandes*** » indique l'existence des concepts de *client* et de *commande*, mais surtout l'existence d'une relation entre ceux-ci, relation qui s'exprime par le verbe *passer*. Nous allons identifier trois catégories de propositions binaire selon l'intention du verbe utilisé¹.

a) Les propositions d'action

Dans une proposition d'action, le verbe exprime l'existence dans le domaine d'application d'une relation spécialisée entre deux concepts. Ceux-ci auront en principe été déjà repérés, mais il ne sera pas rare que l'un d'eux apparaisse pour la première fois dans cette proposition. Les cardinalités de certains rôles peuvent se déduire de la proposition.

Quelques exemples



1. Sur la traduction des verbes, lire notamment la référence [Storey, 2004].

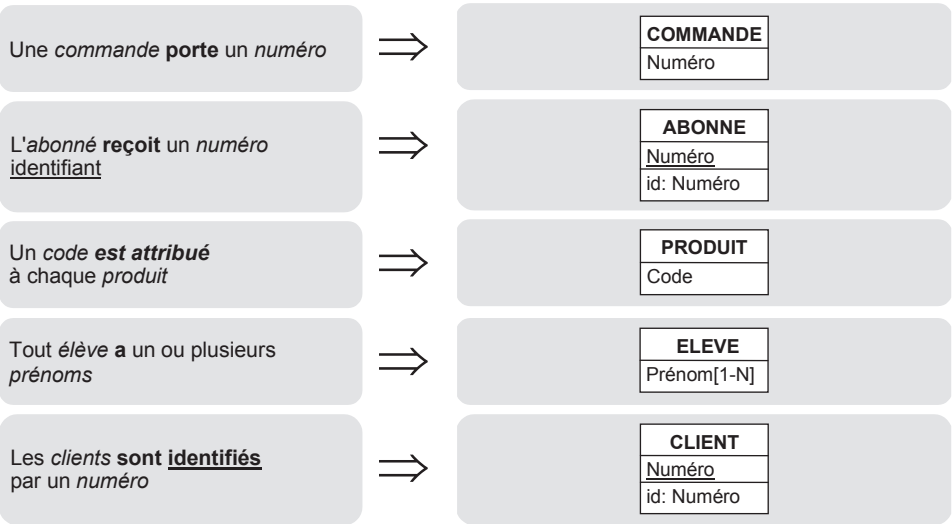


Remarque. Attention aux différentes formes verbales, notamment les voix active et passive qui à ce stade sont synonymes : *passé* et *est passé* représentent le même verbe et sont traduits en un seul type d'associations.

b) Les propositions de caractérisation

Ici, le verbe affecte une propriété caractéristique à un concept. Dans certains cas, le rôle de cette propriété peut être explicitement indiqué (c'est le cas d'un identifiant par exemple). L'existence des attributs multivalués facilitera l'expression des *propriétés multivaluées* dont nous avons discuté à la section 13.5.6.

Quelques exemples

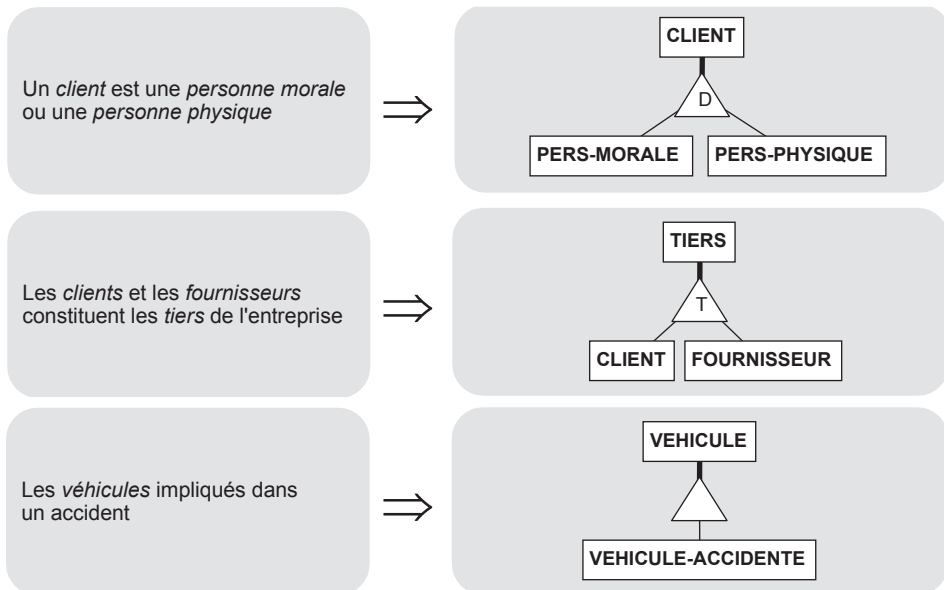


c) Les propositions de classification

Dans une proposition de classification, le verbe exprime un lien de généralisation ou de spécialisation entre deux concepts.

Quelques exemples





d) Résolution des ambiguïtés

L'exposé précédent constitue un guide mais en aucune manière une procédure rigoureuse. Il est à noter que la catégorie d'un verbe ne lui est pas intrinsèque, mais dépend du contexte de la phrase. Dans les phrases « *les clients reçoivent des colis* » et « *les commandes reçoivent un numéro* », le verbe recevoir exprime dans un cas l'action et dans l'autre la caractérisation. Remplacer un verbe par *est caractérisé par* (ou *est une caractéristique de*) permet le plus souvent d'identifier une proposition de caractérisation.

Le verbe avoir en particulier est source de nombreuse ambiguïtés, d'autant plus que le véritable verbe est souvent caché dans le nom d'un des deux concepts identifiés dans la proposition. *Avoir un directeur* doit se comprendre comme *être dirigé par un directeur* (ou l'inverse), lequel pourrait, de manière plus générale être un *employé* ou un *cadre*. Le verbe diriger est ici caché dans le terme directeur.

Le problème n'est en réalité pas aussi critique qu'on pourrait le penser. En effet, traduire un concept sous la forme d'un attribut ou sous celle d'un type d'entités conduit à des schémas qui sont transformables l'un dans l'autre sans perte d'information. Par exemple, la proposition « *Tout ouvrage a un auteur* » peut être interprétée de deux manières largement équivalentes, sinon identiques², illustrées à la figure 18.4.

2. Dans ce cas précis, il est trop tôt pour juger de l'équivalence des deux schémas. Deux divergences possibles, qui devront être évaluées dans le schéma final : (1) dans un schéma une instance d'*auteur* est une entité et dans l'autre elle est une valeur, (2) dans l'un, une instance d'*auteur* pourrait exister indépendamment de l'*ouvrage* (selon la cardinalité du rôle écrire.AUTEUR), dans l'autre ce n'est pas le cas (pas de valeur d'Auteur sans son entité parente OUVRAGE).

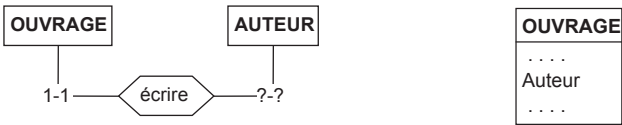


Figure 18.4 - Deux schémas équivalents pour une même proposition

e) Formes synthétiques

Certains raccourcis de langage recouvrent une ou plusieurs propositions binaires :

Raccourci	Propositions explicites
... le prix d' un produit est défini ...	Un produit a un prix. Le prix est défini ...
[département] ... Son directeur est attaché ...	Un département a un directeur. Le directeur est attaché ...
Les véhicules impliqués dans un accident ...	Un véhicule peut être impliqué dans un accident. <i>ou</i> Il existe une catégorie de véhicules : ceux qui sont impliqués dans un accident.
[commande] ... Elle comporte des détails qui sont vérifiés ...	Une commande comporte des détails Les détails sont vérifiés ...
Le cheval est monté par un cavalier qui est identifié par un numéro de dossard	Le cheval est monté par un cavalier. Un cavalier est identifié par un numéro de dossard.
Une commande est passée par un client et possède un montant total	Une commande est passée par un client. Une commande possède un montant total.

18.2.3 Proposition relationnelle n-aire

Une proposition n-aire mentionne **plus de deux** termes (**n** en toute généralité), désignant des concepts déjà connus ou nouveaux, et exprime une relation n-aire entre ceux-ci. Nous distinguerons d'abord les propositions n-aires homogènes, puis celles qui incluent la désignation d'une propriété, qu'on qualifiera d'hétérogènes, et enfin les propositions décomposables.

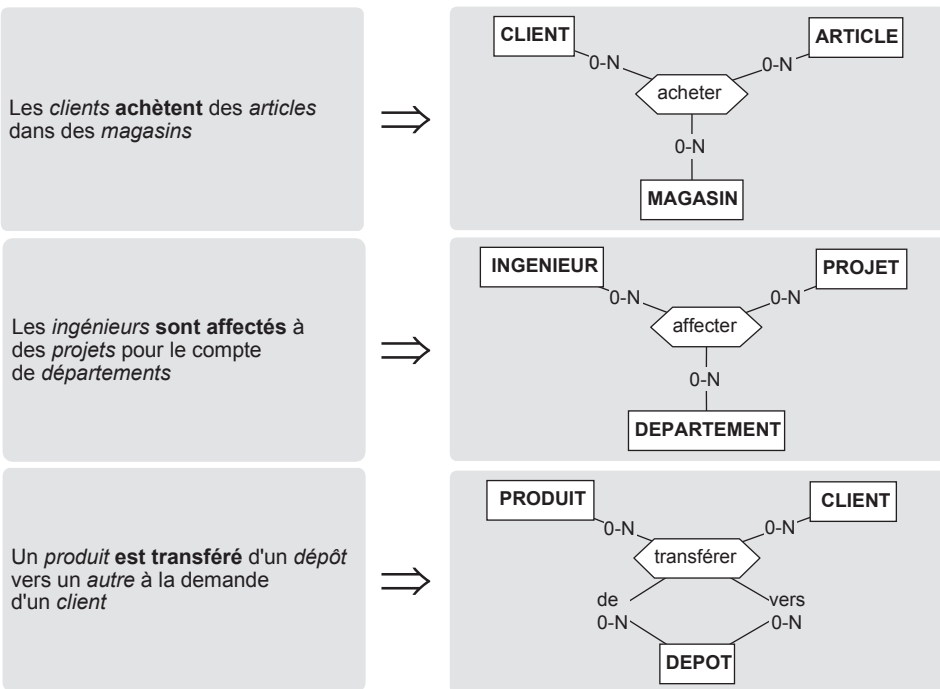
a) Les propositions n-aires homogènes

On qualifiera d'*homogène* une proposition dont les n termes désignent effectivement tous des concepts représentés par des types d'entités.

Remarque

Dans les exemples précédents, on a imposé les contraintes de cardinalité les plus plausibles.

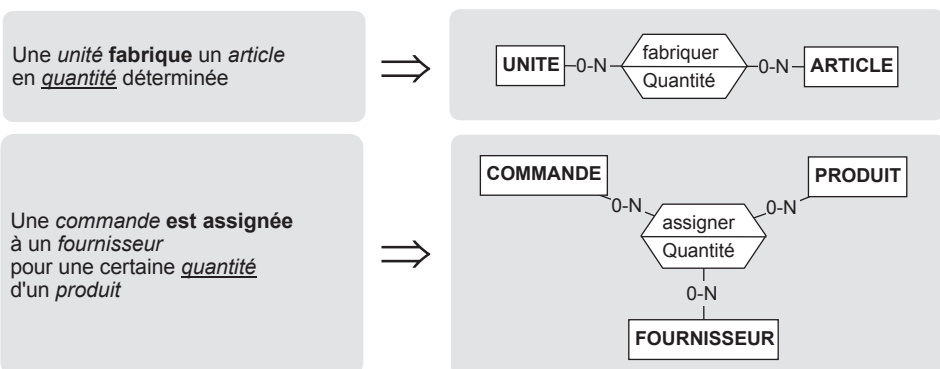
Quelques exemples



b) Les propositions n-aires hétérogènes

Il peut arriver qu'un des n concepts mentionnés n'en soit pas réellement un. Un des termes de la proposition désigne alors une simple propriété de la relation qui unit les $n-1$ autres concepts. Dans ce cas, la proposition se traduit par un type d'associations $(n-1)$ -aire doté d'un attribut.

Quelques exemples





c) Les propositions n-aires irréductibles et décomposables

Une proposition n-aire est dite irréductible lorsqu'elle n'est équivalente à aucun ensemble de propositions binaires. Nous avons déjà examiné le phénomène de décomposition d'une proposition complexe à la section 13.3.4. Ainsi, nous y observons que la proposition ternaire « *les clients achètent des produits chez des fournisseurs* » ne pouvait pas être remplacée par les trois propositions binaires « *les clients achètent des produits* », « *les fournisseurs vendent des produits* » et « *les clients achètent chez des fournisseurs* ».

Nous disposons maintenant de deux notions nouvelles qui vont nous permettre d'étudier et résoudre cette question de manière à la fois moins laborieuse (à comparer avec la discussion 13.3.5) et plus convaincante : les *types d'associations n-aires* et la *normalisation d'une relation*.

La proposition ternaire reprise précédemment peut à présent se traduire sous la forme du type d'associations ternaire acheter entre les types d'entités CLIENT, PRODUIT et FOURNISSEUR de la figure 16.7. Poser la question du caractère décomposable de la proposition revient à se demander si le type d'associations acheter est décomposable ou s'il est irréductible : peut-on remplacer acheter par deux, voire trois, types d'associations binaires ?

Grâce à la sémantique interne du modèle Entité-association (section 16.17), il est très simple de répondre à cette question. Un type d'associations s'interprète comme une relation entre des domaines entités. Il est décomposable si sa relation est elle-même décomposable, c'est-à-dire (selon 3.7) si elle possède au moins trois attributs et s'il existe une dépendance fonctionnelle entre deux sous-ensembles disjoints de ces attributs³. On a donc ici :

acheter(CLIENT, PRODUIT, FOURNISSEUR)

Cette relation est-elle décomposable ? Elle ne peut se décomposer que selon une dépendance fonctionnelle.

Or il n'en existe pas, donc la relation acheter est indécomposable, tout comme le type d'associations qu'elle représente.

En toute généralité, les conditions de décomposabilité s'expriment comme suit.

Décomposabilité d'un type d'associations

- Un type d'associations binaire est indécomposable.
- Un type d'associations n-aire est décomposable s'il existe parmi ses rôles une DF dont le déterminant, le déterminé et le résidu comportent chacun au moins un rôle.

3. Et bien sûr s'il existe au moins un attribut en dehors de ces deux sous-ensembles.

Exemples

Étudions deux exemples supplémentaires. Pour chacun, nous traduirons la proposition ternaire source en un type d'associations ternaire que nous décomposerons si nécessaire.

- Les ingénieurs travaillent sur des projets pour le compte de leur service
 Cette proposition se traduit par le type d'associations ternaire travailler (figure 18.5, gauche). La proposition suggère aussi qu'il n'existe qu'un seul service par ingénieur, ce qu'on exprime par la dépendance fonctionnelle entre deux rôles : $\text{INGENIEUR} \longrightarrow \text{SERVICE}$. L'interprétation relationnelle s'écrit comme suit :

travailler(PROJET, INGENIEUR, SERVICE)
 $\text{INGENIEUR} \longrightarrow \text{SERVICE}$

L'identifiant est {PROJET, INGENIEUR}. Le déterminant {INGENIEUR} n'étant pas un identifiant, on en conclut que la relation n'est pas normalisée (section 3.8.3). Par décomposition selon la DF, on obtient :

travailler(PROJET, INGENIEUR)
 dans(INGENIEUR, SERVICE)
 $\text{travailler}[\text{INGENIEUR}] = \text{dans}[\text{INGENIEUR}]$

On en tire le schéma de droite de la figure 18.5. La contrainte de coexistence indique la simultanéité des associations travailler et dans pour chaque ingénieur. On pourrait supprimer cette contrainte si on admettait que *tout ingénieur est attaché à un service*, ce qui serait assez naturel.

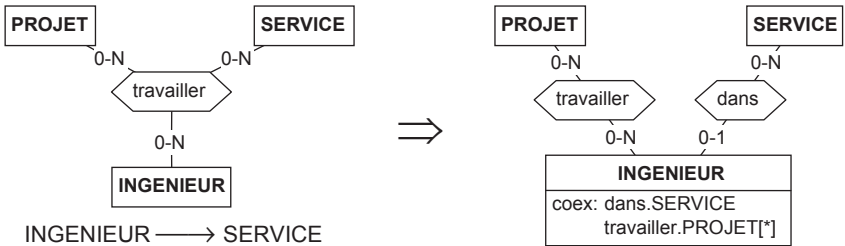


Figure 18.5 - Décomposition d'un type d'associations (1)

- Un contrat est signé par un client pour la couverture d'un véhicule
 Cette deuxième proposition, qui se traduit par le type d'associations ternaire signer (schéma 18.6, gauche), suggère que le contrat est signé par **un** client et qu'il couvre **un** véhicule, d'où la cardinalité 1-1 du rôle signer.CONTRAT. Ce rôle est donc un identifiant implicite de signer. Bien que le type d'associations soit normalisé au sens relationnel du terme (chapitre 3), il est conseillé de le décomposer en deux types d'associations *un-à-plusieurs* (selon figure 16.44), de manière à représenter de manière distincte les deux types de faits « *le contrat est signé par un client* » et « *le contrat couvre un véhicule* ». Une contrainte de coexistence est ici inutile en raison du caractère obligatoire du rôle signer.CONTRAT.

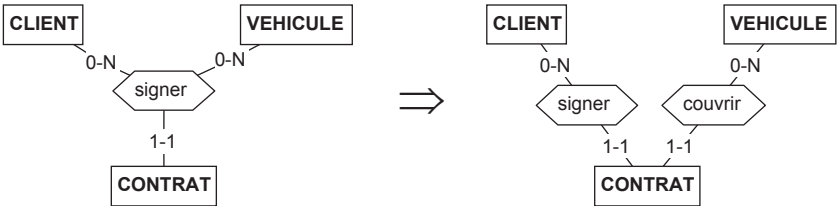


Figure 18.6 - Décomposition d'un type d'associations (2)

18.2.4 Intégration d'une proposition dans un schéma

Le chapitre 13 a montré comment les concepts sous-jacents à une proposition élémentaire étaient représentés et ajoutés au schéma conceptuel en cours de construction. On y suggérait de vérifier la pertinence de la proposition (13.4), on observait que l'ajout des concepts pouvait nécessiter une restructuration de certaines constructions du schéma (13.5.5), on attirait l'attention sur les problèmes de redondance (13.6) et de conflits (13.7). Ces recommandations restent évidemment d'application, mais nous pouvons, dans le présent chapitre, leur donner un cadre de référence à la fois plus général et plus précis.

Les sections qui précèdent ont montré comment tirer d'une proposition un schéma conceptuel élémentaire représentant les concepts exprimés par cette proposition. Nous résumerons l'étape suivante en une seule phrase : *ce schéma élémentaire est ensuite **intégré** au schéma conceptuel en cours de construction*. Or, l'intégration d'un schéma dans un autre, ou plus généralement l'intégration de deux schémas, est un processus que nous étudierons de manière détaillée dans la suite de ce chapitre (section 18.9). Ce processus prenant en charge les problèmes de restructuration, de redondance et de conflit, il constitue une généralisation des recommandations du chapitre 13. À ce stade, nous nous contenterons de traiter un exemple réduit afin d'illustrer intuitivement les principes. Considérons l'état courant du schéma conceptuel dont nous ne retenons que le type d'entités PRODUIT (figure 18.7).

PRODUIT
NumPro
Description
id: NumPro

Figure 18.7 - Schéma conceptuel courant (fragment)

Considérons ensuite le fragment de texte suivant, formé de trois propositions élémentaires dont la représentation (figure 18.8) ne pose pas de problèmes majeurs.

Un produit peut être fabriqué par une unité. Chaque unité de fabrication possède une localisation. Chaque fabrication est caractérisée par ses statistiques annuelles en T/an.

L'intégration du schéma issu de la première proposition au schéma courant se réduit à une simple fusion, qui conduit à ajouter l'attribut Unité au type d'entités PRODUIT (figure 18.9, étape a).

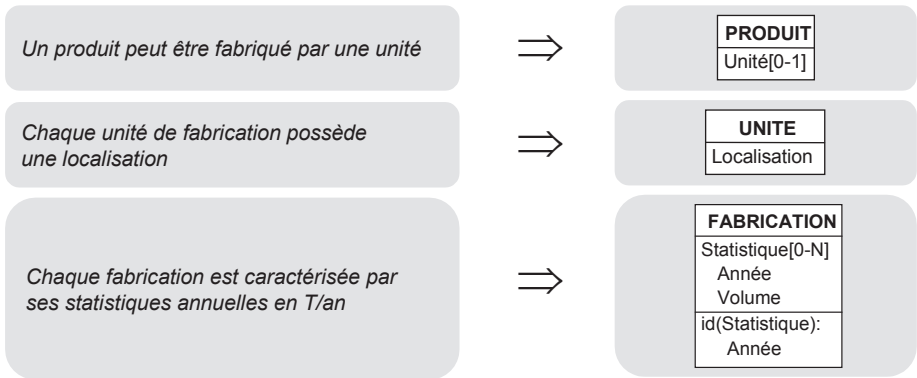


Figure 18.8 - Représentation des trois propositions de l'énoncé

L'intégration du schéma de la deuxième proposition est un peu plus complexe. En effet, on observe une correspondance entre l'attribut Unité et le type d'entités UNITE, identifié par l'attribut Nom Afin d'unifier les deux schémas, l'attribut est transformé en un type d'entités UNITE et un type d'associations fabrique, ce qui permet la fusion des schémas (figure 18.9, étape b). L'intégration du schéma de la troisième proposition est basée sur une correspondance entre le type d'associations fabrique et le type d'entités FABRICATION. L'unification s'obtient par la transformation du premier en un type d'entités FABRICATION (figure 18.9, étape c). On recommandera probablement de normaliser l'attribut complexe Statistique en le transformant en un type d'entités selon les recommandations que nous étudierons à la section 18.7.2.

Le processus d'intégration, et en particulier les opérations de *mise en correspondance*, d'*unification* et de *fusion* seront étudiées plus en détail dans la section 18.9.

18.3 LES PATRONS DE CONCEPTION

Un **patron de conception** (*design pattern* en anglais) est une construction abstraite, dont on trouve fréquemment des **exemples** dans les schémas conceptuels. L'un des premiers patrons que nous avons rencontrés est illustré par le schéma des *ouvrages* et des *exemplaires* (figures 2.11, mais aussi 16.40 et 16.41), dans lequel le premier type d'entités représente un concept abstrait et le second des matérialisations du premier. Le type d'associations de *matérialisation*, ainsi que celui de *composition*, décrits à la section 16.14 sont donc des patrons de conception élémentaires. Les figures 18.10 et 18.11 (schémas de gauche) expriment ces deux constructions de manière abstraite, c'est-à-dire ne retenant que ce que les différents exemples rencontrés ont en commun. Chacun de ces exemples (schémas de droite) constitue une **spécialisation** de ce patron. La figure 16.40 montre que le patron de matérialisation se décline lui-même en deux variantes, selon que l'identifiant du type d'entité matérialisé inclut une dimension {COURS-TYPE, COURS) ou non {FILM, DVD).

Il en existe bien d'autres, certains très généraux, tels que ceux que nous venons de rappeler, et d'autres liés à un domaine d'application. Citons-en quelques-uns.

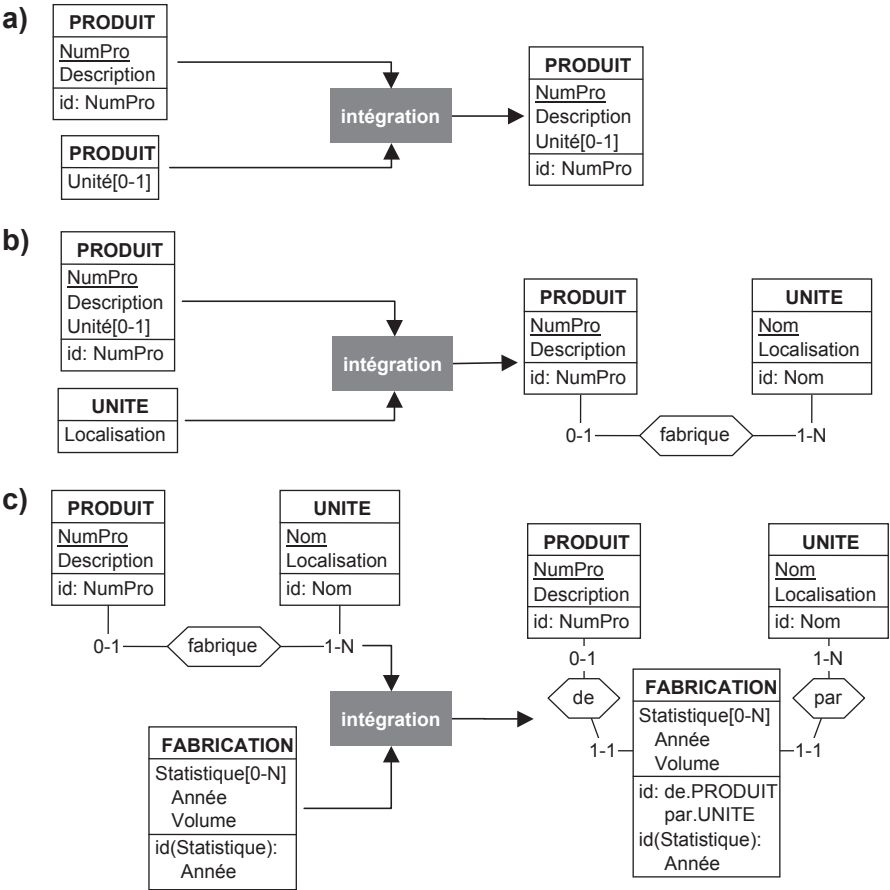


Figure 18.9 - Intégration progressive des trois propositions

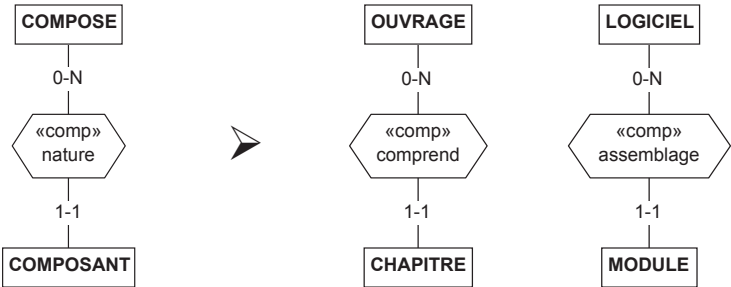


Figure 18.10 - Patron de conception de composition et deux spécialisations

- *Distribution d'un fluide* : patron = {NOEUD, CONDUIT}; quelques spécialisations : distribution d'eau potable (exercice 18.2), égoûts, réseau de gaz, réseau électrique. Ce patron peut être perçu comme étant lui-même une spécialisation

d'un patron plus abstrait, *le graphe*. Lire notamment la section A23.7 de l'annexe A23, *Représentation des graphes*, et ses multiples spécialisations.

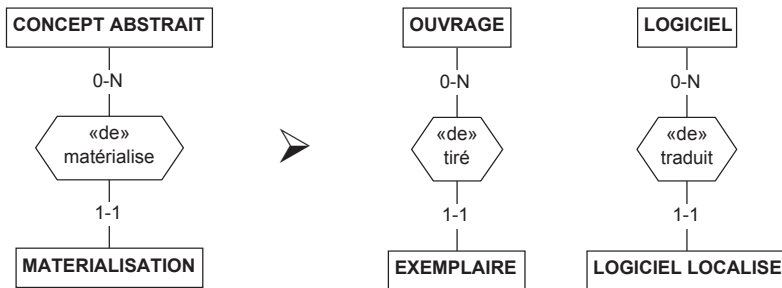


Figure 18.11 - Patron de conception de *matérialisation* et deux spécialisations

- *Objet en mouvement* : patron = {OBJET, DEPLACEMENT}; quelques spécialisations : migration d'un oiseau, course d'une planète, trajet d'une voiture.
- *Echange d'un bien* : patron = {STOCK de départ, STOCK de destination, BIEN}; quelques spécialisations : achat par un détaillant d'un produit à un fournisseur, transfert d'une somme d'un compte bancaire vers un autre, transfert de la clientèle d'une entreprise en cessation vers une autre.
- *Historique* : patron = {OBJET, ETAT}; quelques spécialisations : un compte bancaire et ses états successifs, un patient et son dossier médical, un équipement technique et sa fiche de maintenance.

18.4 ANALYSE DES DOCUMENTS STRUCTURÉS

Si le document textuel relatif aux activités de l'organisation reste un support d'information privilégié durant la phase d'analyse du domaine, il est loin d'être le seul.

Les échanges entre le système d'information et son environnement concernent le plus souvent les objets de ce système. Des agents de l'environnement (un client, un fournisseur, un service administratif) sollicitent le système, généralement via des formulaires papier ou électroniques, pour qu'il exécute des actions (commander un produit, examiner une offre, réclamer une pièce comptable).

Le système répond par l'envoi de documents papier ou électroniques. Il existe donc d'importants échanges d'information dans les deux sens sous la forme de documents structurés. Ces derniers constituent donc une source essentielle de connaissance sur les composants du domaine d'application.

18.4.1 Analyse des documents de l'organisation

Ces documents existent, généralement sous forme informatisée, dès que l'organisation dispose déjà d'un système d'information. L'étude de ce dernier nous apporte des informations essentielles sur le domaine d'application. Lorsqu'un tel système est à créer, une analyse des documents qui sont échangés entre les services, ou avec l'environnement, nous apportera aussi des informations d'une grande importance.

À titre d'exemple, nous examinerons ce qu'une analyse systématique de deux de ces documents peut nous apporter : un **formulaire électronique** d'enregistrement de données (document émis par l'environnement à destination du système) et une **facture** (document émis par le système vers son environnement).

Bien que d'apparence fort différente, l'écran de saisie (figure 18.13) et la facture (figure 18.15) ont des structures abstraites assez similaires. Tous deux sont constitués de champs de données élémentaires, regroupés selon une articulation logique explicitement exprimée par des constructions graphiques intuitives (cadres, ombrages, pavés de couleur), accompagnés de libellés indiquant leur signification, voire de commentaires explicatifs sur leur interprétation. Certains champs de données sont garnis de valeurs qui donnent des indications précieuses sur le type de ces valeurs, sur leur composition, sur les valeurs permises, voire même sur la signification du champ. En résumé, un document d'échange entre le système et son environnement nous renseigne sur les objets importants du domaine via :

- sa *structure* : champs, agrégats de champs, contraintes sur les valeurs ;
- des *renseignements explicatifs* : nom des champs, commentaires, contraintes ;
- des *exemples de valeurs* : type, structure de composition, valeurs admissibles, interprétation.

Nous exprimerons le contenu en information d'un formulaire, qu'il soit sous forme papier ou sous forme électronique, selon une structure arborescente. Il s'agit d'une structure naturelle qu'il est possible de traduire sans effort en XML⁴. D'une manière générale, nous suggérons de représenter chaque document par un type d'entités selon des règles de représentation très simples, telles que celles qui sont décrites à la figure 18.12. Ce tableau pourrait être complété par d'autres objets graphiques tels que les cases à cocher et les boutons radio, dont l'interprétation est assez évidente.

Composant du document	Interprétation
volet indépendant	type d'entités
champ élémentaire	attribut atomique monovalué
cadre multi-champs (ou bloc)	attribut composé
grille multi-colonnes	attribut composé
grille multi-lignes	attribut multivalué
grille multi-colonnes et multi-lignes	attribut multivalué composé

Figure 18.12 - Conversion d'un document en structures de données

Remarque

Les techniques mises en œuvre pour analyser et interpréter la structure des documents relèvent de la *rétro-ingénierie*, qui sera étudiée plus en détail au chapitre 22.

4. C'est d'ailleurs sous cette forme, qui constitue une variante *dématérialisée* des documents standard, que les applications informatiques s'échangent généralement des données.

18.4.2 Analyse d'un écran de saisie de données

Considérons à titre de premier exemple un écran de saisie et de modification de données relatif à une application de gestion de formations professionnelles (figure 18.13). Via cette interface, l'opérateur va enregistrer l'inscription d'un client à une session d'un séminaire, remplacer ce client par un autre (suppléant) en cas de désistement, désinscrire le client et contrôler la facturation et le paiement.

Figure 18.13 - Écran de saisie et de gestion des inscriptions à des séminaires de formation

Le contenu en information de cette interface correspond à une structure de données arborescente à trois niveaux. Le premier niveau est constitué d'un bloc Session, d'un bloc Participant, de divers champs élémentaires (Statut, Prix spécial, ..., Présence, Remarque) puis d'un bloc Désinscription. Chacun des blocs est à son tour constitué de champs et/ou de blocs.

La traduction de cette structure en un type d'entités est représentée à la figure 18.14 (schéma de gauche). Les composants de niveau 1 ont été traduits dans les attributs composés Session, Participant et Désinscription, et dans les attributs atomiques Statut, Prix spécial, ... et Présence (leur ordre est ici indifférent). L'attribut Session est lui-même constitué de l'attribut composé Séminaire et des attributs monovalués et atomiques Code et Date. Le bloc Suppléants du bloc Participant est interprété comme un attribut multivalué composé.

Le résultat repris à la figure 18.14 (gauche) n'est pas un schéma conceptuel au sens strict mais bien une *vue* présentée à l'utilisateur externe. Les données qui y sont représentées ne correspondent pas nécessairement au contenu de la base de données (existante ou future). On y trouvera des données en double, des données calculées, extraites d'autres sources ou qui ne sont pas destinées à être enregistrées dans la base de données. D'autre part, l'analyse d'une telle interface ne permet pas d'obtenir certaines contraintes d'intégrité, telles que tous les attributs obligatoires, les cardinalités exactes, les identifiants ou les contraintes référentielles. Nous devons donc compléter cette analyse par des informations issues d'autres sources.

Il est cependant possible de tirer de cette structure arborescente un schéma conceptuel qui représente de manière explicite les concepts sous-jacents et leurs

caractéristiques. Le schéma de droite de la figure 18.14 a été obtenu à partir de celui de gauche par la transformation des attributs complexes en types d'entités. Nous y avons également ajouté quelques contraintes importantes, telles que les identifiants, qui sont absentes du schéma initial.

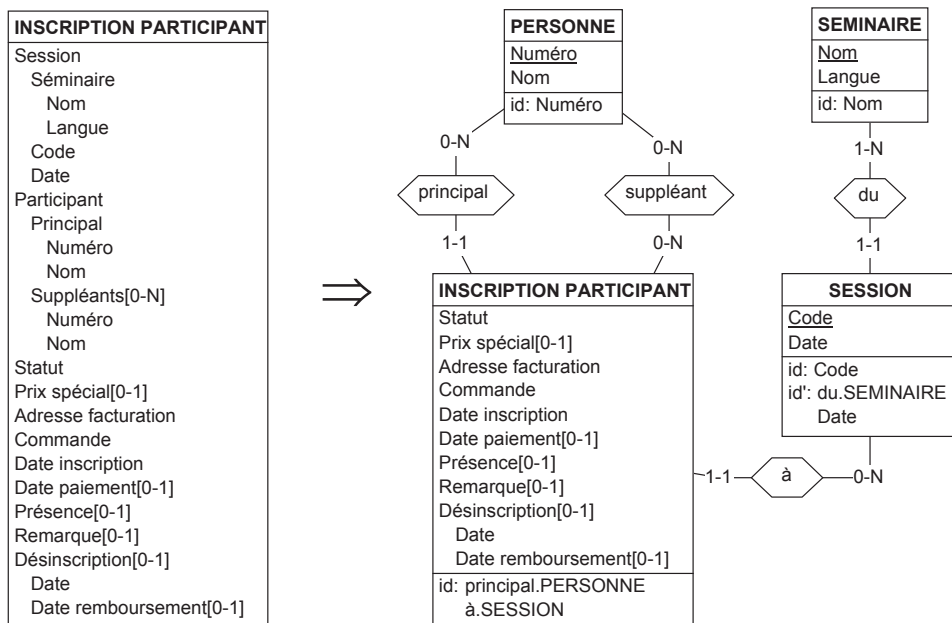


Figure 18.14 - Structure de données brute (gauche) et conceptualisation (droite) du contenu informationnel de l'écran de la figure 18.13

Nous avons aussi interprété certaines structures selon notre compréhension du problème : c'est ainsi que nous avons considéré que Principal et Suppléants étaient des Personnes, ce que le schéma de gauche n'exprime pas explicitement. Le schéma de droite résulte de la **conceptualisation** de la structure de données brute.

18.4.3 Analyse d'un document papier

Examinons attentivement la facture représentée à la figure 18.15, visiblement produite par informatique. Pour la rendre lisible, on lui a donné une structure graphique qui permet au destinataire d'identifier clairement des groupes de données. La présence de valeurs inscrites dans les champs du formulaire entraîne deux nouveautés par rapport à l'analyse de l'écran :

- certaines données ne sont pas accompagnées d'un libellé explicatif ; c'est la valeur elle-même qui en suggérera la signification (les coordonnées de l'entreprise par exemple, dans le coin supérieur gauche) ;
- la structure de certaines données suggère une décomposition significative que la disposition graphique ne permet pas de deviner ; par exemple, la valeur « G. Desmedt, rue Garamond 104 Verviers » attachée au champ Client montre clairement une décomposition (Nom, Adresse).

L'analyse des données permet également d'inférer avec une bonne probabilité les attributs calculés (Répartition TVA[*].Montant TVA, Total TVA, etc.) et les identifiants d'attribut (Répartition TVA par exemple).

N.V. BELIMPORT S.A. 123, rue Neuve - 6743 Bertelmont TVA BE 546.901.442 - 059/45.56.67		N° 0030127 Date 24-07-2009	
Client G. Desmedt, rue Garamond 104 Verviers			
N° TVA		Num. Cmde	

DESCRIPTION	CODE	QTE	%TVA	PRIX TVAC	REMISE	TOTAL
BOOKMAN BS 620	NF-00453-34	1	21	65	0	65
Aqua Crystal 12L	FF-10067-01	5	6	25	0	25
Cafe COLOMBO 22	FF-77029-07	12	6	23,2	0	23,2
<i>Dispense de certification et de signature par autorisation 1645 du 04/08/1995. Exonération ecotaxe (Art 372, 373, 378)</i>						

Total montants TVA et marchandise HTVA par taux TVA

TVA 21%	TVA 6%	TVA	TVA	Tot. TVA	Tot. March.	TOTAL
11,28	8,37			19,65	93,55	113,2
Marchandise	Marchandise	Marchandise	Marchandise			
53,72	39,83					
						Reste à payer
						0

Figure 18.15 - Une facture de l'entreprise BELIMPORT

Le résultat de l'analyse de cette facture est exprimé dans le schéma de gauche de la figure 18.16. Sa conceptualisation est proposée à droite de la même figure.

18.5 DÉNOMINATION DES OBJETS D'UN SCHÉMA

La section 16.13 a proposé différentes variantes de règles systématiques de formation des noms à assigner aux différents objets du schéma et notamment de leur forme. Il est aussi important d'observer une stricte régularité dans la structure des noms d'un même schéma, de manière à améliorer la lisibilité du schéma et, par là, sa maintenabilité.

Lorsque plusieurs concepts similaires apparaissent dans un schéma, convient-il de leur donner le même nom ou au contraire des noms distincts ? Le cas le plus fréquent est celui des attributs. Considérons par exemple les types d'entités EMPLOYE et CLIENT, tous deux dotés d'un attribut Téléphone. Deux approches sont fréquemment adoptées.

- **Les attributs similaires reçoivent le même nom.** On les distingue selon le nom de leur parent : EMPLOYE.Téléphone et CLIENT.Téléphone.

- **Les attributs du schéma reçoivent des noms distincts.** Les attributs évoqués seront par exemple dénommés respectivement TéléphoneEmployé et Téléphone-Client.

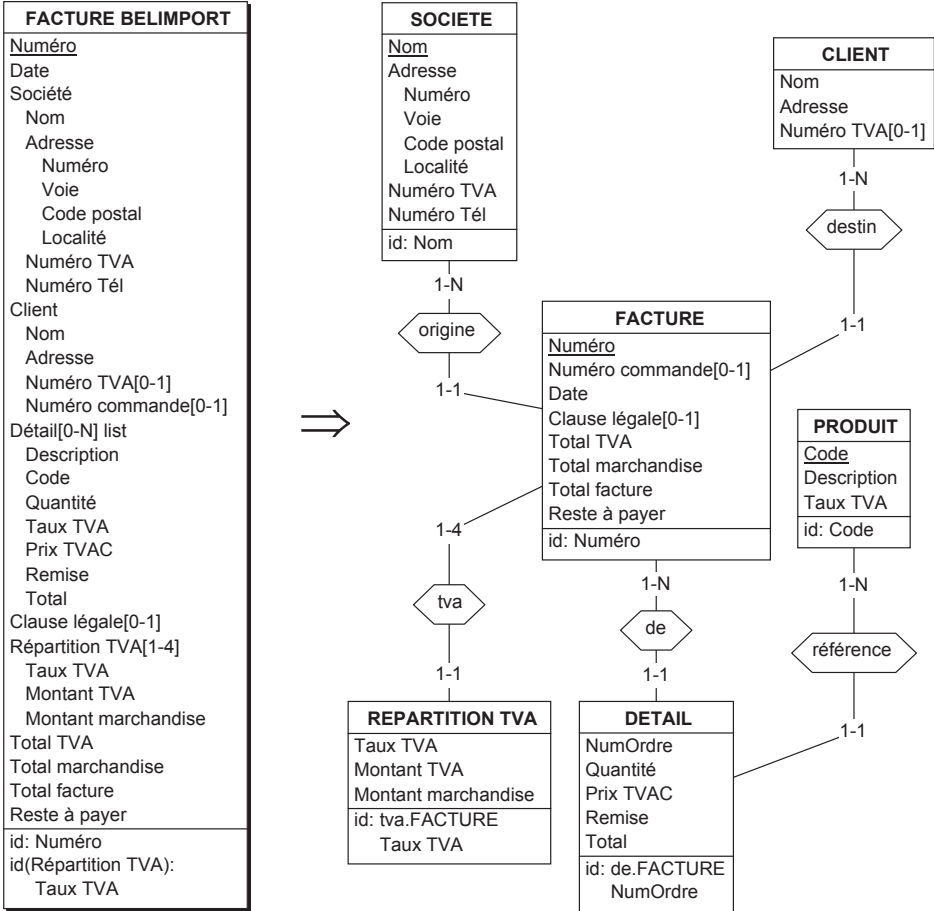


Figure 18.16 - Structure de données brute (gauche) et conceptualisation (droite) du contenu informationnel de la facture de la figure 18.15

18.6 PROCESSUS DE CORRECTION D'UN SCHÉMA

Un schéma conceptuel est un document généralement volumineux et complexe, et à ce titre susceptible de contenir des **défauts**. On considérera trois types de défauts, que nous analyserons successivement dans les sections suivantes :

1. **les erreurs de schéma** Il s'agit d'un type d'erreurs identifiable formellement et qu'il est indispensable de corriger avant de procéder à la normalisation et à la validation. Linguistiquement, ces erreurs correspondent à des **erreurs syntaxiques**. Les erreurs de schéma sont étudiées dans la présente section.

2. Les **maladresses de formulation**, selon lesquelles des constructions du schéma, par ailleurs correct, sont mal exprimées. Linguistiquement, ces erreurs correspondent à des *maladresse de style*. Ces erreurs seront traitées dans la section 18.7, consacrée à la *normalisation* du schéma.
3. Les **erreurs sémantiques**, qui font que le schéma ne représente pas correctement les exigences des utilisateurs. Linguistiquement, ces erreurs correspondent à des *erreurs de fond*. Elles seront traitées à la section 18.8, consacrée à l'*évaluation* du schéma.

Parmi les erreurs de schéma, on distinguera trois phénomènes distincts : les constructions *syntactiquement incorrectes*, les constructions *non satisfiables* et les constructions *incohérentes*.⁵

18.6.1 Constructions syntaxiquement incorrectes

Une erreur syntaxique affecte une construction dont la formulation ne respecte pas les règles de constitution propres au modèle choisi (chapitres 12 et 16). En principe, ce type d'erreurs est plutôt rare lorsque le schéma est construit à l'aide d'un AGL, lequel effectue au vol ou à la demande de l'analyste des vérifications de forme. Des erreurs peuvent cependant subsister suite à une utilisation abusive de l'atelier, au laxisme de ce dernier ou lorsque l'analyste utilise pour construire son schéma un logiciel de dessin généraliste tel que *Visio* ou *Powerpoint*. Le schéma de la figure 18.17 comporte quelques erreurs syntaxiques classiques :

- le type d'associations affilié n'a qu'un seul rôle, qui d'ailleurs est affecté d'une contrainte de cardinalité incorrecte,
- deux types d'entités portent le même nom,
- la composition de l'identifiant de ORDRE est invalide (pas d'attribut multi-valué dans un identifiant multi-composants),
- la contrainte de coexistence de TITRE (droite) comporte un composant obligatoire,
- l'identifiant primaire de ORDRE comporte un composant facultatif,
- l'identifiant de acheter est invalide (doit inclure au moins un rôle).

Il existe bien d'autres erreurs de ce type qui dérivent du non respect des propriétés du modèle utilisé. Le relecture attentive du chapitre 16 permettra d'en repérer un grand nombre. La figure 18.18 en suggère trois, liées aux structures de relations *is-a*.

- *Schéma de gauche* : un type d'entités ne peut être un sous-type de lui-même ; en d'autres termes, le graphe des relations *is-a* ne peut comprendre de circuits⁶.

5. Sur la détection et la correction des erreurs de cohérence, on consultera par exemple [Hartmann, 2001].

6. Curieusement, les types d'entités de ce schéma sont parfaitement *satisfiables* dès qu'on admet que les trois types d'entités sont identiques : mêmes composants et même population.

a) Sous-types non satisfiables

Le problème du sous-type vide par construction a été décrit à la section 16.9.5. Il peut se poser dès qu'un type d'entités possède plus d'un surtype. La figure 18.19, qui est un simple exemple de la figure 16.21, illustre les deux cas représentatifs. Dans le premier schéma EPARGNE-ASSURANCE est non satisfiable en raison du principe selon lequel les populations de deux types d'entités sans surtype commun sont *disjointes par définition*. Tout sous-type qui leur serait commun serait non satisfiable. Le second schéma pose le même problème, les deux sous-types VEHICULE TERRESTRE et VEHICULE AQUATIQUE étant *disjoints par déclaration*.

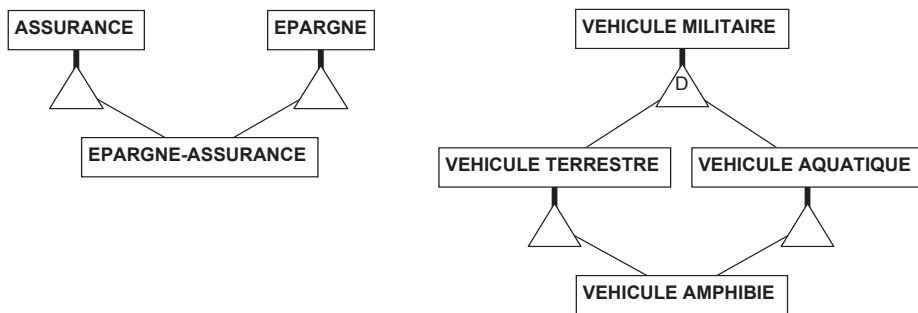


Figure 18.19 - Les sous-types EPARGNE-ASSURANCE et VEHICULE AMPHIBIE sont non satisfiables

La question de la satisfiabilité dans les hiérarchies *is-a* a été étudiée par plusieurs auteurs. Voir [Balaban, 2006] par exemple comme point d'entrée.

Correction

Sauf dans des cas très particuliers⁸, une construction non satisfiable résulte d'une erreur d'analyse à corriger. Les types d'entités non satisfiables sont assez faciles à identifier, selon les règles de détection qui sont décrites à l'annexe A18.

b) Contraintes d'existence non satisfiables

Considérons un type d'entités auquel sont associés des composants facultatifs (attributs et/ou rôles) et faisant l'objet de contraintes d'existence (section 16.11.3). Des contraintes mal définies peuvent conduire à l'impossibilité de donner une valeur à certains de ces composants ou même à rendre impossible la création d'une entité ou d'une association.

Dans le premier cas (figure 18.20, gauche), les composants A2 et A3 sont non satisfiables, et doivent dans tous les cas de figure être sans valeur.

8. Par exemple, lorsqu'il est prévu que le schéma évolue dans l'avenir et que la contrainte aujourd'hui excessive soit relâchée plus tard, permettant alors la création d'entités du type non satisfiable.

Dans le second cas (figure 18.20, droite), c'est le type d'entités tout entier qui est non satisfiable. Il n'existe en effet aucun ensemble de valeurs ou absence de valeur pour B2, B3 et B4 satisfaisant simultanément les trois contraintes d'existence.

A	B
A1	B1
A2[0-1]	B2[0-1]
A3[0-1]	B3[0-1]
coex: A2	B4[0-1]
A3	coex: B2
excl: A2	B4
A3	exact-1: B2
	B3
	B4
	if: B3
	B4

Figure 18.20 - Contraintes d'existence non satisfiables

Des règles de détection de ce type d'incohérence sont proposées à l'annexe A18.

c) Cardinalités non satisfiables dans un cycle de types d'associations

Certaines valeurs des contraintes de cardinalité de types d'associations rendent impossible la création d'entités impliquées dans des associations de ces types. Le schéma de la figure 18.21 illustre le phénomène.

En appliquant les règles de calcul des populations des types d'entités vues à la section 16.8.6, on tire les observations suivantes :

- via S, il y a au moins autant d'entités A que d'entités B
- via R, il y a au moins deux fois plus d'entités B que d'entités A
- en combinant ces deux relations, on conclut qu'il y a au moins **deux fois plus d'entités A que d'entités A**.

La seule manière de satisfaire cette conclusion est de garantir que la population de A soit toujours vide. Il en sera de même de la population de B.

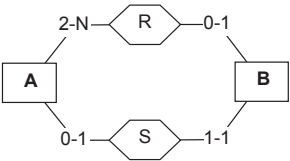


Figure 18.21 - Cardinalités non satisfiables

La figure 18.22 propose un autre exemple de construction non satisfiable. On y indique que chaque membre du personnel a pour suppléant un expert, lequel est lui-même un membre du personnel. Il y a donc *au moins autant d'experts que de membres du personnel*, comme EXPERT est un sous-type de PERSONNEL, il n'y a *pas plus d'experts que de membres du personnel*. De ces deux inégalités, on tire que

les populations de PERSONNEL et d'EXPERT sont les mêmes, et donc que la population d'EMPLOYE est structurellement vide.

Des règles de détection de ce type de constructions sont proposées à l'annexe A18.

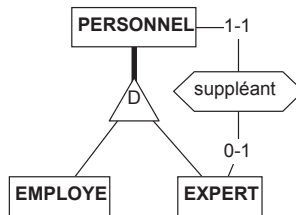


Figure 18.22 - Problème collatéral pour le type d'entités EMPLOYE

Mentionnons une variante faible de la non-satisfiabilité : la **non-initialisabilité**. La figure 18.23 reprend une structure que nous avons souvent utilisée, à ceci près que nous imposons ici que *toute personne ait un supérieur* (cardinalité [1-1]). Si à ce schéma doit, à tout instant, correspondre un graphe des instances acyclique (une personne ne peut être son propre responsable, ni directement ni indirectement), alors le type PERSONNE est *non initialisable*. Il est en effet impossible d'introduire la première entité PERSONNE puisque cette opération suppose que l'entité de son supérieur (qui lui est différente) soit déjà présente. En revanche, PERSONNE admet des instances valides et est donc satisfiable, même si on peut se demander comment la première a été créée⁹ ! En revanche, si on relâche la contrainte d'acyclicité, et si on admet, au minimum, que certaines personnes puissent être leur propre supérieur, alors PERSONNE devient satisfiable et initialisable.

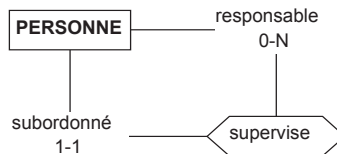


Figure 18.23 - Structure hiérarchique problématique

On trouvera dans l'annexe A18, à titre d'exercice, d'autres schémas présentant les mêmes anomalies d'initialisabilité. Les problèmes de non-satisfiabilité dus aux cardinalités ont été assez bien étudiés dans la littérature. On citera en particulier [Thalheim, 1992] et [Boufares, 2004].

9. La reconnaissance de cette *anomalie utile* et sa résolution sont très anciennes, puisqu'on les trouve dans les premiers SGBD CODASYL (début des années 70 mais toujours utilisés aujourd'hui), sous la forme des types d'associations (*set types*) dont un rôle est *mandatory manual*. On en trouvera une description sur le site de l'ouvrage. L'adaptation de cette solution simple et élégante au mécanisme des clés étrangères aurait été très facile mais SQL ne l'a malheureusement pas adoptée, forçant le programmeur à recourir à des techniques complexes basées sur des transactions, des méthodes de tables typées ou des procédures SQL, qui seront décrites à la section 21.7.

d) Contraintes d'inclusion incohérentes

La section 3.5 a montré que les contraintes d'inclusion avaient la propriété de transmettre d'une relation vers une autre des contraintes telles que les dépendances fonctionnelles et certaines contraintes d'inclusion. Ces contraintes *héritées* peuvent entrer en conflit avec des contraintes locales.

La figure 18.24 illustre deux situations typiques. Dans le schéma de gauche, la projection de assignée sur les rôles FOURNISSEUR et PIECE est clairement *plusieurs-à-plusieurs*. L'ensemble de couples qui lui correspond ne peut en toute généralité être une partie de la population de offre, qui est *un-à-plusieurs*. De même, le schéma de droite, dont la contrainte d'inclusion exprime le fait que *le directeur d'un service est un de ses employés*, posera un problème lorsqu'il s'agira de préciser le directeur d'un service qui n'a pas d'employé¹⁰.

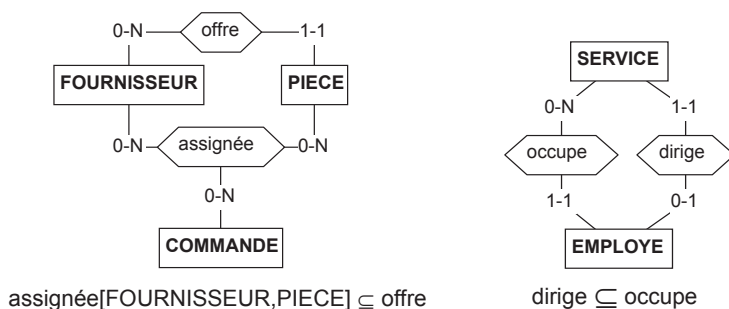


Figure 18.24 - Deux structures incohérentes

18.7 PROCESSUS DE NORMALISATION DU SCHÉMA CONCEPTUEL

Un schéma correct doit encore satisfaire des critères de forme qui lui conféreront des qualités essentielles telles que la facilité d'utilisation des (futurs) données, la lisibilité, l'expressivité, la concision, la clarté ou l'évolutivité. On dira d'une construction qui respecte ces critères qu'elle est **normalisée**. S'assurer qu'un schéma respecte ces critères et le corriger si tel n'est pas le cas est un processus bien identifié appelé **normalisation**.

Une construction non normalisée n'est pas erronée, au sens défini dans la section 18.6, mais elle ne correspond cependant pas aux *bonnes pratiques* recommandées *par la profession*. Elle est, dans un certain sens, *non optimale* ou *maladroite*. On mettra ici en cause, non sa sémantique, mais sa forme ou son style¹¹. Normaliser une

10. Le lecteur attentif se souviendra peut-être que ce cas a déjà été rencontré sous une autre forme à la section 13.7, comme illustration de constructions contradictoires.

11. On appellera *style* d'un schéma l'ensemble des raisonnements systématiques qui président au choix des règles de représentation des types de faits. Un style est propre à un concepteur (on reconnaît un auteur à son style), à une méthode ou à une école de pensée.

construction consistera à améliorer cette forme sans modifier ce qu'elle exprime.

Nous avons déjà rencontré le concept de *normalisation* au chapitre 3. Il s'agissait alors de transformer une relation de manière que chaque fait y soit représenté une et une seule fois. Une relation non normalisée n'est pas sémantiquement incorrecte, mais elle pose des problèmes de *lisibilité* (deux types de faits dans une même structure), de *gestion* des données (mises à jour complexes), de *place* occupée (redondance) et d'*évolutivité* (il est délicat d'introduire la représentation de nouveaux types de faits).

Un schéma conceptuel est sujet à des difficultés du même type, qui seront résolues également par des transformations qui amélioreront les qualités recherchées que nous venons de mentionner. La méthode simplifiée décrite au chapitre 13 comportait un processus *Normalisation du schéma* (section 13.9) conduisant à une simplification du schéma ainsi qu'à l'élimination de redondances internes. Nous allons, dans la suite de cette section, approfondir la notion de normalisation en considérant aussi d'autres critères ainsi que d'autres transformations de schéma.

On admet que la principale préoccupation est la **lisibilité**, parfois appelée **compréhensibilité**. Cette qualité, qui n'est pas aisée à définir avec précision, mesure l'efficacité avec laquelle le schéma transmet l'intention de son auteur (la sémantique externe du schéma) à un lecteur quelconque, de compétence moyenne. Cette qualité est en effet essentielle dans tous les processus qui se basent sur l'exploitation d'un schéma de base de données : programmation, diffusion des données, administration des données, maintenance de la base de données, évolution de la base de données, etc. Comment en effet utiliser et manipuler de manière judicieuse des données dont on ne comprend pas parfaitement la signification et les contraintes qui leur sont imposées ?

On admet aussi que la qualité de lisibilité est favorisée par un nombre limité de caractéristiques formelles (donc identifiables avec une certaine précision, voire mesurables). Nous en retiendrons sept, que nous allons décrire brièvement dans cette section et qui seront traitées de manière plus détaillée à l'annexe A18.

- **simplicité** (ou **minimalité**) : un type de faits doit être représenté le plus simplement possible ; il est inutile par exemple de représenter une simple propriété par un type d'entités (18.7.1) ;
- **expressivité** : un type de faits doit être représenté par une construction qui évoque naturellement et clairement sa nature ; c'est ainsi par exemple qu'on utilisera une relation *is-a* pour indiquer qu'un type d'entités est un cas particulier d'un autre (18.7.2) ;
- **absence de redondance** : un schéma doit exprimer un type de faits une et une seule fois ; il en est de même des instances futures ou existantes des constructions ; on reconnaîtra ici une qualité similaire à celle qui est recommandée pour les schémas relationnels au chapitre 3 (18.7.3 et 18.7.4) ;
- **autres critères de qualité** : on évoquera sans les détailler d'autres critères de qualité tels que l'*absence de sur-spécifications*, l'*absence de construction anormales*, la *régularité d'expression*, la *qualité graphique* et le *respect des standards*.

La normalisation sera sans doute un processus léger, voire superflu pour un schéma de petite taille élaboré par un analyste expérimenté. En revanche, il s'avérera indispensable lorsqu'un schéma conceptuel complexe est construit en plusieurs étapes par des concepteurs différents, à différentes époques, ou encore lorsque les schémas sont produits de manière semi-automatique, notamment lors de l'extraction à partir de formulaires, d'écrans ou lors de l'analyse automatique de texte. Il constituera enfin un processus essentiel en rétro-ingénierie, lorsque les schémas extraits sont fortement *colorés* (ou *pollués*) par les caractéristiques des SGBD sources ou par la maladie des générations de concepteurs qui sont intervenus sur ces schémas au cours du temps.

18.7.1 Simplification des constructions non minimales

Une construction non minimale traduit un type de faits pour lequel il existe une représentation plus légère généralement jugée plus appropriée. Un trop grand nombre de ces constructions risque de rendre la lecture du schéma plus laborieuse et donc d'affecter négativement sa maintenabilité. C'est à l'analyste de juger de l'opportunité de corriger ce problème. Attention cependant, un excès de correction peut conduire à un défaut d'expressivité ou peut contrarier l'évolutivité du schéma. En effet, une construction excessive peut n'être que temporaire. Par exemple, un *type d'entités attribut* pourrait recevoir d'autres attributs dans un avenir proche.

a) Type d'entités attribut

Le *type d'entités attribut* est une construction que nous avons rencontrée à plusieurs occasions, et en particulier à la section 13.9.1, figure 13.22. Il a pour objectif de *promouvoir* un attribut à un statut supérieur, notamment en vue de l'intégrer dans un réseau de relations avec d'autres concepts ou d'affiner sa description en lui associant des attributs complémentaires. La figure 18.25 en montre deux autres exemples.

Dans le schéma supérieur, PRENOM est considéré comme un *type d'entités attribut* dans la mesure où son rôle se limite apparemment à associer de 0 à 4 valeurs de prénoms aux entités ELEVE. On notera le caractère obligatoire du rôle opposé qui lie l'existence de chaque entité PRENOM à celle d'au moins une entité ELEVE. On notera aussi l'identifiant de PRENOM, qui exprime de fait que chaque entité représente une **valeur** distincte de prénom.

Le schéma inférieur exprime exactement la même sémantique externe. Cependant, on y représente par une entité PRENOM D'ELEVE, non pas une valeur distincte, mais une **instance** de cette valeur. Le lecteur se reportera aux figures 13.8 et 13.9 qui décrivent clairement les différences entre ces deux représentations.

On suggère de transformer les types d'entités PRENOM et PRENOM D'ELEVE en un attribut Prénom qui exprime plus clairement l'intention de ces constructions.

Cette construction n'est pas limitée à la représentation de propriétés multiples, mais peut être généralisée à tout type d'attribut. La figure 18.26 montre son application à des attributs monovalués.

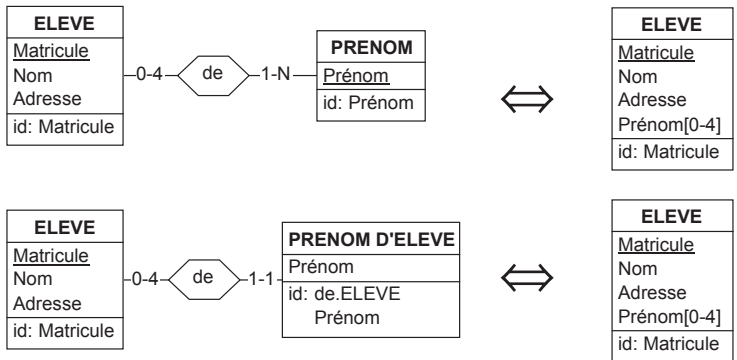


Figure 18.25 - Les types d'entités attribut PRENOM et PRENOM D'ELEVE sont chacun transformés en un attribut multivalué Prénom

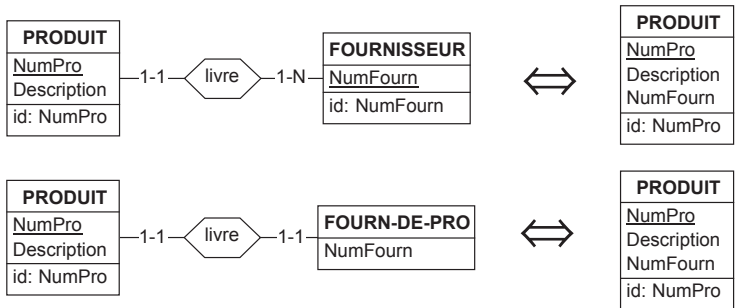


Figure 18.26 - Types d'entités attribut transformés en attributs monovalués

Règle

Les différentes configurations qui définissent les types d'entités attributs sont décrites à l'annexe A25, section A25.8.2. En particulier, on soupçonnera l'existence d'un *type d'entités attribut* EA s'il vérifie notamment les conditions suivantes :

1. EA apparaît dans un et un seul type d'associations R, binaire, non cyclique,
2. les rôles de R sont non polymorphiques, le rôle R.EA est obligatoire,
3. EA possède au moins un attribut,
4. EA n'appartient pas à une hiérarchie *is-a*,
5. EA possède au plus un identifiant,
6. la fonction de EA est clairement d'affecter ses attributs à E, jouant l'autre rôle dans R.

On envisagera alors la transformation de EA en attributs de E.

b) Type d'associations n-aire avec rôle [1-1] ou [0-1]

Un premier exemple a été proposé à la figure 18.6. La figure 18.27 (gauche) en propose une variante caractérisée par un rôle facultatif. La décomposition produit une contrainte de coexistence (schéma central) qui elle-même peut être éliminée par un sous-type selon le procédé discuté à la section 18.7.2. Il est clair que la décomposition améliore la lisibilité dans le cas d'un rôle obligatoire. Lorsque le rôle est facultatif, c'est à l'analyste de décider de l'utilité de procéder à une décomposition.

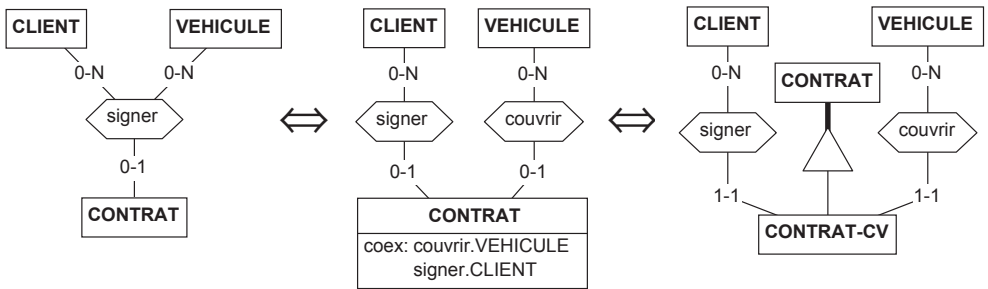


Figure 18.27 - Simplification d'un type d'associations n-aire

c) Attribut composé d'un seul composant

Au terme de l'analyse, il n'est pas rare qu'un attribut initialement considéré comme composé se retrouve au final avec un seul composant. Son remplacement par ce dernier simplifiera le schéma sans perte d'information¹² (figure 18.28).

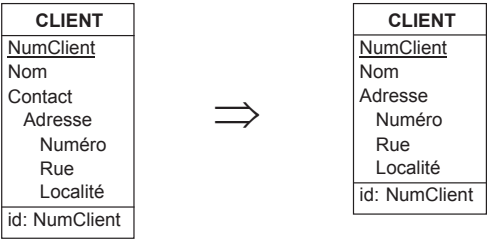


Figure 18.28 - Désagrégation d'un attribut à un seul composant

d) Type d'associations un-à-un à rôles obligatoires

Un type d'entités *un-à-un* dont les rôles sont obligatoires définit une liaison très forte entre ses deux types d'entités, à tel point qu'on pourrait se demander si ceux-ci ne constituent pas en réalité deux fragments d'un même type d'entités. Il serait alors sans doute recommandé de les intégrer (figure 18.29).

12. On remarque que la figure est moins affirmative sur ce point. On y utilise en effet le symbole $\Rightarrow$ au lieu de l'équivalence $\Leftrightarrow$. Cette prudence est justifiée par la perte de la dénomination « Contact » dont l'importance doit être évaluée par l'analyste.



Figure 18.29 - Un type d'entités fragmenté

e) Sous-type faiblement spécifique

Si un sous-type ne se distingue de son surtype que par une seule caractéristique obligatoire (un attribut, un rôle [1-j]), on peut envisager de le fusionner avec son surtype, la caractéristique y devenant facultative. Dans la figure 18.30, ARTICLE COMMANDE (lire *article commandé*) représente les articles référencés par une commande. Le schéma de droite, plus simple, n'entraîne pas de perte d'information.

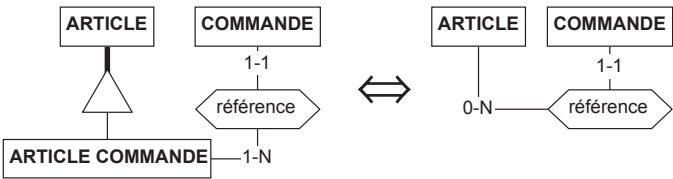


Figure 18.30 - Suppression d'un sous-type faiblement spécifique

f) Type d'entités doté d'un seul sous-type sans attributs ni rôle

De même, un type d'entités dont on avait pressenti qu'il devait posséder des sous-types peut en définitive se retrouver avec une descendance réduite qui ne justifie plus l'usage d'une relation *is-a*.

La figure 18.31 montre la représentation d'un sous-type unique sans rôle ni attribut par un simple booléen. Il s'agit d'un cas limite de sous-type faiblement spécifique.

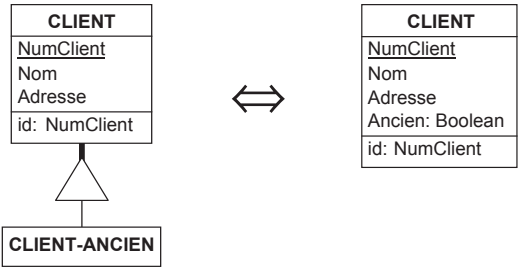


Figure 18.31 - Simplification d'un sous-type élémentaire

18.7.2 Explicitation des constructions insuffisamment expressives

Une construction *insuffisamment expressive* traduit un type de faits pour lequel il existe une représentation plus appropriée, qui évoque naturellement et clairement la nature de celui-ci. Elle trouve souvent son origine dans l'analyse de formulaires dont la structure est trop pauvre (listes, arbres) pour représenter explicitement les relations entre les données. Une telle construction n'est pas une erreur à proprement parler, mais elle risque d'induire des difficultés de compréhension du schéma, et donc d'affecter négativement son utilisabilité et sa maintenabilité.

a) Type d'entités association

Cette construction a souvent été rencontrée, par exemple à la section 13.9.1, figure 13.23. Considérons encore la figure 18.32, où ECRITURE est considéré comme un *type d'entités association* dans la mesure où son rôle se limite à représenter des associations entre les entités OUVRAGE et AUTEUR. On suggère de le transformer en un type d'associations écrit qui exprime mieux l'intention de cette construction.

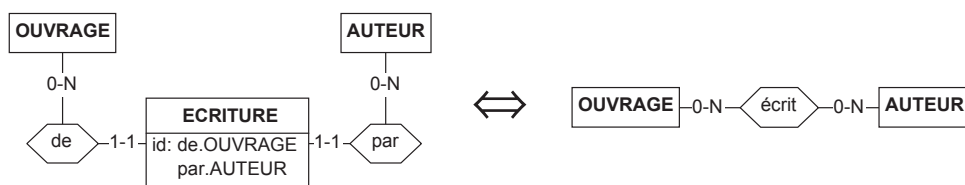


Figure 18.32 - Un type d'entités association transformé en un type d'associations

L'exemple de la figure 18.33 est un peu plus complexe. Le type d'entités incriminé associe trois types d'entités, possède des attributs et un identifiant hybride. Il peut être transformé en un type d'associations ternaire emprunte qui hérite de ces attributs et de cet identifiant.

La décision de transformer de tels types d'entités est de la responsabilité de l'analyste. En effet, si le schéma devient plus expressif, il pourrait en revanche s'avérer moins évolutif.

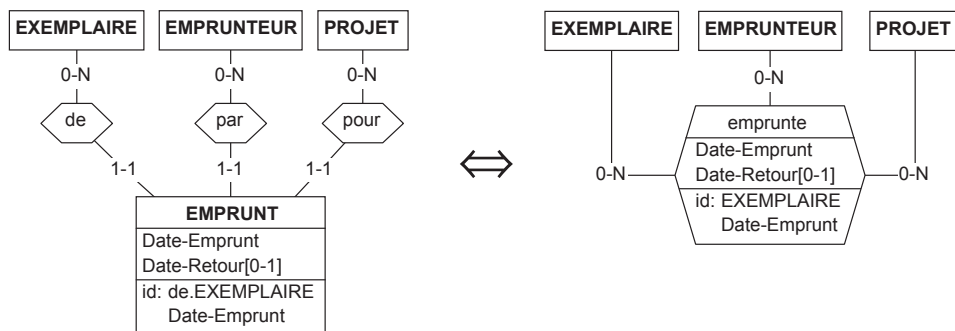


Figure 18.33 - Un type d'entités association transformé en un type d'associations ternaire

Si un *type d'entités association* transformé en un type d'associations devait dans l'avenir jouer d'autres rôles ou s'inscrire dans une hiérarchie de relations *is-a*, il faudrait alors lui rendre sa forme primitive.

Règle

Les critères d'identification des types d'entités association sont décrites à l'annexe A25, section A25.8.1. En particulier, on soupçonnera l'existence d'un *type d'entités association* ER s'il vérifie notamment les conditions suivantes :

1. ER joue au moins deux rôles, appartenant tous à des types d'associations fonctionnels non cycliques,
2. chacun de ces rôles est de cardinalité [1-1] et est non polymorphique,
3. chacun des identifiants explicites ou implicites de ER comporte au moins un rôle opposé,
4. ER n'appartient pas à une hiérarchie *is-a* et
5. la fonction de ER est clairement d'associer tous les types d'entités jouant les rôles opposés.

On envisagera alors la transformation de ER en un type d'associations.

b) Attribut complexe

Un attribut composé et multivalué, surtout s'il est soumis à des contraintes propres (identifiant par exemple), peut cacher un concept important du domaine d'application. Il y a lieu, dans ce cas, d'envisager sa transformation en un type d'entités qui représente explicitement ce concept. L'attribut *Détail* du type d'entités *COMMANDE* (figure 18.34) en est certainement un exemple. Le type d'entités qui dérive de sa transformation exprime de manière plus claire la nature de ce concept.

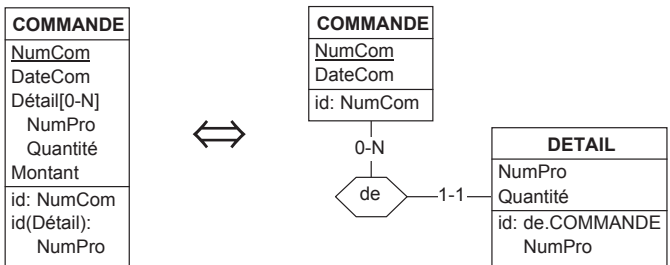


Figure 18.34 - Expression d'un attribut complexe sous la forme d'un type d'entités

c) Attribut de référence

Une valeur d'un attribut de référence a pour but de désigner une entité liée à l'entité source. Un tel attribut agit comme une clé étrangère dans un schéma relationnel. Il s'agit d'une construction faible qui sera avantageusement transformée en un type d'associations, comme illustré par la figure 18.35. Les attributs de référence se trou-

veront souvent dans des schémas résultant de l'analyse de formulaires ou de la rétro-ingénierie de bases de données existantes.

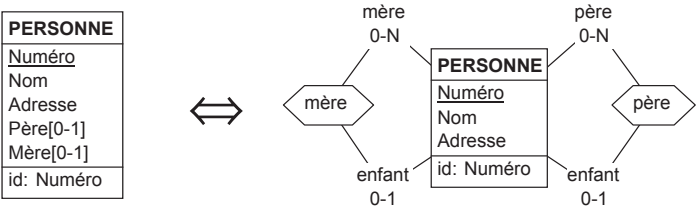


Figure 18.35 - Conversion d'attributs de référence en types d'associations

d) Attributs sériels et concaténés

Dans la méthode simplifiée d'analyse conceptuelle, nous avons abordé la représentation de propriétés multivaluées (section 13.5.6). Nous y avons en particulier signalé deux techniques à éviter (figure 13.15, reprise ci-dessous), cependant très populaires : les attributs sériels (Prénom1, ..., Prénom4) et les attributs concaténés (Prénoms). Si ces constructions apparaissent dans un schéma conceptuel, il est recommandé de les remplacer par leur *intention*, c'est-à-dire, un attribut multivalué.

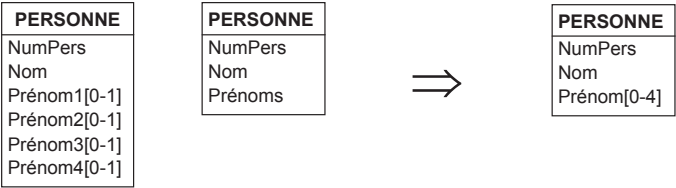


Figure 18.36 - Explicitation d'un attribut multivalué caché sous forme d'une liste d'attributs sériels ou concaténés.

e) Contrainte de coexistence (coex)

Une contrainte de coexistence (coex:) est souvent la trace d'un concept qui devrait être représenté de manière explicite. Dans la figure 18.37 la contrainte du schéma de gauche porte sur un rôle et un attribut. En extrayant ces composants et en les isolant dans un type d'entités spécifique AFFECTATION (schéma central), la contrainte est traduite en une structure explicite plus lisible et disparaît du schéma. Il est aussi possible de considérer les composants coexistants comme les caractéristiques d'un sous-type du type d'entités d'origine (EMPLOYE AFFECTE dans le schéma de droite). Nous pourrions également considérer AFFECTATION comme un *type d'entités association*, auquel cas on le transformera dans le type d'associations affecté (schéma inférieur). Ces quatre schémas sont sémantiquement équivalents, mais le choix de la forme la plus appropriée est de la responsabilité de l'analyste.

f) Contrainte d'implication (if)

Les contraintes d'implication (if:) se traitent de manière similaire.

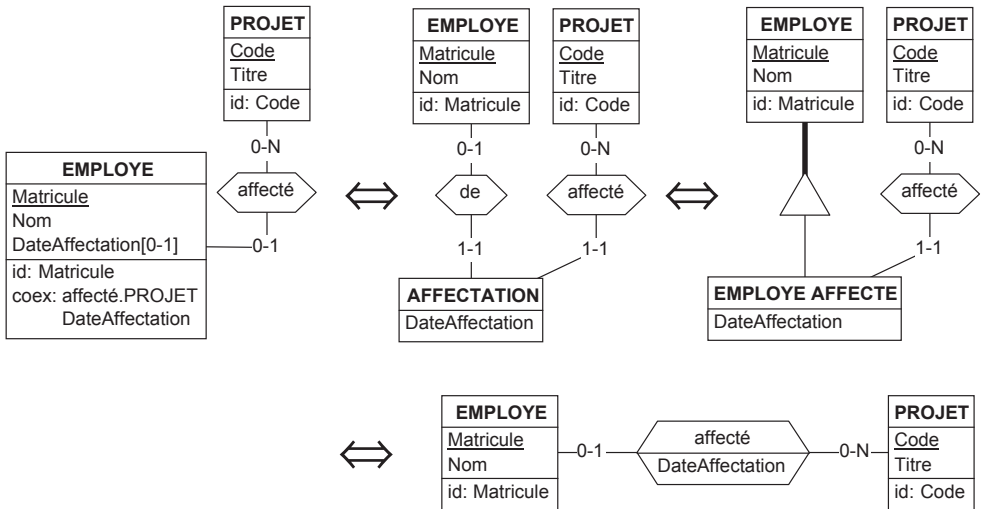


Figure 18.37 - Trois structures équivalentes à une contrainte de coexistence

La figure 18.38 (schéma de gauche) indique que si une valeur de PréfixeTéléph est présente, alors NuméroLocal possède aussi une valeur. Le regroupement de ces attributs en un unique attribut facultatif Téléphone permet d'éliminer cette contrainte complexe.

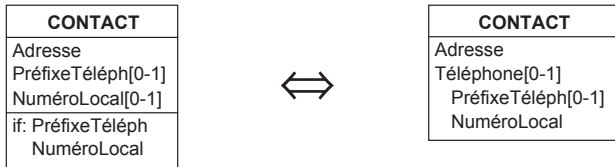


Figure 18.38 - Transformation d'une contrainte d'implication en structure

Le schéma de gauche de la figure 18.39 exprime la règle selon laquelle *un sportif ne peut s'inscrire à une école de spécialité que s'il a obtenu une habilitation*.

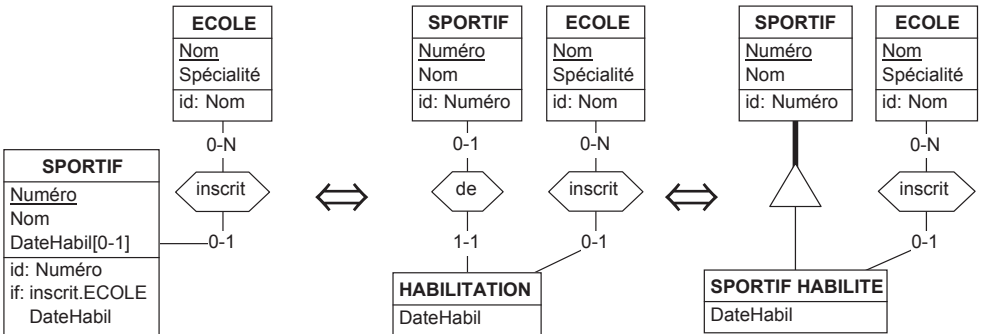


Figure 18.39 - Deux expressions structurelles équivalentes d'une contrainte d'implication

La traduction structurelle de cette contrainte s'obtient par l'extraction des composants de la contrainte sous la forme du type d'entités HABILITATION (schéma central).

Ici, le premier terme de la contrainte (DateHabil) est obligatoire tandis que le second (inscrit.ECOLE) est facultatif. Si la chose est utile, on peut également exprimer ces composants sous forme d'un sous-type (schéma de droite). La contrainte s'exprime alors de manière très intuitive : *seuls les sportifs habilités peuvent s'inscrire dans une école*.

g) Identifiant semi-obligatoire

Le rôle d'un identifiant est avant tout de repérer univoquement chaque entité dans sa population. Il peut également exprimer une contrainte d'intégrité (section 12.6.4). Dans ce dernier cas, il est possible qu'il possède une composition complexe, faite de composants obligatoires et facultatifs. L'identifiant secondaire de la figure 18.40 en est un exemple. La sémantique de ce schéma précise que (1) seuls les étudiants de terminale réalisent un mémoire et (2) les mémoires d'une année académique possèdent des titres distincts. Sa structure est valide mais complexe. Il est conseillé de rendre explicite le concept d'*étudiant de terminale*, de manière à normaliser l'identifiant (schéma de droite).

h) Relations is-a implicites

Nous étudierons au chapitre 19 des techniques de transformation d'une relation *is-a* en structures plus simples. Ces dernières se retrouveront parfois, *par inadvertance*, dans un schéma conceptuel, l'analyste n'ayant pas identifié, au travers de ces constructions primitives, les relations *is-a* sous-jacentes, lesquelles restent *implicites*.

La figure 18.41 représente une de ces structures. L'analyste y a représenté un type d'entités maître (SALARIE) auquel il a associé des types d'entités dépendants qui apportent chacun un complément d'information. Leurs noms suggèrent que ces compléments sont fonction de la nature de l'entité maître : *employé* ou *ouvrier*.

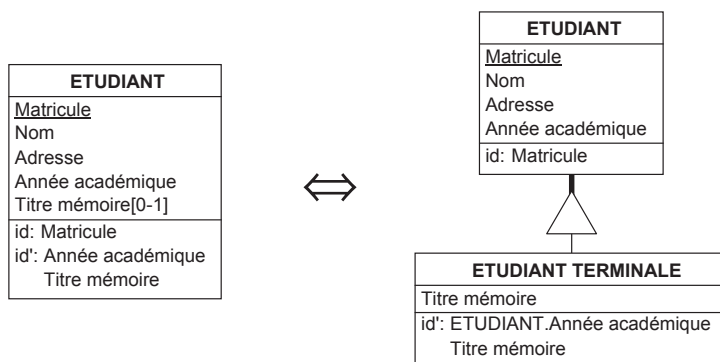


Figure 18.40 - Un identifiant semi-obligatoire et sa normalisation

Une telle structure pourrait résulter de l'analyse d'un formulaire à trois volets à remplir par un salarié, lequel aurait à remplir le volet principal (SALARIE) et un des

volets complémentaires (EMPLOYE ou OUVRIER) selon son statut.

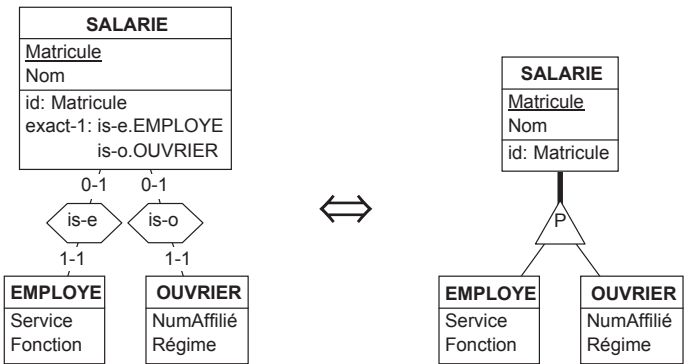


Figure 18.41 - Hiérarchie *is-a* cachée sous forme de types d'associations *un-à-un*

La figure 18.42 montre trois autres structures qui pourraient également suggérer des relations *is-a* implicites. Le schéma de gauche est assez proche de celui qui vient d'être discuté. Les types d'associations *un-à-un* sont remplacés par des attributs de référence (clés étrangères implicites non explicitées) sous forme des attributs identifiants *Matricule*. Le schéma central suggère une représentation des données d'un formulaire qui serait restée à l'état brut (voir section 18.4.3). Le type d'entités correspond au surtype tandis que les deux attributs complexes représentent les propriétés spécifiques aux deux sous-types implicites. Le schéma de droite représente les deux sous-types. On observe qu'ils possèdent des attributs en commun, lesquels pourraient être extraits pour former le surtype¹³.

Chacun des schémas de la figure 18.42 peut, après validation par l'analyste, être transformé dans la hiérarchie *is-a* de la figure 18.41.

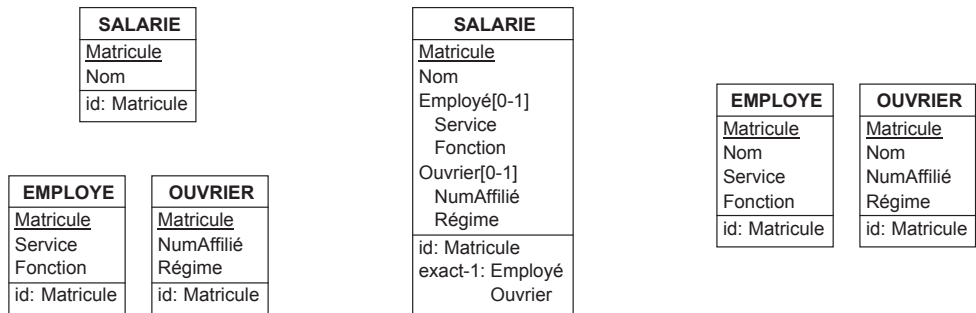


Figure 18.42 - Trois autres variantes de hiérarchie *is-a* cachée

13. L'extraction des composants communs n'est pas toujours aussi simple que ce que cet exemple semble suggérer. En toute généralité, la transformation relève d'une technique dénommée *Formal Concept Analysis*, elle-même basée sur la théorie des treillis de Galois. Nous en reparlerons à l'annexe A23.

18.7.3 Élimination des redondances structurelles

Une redondance structurelle existe lorsque la population d'une construction, dite redondante, est calculable à partir de celles d'autres constructions. En toute rigueur, une redondance structurelle est aisée à repérer lorsqu'une contrainte de redondance lui est associée. Celle-ci s'exprime sous la forme $C = \text{expression}$, où C est le nom de la construction redondante et expression est une expression dont l'évaluation renvoie une population qui, à tout instant doit être celle de C . La difficulté réside dans l'absence fréquente d'une telle contrainte, laquelle doit être recherchée et formulée avec précision sur la base d'indices.

Une construction redondante doit être supprimée, à moins qu'il n'existe des raisons impérieuses de la conserver. On exprimera alors la contrainte de redondance associée. Indépendamment de l'expertise de l'analyste, les schémas obtenus par l'analyse de documents structurés, par rétro-ingénierie et par intégration de schémas indépendants sont susceptibles de contenir de telles constructions.

a) Type d'associations et attributs de référence

Le schéma contient un type d'associations, généralement fonctionnel, mais aussi des attributs permettant de désigner les entités jouant le rôle opposé (figure 18.43). On trouvera souvent cette anomalie dans des schémas construits par des analystes débutants, mais aussi dans des schémas extraits de formulaires et de bases de données anciennes (TOTAL/IMAGE par exemple). Il est à noter que plusieurs formalismes hybrides de représentation de schémas relationnels indiquent à la fois les clés étrangères et des arcs symbolisant les associations qu'elles représentent : *MS Access*, *UML Data Profile* et *Oracle Designer* en sont trois exemples.

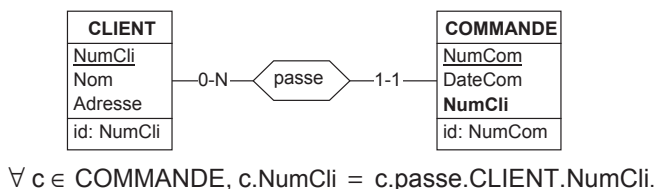


Figure 18.43 - L'attribut NumCli de COMMANDE est redondant avec le type d'associations passe

b) Attributs copiés et attributs calculables

Un attribut est repris d'un objet associé ou a des valeurs qui sont calculables à partir d'instances d'objets voisins. Le schéma de la figure 16.37 contient deux attributs de cette nature.

c) Types d'associations composés et projetés

Ce type de redondance existe dans un cycle de types d'associations (voir section 16.12.3). La figure 18.44 en montre un premier exemple élémentaire.

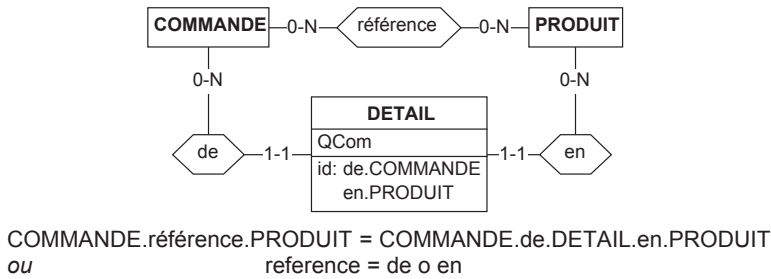


Figure 18.44 - *référence* est un type d'associations transitif à supprimer

Le type d'associations *référence* associe à chaque commande les produits que référencent ses détails. Il correspond à la composition des types d'associations *de* et *en*, et peut donc être supprimé.

Le schéma de la figure 18.45 contient un type d'association (*affectée*) qui est équivalent à une projection du type d'associations assignée. Il peut être supprimé. Attention cependant à ne pas confondre une contrainte de redondance avec une simple contrainte d'intégrité. Le même schéma contient une deuxième contrainte relative aux types d'associations assignée et offre. Ici, malgré la symétrie apparente avec le cas précédent, il ne s'agit pas d'une contrainte de redondance qui permettrait de supprimer une construction superflue, mais bien d'une contrainte d'intégrité qu'il est important de conserver.

Les deux exemples de redondance que nous venons d'analyser peuvent s'exprimer sous la forme d'expressions basées sur les opérateurs algébriques de *composition* (*o*) et de *projection* (*⌈*). Toute expression algébrique peut être utilisée pour définir des contraintes de redondance de complexité quelconque.

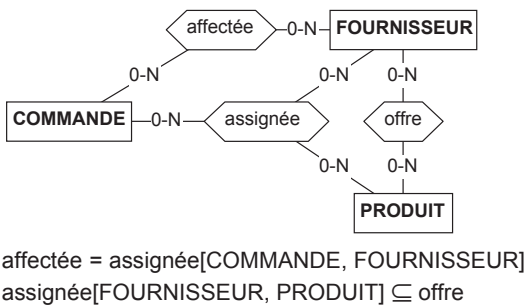


Figure 18.45 - Contrainte de redondance et contrainte d'intégrité

d) Types d'entités association et types d'associations

Un *type d'entités association* peut subsister dans un schéma alors que les associations qu'il représente ont été exprimées par un type d'associations (figure 18.46). On supprimera ce type d'entités.

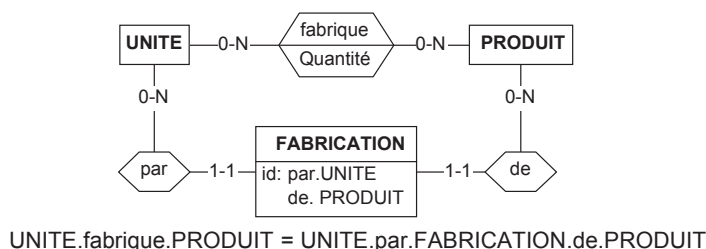


Figure 18.46 - Type d'entités association redondant

La différence entre ce cas et celui de la figure 18.44, auquel il ressemble fortement, réside dans l'identification de la construction la plus complète sur le plan de l'information : on conserve celle-ci et on supprime la plus faible. Si chacune apporte des informations spécifiques (des attributs par exemple), on procédera à une intégration des deux constructions (cas d'*auto-intégration*, section 18.9.7).

e) Constructions en double

Il n'est pas rare qu'un grand schéma comporte des constructions distinctes mais modélisant les mêmes concepts du domaine d'application, cette similitude ayant échappé à l'analyste¹⁴. Il s'agit d'identifier puis de réduire ces redondances par les techniques d'intégration de schéma, ici, par *auto-intégration*.

18.7.4 Élimination des redondances internes

Le phénomène de redondance interne a déjà été largement exploré aux sections 2.11 et 13.9.2 et surtout au chapitre 3. Il trouve son origine dans la présence de dépendances fonctionnelles dont le déterminant n'est pas un identifiant. On résout le problème par une décomposition de la table ou de la relation. Un schéma Entité-association peut également être le siège de constructions problématiques analogues à ces redondances internes.

a) Normalisation d'un type d'entités

Le schéma de gauche de la figure 18.47 comporte un type d'entités dont les attributs sont le siège de dépendances fonctionnelles anormales.

L'élimination de cette redondance est assez simple. On procède comme suit¹⁵ pour un ensemble de DF de même déterminant :

1. on définit un attribut composé D (le nom importe peu) qui regroupe tous les attributs *litigieux*, soit ceux qui forment le déterminant commun et ceux de tous les déterminés (figure 18.47, schéma central) ; les DF sont bien sûr toujours présentes au sein des composants de D ;

14. Cette anomalie est aussi observée dans les programmes, où on parle de *clones* ou de *quasi-duplication* (*near duplication*).

15. Évidemment, avec un peu d'habitude, on passe directement au schéma final sans expliciter le groupement des attributs litigieux.

2. on transforme l'attribut D en un type d'entités, selon la technique décrite à la section 13.5.5, figure 13.8 ;
3. on obtient le schéma de droite de la figure 18.47 après avoir nommé le nouveau type d'entités **DEPARTEMENT**¹⁶ et après avoir désagrégé l'attribut D, désormais inutile ; l'identifiant est le déterminant des DF, par application de la procédure de la section 3.6.2.

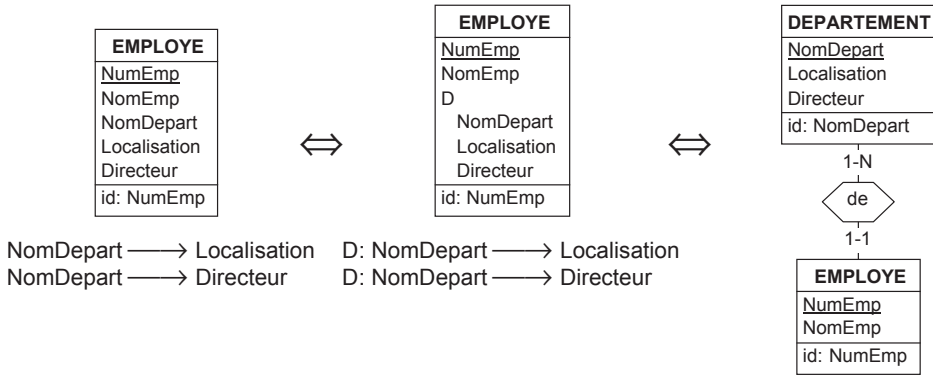


Figure 18.47 - Un type d'entités présentant des redondances internes et ses transformations de normalisation

Si on s'en tient à la stricte équivalence entre les schémas de gauche et de droite, la cardinalité du rôle de **DEPARTEMENT** est **[1-N]**. En effet, d'une part, plusieurs employés peuvent appartenir au même département (**N**) et d'autre part, il est impossible de représenter un département en dehors d'une entité **EMPLOYE** (**1**) dans le schéma de gauche. Cependant, puisque nous sommes dans une phase d'*amélioration* du schéma, nous pourrions en profiter pour assouplir cette dernière contrainte, qui apparaît inutilement restrictive, en assignant à ce rôle la cardinalité **[0-N]**.

b) Normalisation d'un type d'associations

Le premier cas est celui de l'exemple 1 de la section 18.2.3, sous-section c). On y a montré que la décomposition du type d'associations travailler résulte strictement de la procédure de décomposition de son expression relationnelle. Nous reprenons cet exemple à la figure 18.48.

La contrainte de coexistence, qui résulte de la stricte équivalence (symbole $\Leftrightarrow$) entre les deux schémas, nous rappelle la transformation de la figure 18.37 qui semble applicable ici. En mettant en évidence le concept d'*ingénieur actif*, on obtient le schéma de gauche de la figure 18.49. La contrainte de cardinalité **[1-N]** ainsi introduite est correcte mais peut être perçue comme indésirable. Assouplissons-la en la remplaçant par **[0-N]** (schéma central). Enfin, si après analyse plus fine du domaine, nous arrivons à la conclusion que **tous** les ingénieurs sont associés à un service, alors on adoptera le schéma simplifié de droite. Seul le schéma de gauche

16. Devinette : qu'est-ce qui est identifié par un nom de département et caractérisé par une localisation et un directeur ? Réponse : un **département** !

est strictement équivalent au schéma source.

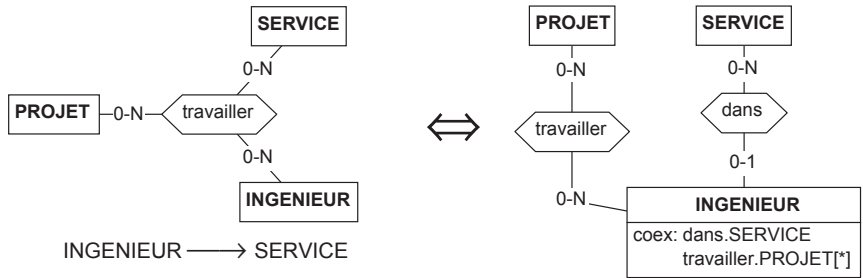


Figure 18.48 - Normalisation d'un type d'associations ternaire (1)

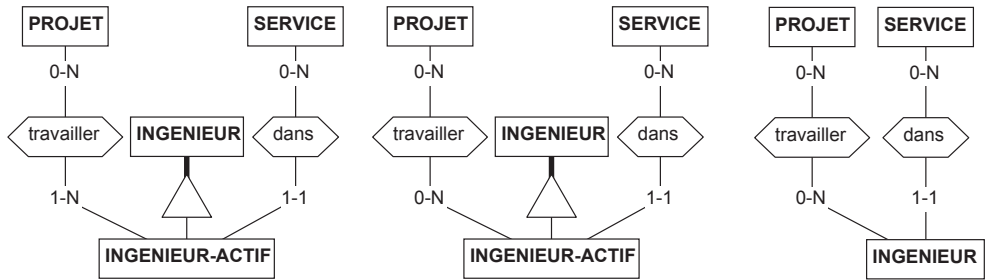


Figure 18.49 - Assouplissement de la contrainte de coexistence - 3 variantes

Considérons à présent l'exemple plus complexe de la figure 18.50. Ce schéma ne nous est pas inconnu puisque nous l'avons déjà rencontré à la section 16.11.4, mais surtout lors de l'étude de la normalisation d'une relation (section 3.8.5).

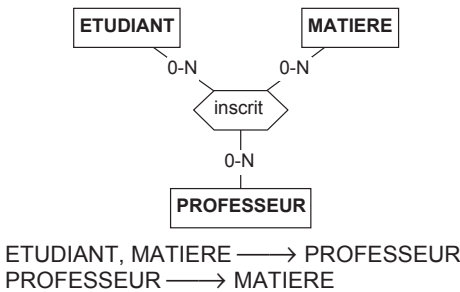


Figure 18.50 - Type d'associations ternaire à normaliser

Nous avons observé que ce schéma était en 3^e forme normale (3FN) mais pas en forme normale de Boyce-Codd (FNBC). Il n'est donc pas suffisamment normalisé. Nous avons vu qu'il admettait trois structures équivalentes dont aucune, malheureusement, n'était exempte de défauts. La première structure conserve le schéma (et les redondances internes qu'il implique) ; on y a simplement traduit la première DF sous la forme d'un identifiant. La deuxième structure est en FNBC. Les redondances

internes ont disparu mais au prix d'une contrainte complexe. La troisième structure obéit à la forme normale à clé élémentaire (FNCE) : ce schéma est plus complexe, implique de fortes redondances, mais admet une implémentation relationnelle astucieuse assez pratique (section 3.8.5). Ces structures ont fait l'objet de l'exercice 16.8. On se reportera donc à la solution proposée à la fin du chapitre 16.¹⁷

18.7.5 Autres critères de qualité

On évoquera sans les détailler d'autres critères de qualité dont l'importance dépendra de la complexité du schéma et de ses contextes d'élaboration et d'utilisation. Ces critères additionnels sont étudiés en détail à l'annexe A18.

- **absence de sur-spécifications** : le schéma représente les types de faits raisonnablement utiles, sans préjuger de la manière dont ils seront implémentés.
- **absence de constructions anormales** : on évitera notamment les constructions *anormales* (cas limites d'utilisation d'une construction, style inhabituel) ou *irrégulières* (changement injustifié de style, style incohérent).
- **régularité d'expression ou prévisibilité** : le lecteur ne doit pas être *surpris* par l'usage d'une construction pour exprimer un certain type de faits ; on évitera notamment les changements injustifiés de style et les styles incohérents.
- **qualité graphique** : la manière dont les constructions d'un schéma sont disposées dans l'espace graphique peut améliorer considérablement sa lisibilité ; il n'est pas recommandé par exemple de recouvrir le type d'entité DETAIL par le type d'entités COMMANDE (de sorte qu'il soit invisible) ou au contraire de le positionner 10 mètres à droite.
- **respect des standards** : le schéma doit respecter certaines règles imposées par l'organisme pour lequel (ou dans lequel) il est développé, quand bien même le concepteur les estimerait inutiles, voire absurdes.

18.8 ÉVALUATION D'UN SCHÉMA

Il s'agit ici de vérifier qu'un schéma correct (section 18.6) et normalisé (section 18.7) est conforme aux exigences des utilisateurs. Le schéma doit contenir les objets et les propriétés traduisant correctement les besoins en information des utilisateurs. Pas plus, pas moins.

Il est évident que seul l'utilisateur peut certifier que le schéma qui lui est soumis est satisfaisant. Il n'est pas possible en effet, comme nous l'avons fait pour la correction et la normalisation du schéma, de définir des règles structurelles garantissant que le schéma correspond à ce que veut l'utilisateur. Il est cependant tout aussi évident qu'un utilisateur normalement constitué se sentira démuné face à la respon-

17. La contrainte de coexistence de la forme FNBC pourrait être traitée selon le procédé de la figure 18.49, en mettant en évidence le concept de *professeur actif*.

sabilité de certifier un schéma. Tant la taille du schéma (plusieurs centaines de types d'entités) que le formalisme utilisé pour l'exprimer, fût-il graphique, le rendent souvent peu lisible pour un non-spécialiste de la modélisation¹⁸. Il faut donc recourir à des techniques indirectes permettant à l'utilisateur d'exprimer son opinion sur le schéma qui lui est proposé. On exploite le plus souvent deux aptitudes qu'il est normal d'exiger d'un utilisateur : la compréhension de propositions simples exprimées en langage naturel et la conformité du comportement d'un logiciel aux attentes de l'utilisateur. Ce qui conduit à deux techniques très efficaces d'évaluation d'un schéma : la *paraphrase* et le *prototypage*. Nous évoquerons également deux techniques complémentaires : l'*inférence structurelle* et les *heuristiques d'erreurs*.

18.8.1 Paraphrase du schéma

La paraphrase, ou réinterprétation, d'un schéma conceptuel consiste à reformuler sous forme de propositions en langage naturel chaque concept présent dans le schéma. Ces propositions sont alors soumises à l'approbation de l'utilisateur.

La paraphrase apparaît comme le processus inverse de l'analyse d'énoncés textuels décrite à la section 18.2. À titre d'exemple, le schéma de la figure 16.42 pourrait produire un texte dont on reprend quelques extraits à la figure 18.51.

- Le domaine comprend des *documents*, des *rapports*, des *ouvrages*, des *exemplaires*, des *auteurs*, des *emprunteurs* et des *projets*.
- Tout *document* est identifié par son *ID-Doc*. Il est caractérisé par un *Titre*, une *date de publication* et possède de 0 à 10 *mots-clés*.
- Un *document* peut être un *rapport* ou un *ouvrage*. Un *rapport* ne peut pas être un *ouvrage*.
- Un *rapport* est identifié par son *Code-Rapport*. Il est caractérisé par un *Projet*.
- Un *document* peut être écrit par un nombre quelconque d'*auteurs*. Tout *auteur* écrit au moins un *document*.
- Un *document* peut être *réservé* par un nombre quelconque d'*emprunteurs*. Chacun de ces faits est caractérisé par une *date de réservation*.

Figure 18.51 - Exemple de paraphrase

La production d'un tel texte est très systématique et peut être aisément automatisée. Ainsi, les phrases reprises ci-dessus ont été générées par l'atelier DB-MAIN. Cependant, la seule utilisation des noms apparaissant dans le schéma ne suffit pas à produire un texte correct. Lors de l'utilisation de l'atelier DB-MAIN, il sera nécessaire de préciser pour certains objets des paramètres tels que les suivants :

- le *nom naturel* de l'objet : *MotClé* s'écrit *mot clé* ;
- le *genre* de l'objet : *DateRéservation* est féminin ;
- le *pluriel* de l'objet : le pluriel de *MotClé* est *mots clés*.

18. Une analyse critique de la lisibilité d'un schéma Entité-associations pour des non-spécialistes a été publiée dans [Moody, 1996].

18.8.2 Prototypage de la base de données

Le prototypage consiste à produire rapidement, de manière automatique, une base de données conforme au schéma conceptuel. On confie alors à l'utilisateur le soin d'expérimenter ce prototype en y introduisant un jeu représentatif de données et d'exploiter ces données à l'aide de requêtes typiques. L'utilisateur est chargé de signaler les anomalies de comportement du prototype.

Exemples

- Lors de l'utilisation du prototype produit à partir du schéma de la figure 16.34, l'utilisateur s'étonne qu'il ne lui est pas permis d'inscrire une deuxième fois un étudiant à une matière déterminée. S'il n'admet pas cette restriction, l'analyste devra modifier l'identifiant du type d'associations inscrit.
- À partir du prototype dérivé du schéma de la figure 18.45, l'utilisateur se voit refuser l'assignation d'une commande à un fournisseur. Ce refus est la conséquence de la contrainte d'inclusion qui impose que la commande soit assignée à un fournisseur qui offre le produit commandé.

L'annexe A18 propose une illustration de la technique de prototypage appliquée à des bases de données de petite taille (quelque dizaines de tables). Le prototype permet de créer, modifier et explorer les données de manière navigationnelle. Ce prototype est écrit en SQLfast et est produit automatiquement par un générateur, lui-même écrit en SQLfast, exploitant les tables du catalogue de la base de données en cours d'évaluation.

18.8.3 Inférence structurelle

Alors que la paraphrase exprime d'une manière compréhensible les structures de base du schéma, l'inférence structurelle décrit les propriétés dérivées de celles-ci. Elle rend conscient l'utilisateur des conséquences des besoins qu'il a exprimés et qui ont été traduits dans ces structures de base. Nous donnerons quelques exemples basés sur des schémas rencontrés dans les pages précédentes.

- Schéma 16.8 : il est possible qu'un *immeuble* soit *vendu* à son *vendeur*.
- Schéma 16.16 : il est possible qu'un *employé* soit *affilié* à une *caisse d'assurance*; un *ouvrier* est identifié par son *matricule*.
- Schéma 16.36 : il est possible qu'une *personne* soit son propre *responsable*.
- Schéma de l'exercice 16.7 : pour identifier un *détail de facture*, il faut préciser deux valeurs de *NumCli* ; une *facture* peut référencer plusieurs fois le même *produit*, ce qui n'est pas le cas des *commandes*.
- Schéma de l'exercice 16.4 : un même *numéro de commande* peut apparaître chez deux *clients*.
- Les trois attributs facultatifs A1, A2, A3 de E sont soumis aux contraintes excl: A1, A2 et excl: A2, A3 : les attributs A1 et A3 ne sont soumis à aucune contrainte d'existence ; il est possible que ces trois attributs soient simultanément sans valeur.

18.8.4 Les heuristiques d'erreurs

Cette technique est celle que privilégie le professeur d'informatique chargé de corriger des schémas conceptuels élaborés par ses étudiants. Il existe deux techniques de correction. La première postule *a priori* que le schéma est complètement erroné (note initiale de 0) et que sa valeur est mesurée par le nombre de constructions correctes qu'on y détecte (c'est-à-dire qui sont équivalentes à celles que l'enseignant estime correctes). La seconde, plus rapide, part de l'hypothèse que le schéma est correct (note initiale maximale) et que sa valeur s'obtient par soustraction de celle des constructions erronées. L'enseignant sait que les erreurs se produisent dans certaines constructions critiques du schéma. Il est peu probable qu'un type d'associations binaire 1:N et sans attribut soit erroné. C'est évidemment parmi les types d'associations n-aires, dotés d'attributs et soumis à des contraintes complexes que les erreurs éventuelles sont à rechercher. L'enseignant ignorera donc les parties du schéma ne comportant que des constructions simples pour se concentrer sur les **constructions à risque**. C'est cette seconde technique qui est suggérée ici. On en tire une procédure d'analyse du schéma en vue d'identifier les erreurs potentielles. Elle se décrit comme suit :

1. On identifie dans le schéma les *constructions à risque*. On recherchera en particulier les types d'associations complexes, les circuits de types d'associations, les relations *is-a*, les types d'entités sans attribut ou sans identifiant, les identifiants complexes, les attributs complexes (composés, multivalués, dotés d'identifiants), les identifiants.
2. Pour chaque construction à risque, on vérifie, en retournant aux sources, sa validité. Cette validité est éventuellement confirmée par l'utilisateur.
3. S'il apparaît que la construction est erronée, elle est corrigée.

Commentaires

- Les constructions à risque peuvent être définies formellement, de sorte que leur identification peut être automatisée dans un AGL.
- On limite la vérification du schéma à une partie réduite de celui-ci.
- Une erreur dans une construction non à risque ne sera pas détectée, mais on sait qu'elle sera peu probable.
- C'est notamment sur la base de cette approche que sont en général construits les systèmes de **métriques** utilisés pour évaluer la qualité d'un schéma. En évaluant la fréquence des constructions à risque, on obtient une mesure qui permet de situer un schéma dans une moyenne. Par exemple, une trop grande fréquence de hiérarchie *is-a* de grande profondeur (par exemple au moins cinq niveaux de sous-types) est jugé anormal et conduira à une vérification du schéma. De même, on peut tolérer des types d'entités sans identifiants, mais en nombre limité.

18.8.5 Remarque

Lorsque l'analyse du domaine couvre également ses aspects fonctionnels et dynamiques, à côté du schéma conceptuel seront également développées des spécifications

telles que les diagrammes d'état et les diagrammes d'activité selon la terminologie d'UML. Ceux-ci font référence à des données décrivant les (classes d')objets du domaine. Il est alors possible de procéder à un autre type de validation.

1. Tous les objets et types d'objets utilisés dans les diagrammes fonctionnels et dynamiques sont-ils décrits dans le schéma conceptuel ? Si tel n'est pas le cas, on s'assurera que les objets manquants ne sont pas des objets statiques naturels du domaine.
2. Tous les objets appartenant au schéma conceptuel sont-ils utilisés dans au moins un diagramme fonctionnel ou dynamique ? Si tel n'est pas le cas, on s'assurera qu'il n'y a pas une erreur soit dans le schéma conceptuel, soit dans un de ces diagrammes.

Ce type de validation peut être effectué de manière automatique.

18.9 INTÉGRATION DE SCHÉMAS

L'analyse du domaine basée sur une décomposition de celui-ci en sous-systèmes homogènes conduit à une collection de sous-schémas corrects, normalisés et validés. Le schéma conceptuel global s'obtient à partir de ceux-ci via le processus d'**intégration**. D'autre part, chaque sous-schéma conceptuel est généralement obtenu par l'intégration de fragments issus de l'analyse de diverses sources indépendantes, textuelles ou formulaires. Dans les organisations d'une certaine taille, il n'est pas rare que chaque sous-système possède sa propre culture et en particulier sa propre manière de percevoir et de nommer les choses. On comprend dès lors que l'intégration de sous-schémas conceptuels qui modélisent des parties identiques du domaine d'application puisse constituer une opération complexe, qui ne se limite pas à *additionner* ces schémas.

L'intégration de schémas, qui est un des problèmes les plus complexes, fait l'objet de nombreux travaux depuis plusieurs dizaines d'années. Il n'existe pas à l'heure actuelle de solution générale, raison pour laquelle nous proposerons ici une description du problème et des pistes de résolution sans prétendre à l'exhaustivité. Son principe est le suivant.

Étant donné un ensemble $\mathbf{SS} = \{\mathbf{S}_1, \mathbf{S}_2, \dots, \mathbf{S}_N\}$ de N schémas relatifs à N sous-systèmes, non nécessairement disjoints, d'un même domaine d'application, l'intégration de $\mathbf{SS}$ consiste à produire un schéma global unique $\mathbf{S}$ qui représente, *sans redondance, tous les concepts* des schémas de $\mathbf{SS}$ et eux seulement. Tout type de faits représenté dans un schéma $\mathbf{S}_i$ est aussi représenté dans $\mathbf{S}$. Tout type de faits représenté dans $\mathbf{S}$ l'est aussi dans au moins un des sous-schémas de $\mathbf{SS}$. Après intégration complète, chaque sous-schéma conceptuel $\mathbf{S}_i$ apparaît comme une vue du schéma global $\mathbf{S}$ ¹⁹.

19. En d'autres termes, les données correspondant à un sous-schéma sont calculables à partir des données correspondant au schéma global. Cette propriété apparaîtra de manière concrète dans les sections 19.9 et 21.10.

Nous allons nous heurter à plusieurs difficultés lorsque nous essayerons de détecter les types de faits qui sont représentés simultanément dans deux schémas. En effet, ces types de faits auront souvent des représentations différentes (conflits de nom, conflit syntaxique), mais en outre ils seront parfois perçus de manière différente (conflit sémantique). La procédure d'intégration devra identifier ces conflits et les résoudre.

18.9.1 Scénarios d'intégration

Sauf cas particuliers (schémas peu nombreux et quasi identiques), on suggère généralement de procéder à l'intégration de SS de manière itérative, en intégrant une paire de schémas à la fois. On réduit ainsi le processus général à une suite d'**intégrations binaires**. La figure 18.52 illustre le scénario que nous retiendrons.

Il consiste à ordonner les schémas par ordre de qualité décroissante, le premier, S_1 , étant le schéma le plus riche, le plus expressif, le plus complet et le plus fiable. On intègre ensuite S_1 avec S_2 , ce qui produit un premier schéma intégré S_{12} . Le schéma S_3 est alors intégré à S_{12} , ce qui donne S_{123} . En procédant de la sorte, dans cet ordre, jusqu'à épuisement de SS, on obtient le schéma global S .

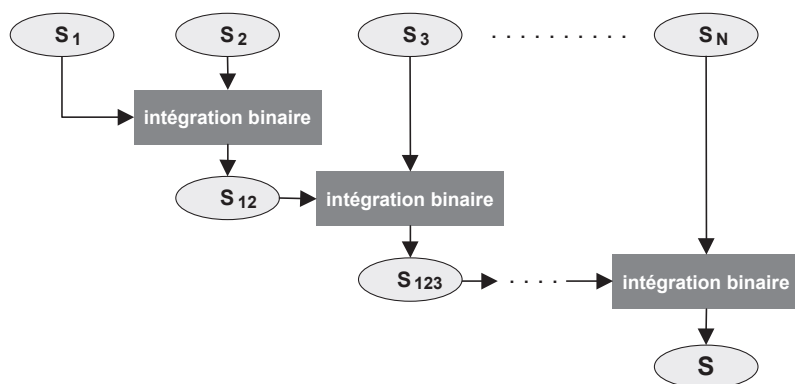


Figure 18.52 - Procédure pratique d'intégration de N schémas en N - 1 étapes

18.9.2 Principes de l'intégration binaire de schémas

Supposons que les schémas S_1 et S_2 à intégrer soient parfaitement *alignés*, de sorte que tout type de faits représentés dans les deux schémas est modélisé exactement de la même manière : même *nature* d'objet²⁰ et même *nom*. Dans ce cas, l'intégration se réduit à une simple fusion technique. Ce processus, illustré à la figure 18.53, est particulièrement simple.

20. Type d'entités, type d'associations, domaine de valeurs, attribut ou contrainte.

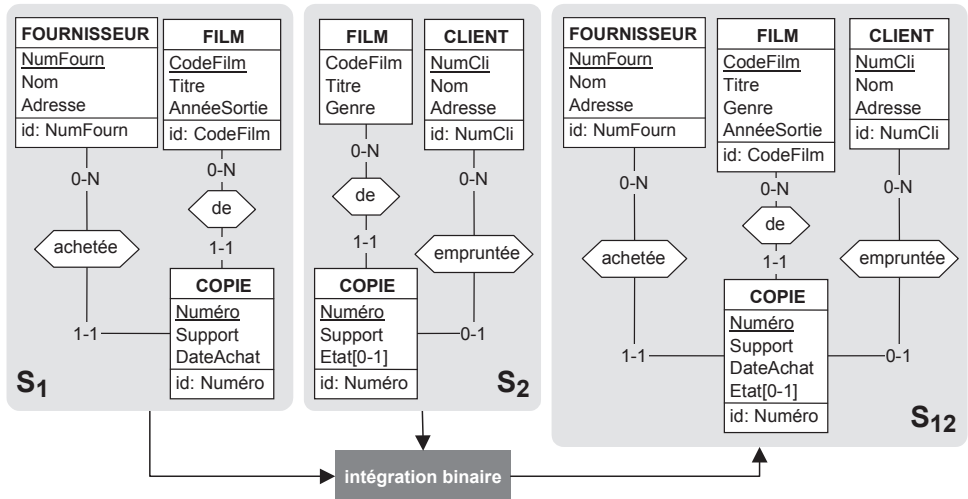


Figure 18.53 - Une intégration réduite à une simple fusion

Malheureusement, les choses ne sont pas toujours aussi simples. Par exemple, un type de faits sera représenté dans un schéma par un attribut et dans l'autre par un type d'entités, sous des noms différents, et soumis à des contraintes différentes, voire contradictoires. Le principe de la fusion que nous venons d'illustrer reste valable dans ce cas, pour autant que nous ayons auparavant *préparé* les deux schémas. La procédure générale d'intégration binaire comporte trois étapes :

1. recherche des correspondances ;
2. unification des schémas ;
3. fusion des schémas.

Nous allons examiner en détail et illustrer chacune de ces étapes, avant de développer une étude de cas complète.

18.9.3 Intégration basée sur les correspondances

On rappelle d'abord que toute instance d'un objet d'un schéma *représente* (ou *dénote*, ou *désigne*) un objet ou un fait du domaine d'application : une entité d'un type représente un objet ou fait réel, une valeur d'attribut représente l'état d'une propriété d'un objet réel et une association représente une relation entre des objets réels. Soit O_1 un objet du schéma S_1 ; on dénotera par D_1 l'ensemble des objets ou faits du domaine d'application que désignent les instances de O_1 (figure 18.54, où O_1 est un type d'entités).

Considérons à présent les schéma S_1 et S_2 à intégrer. Les objets O_1 et O_2 y représentent respectivement les ensembles D_1 et D_2 du domaine d'application. Nous soupçonnons qu'il existe un lien de **similitude** entre O_1 et O_2 , similitude dont il faudra tenir compte lorsque nous intégrerons S_1 et S_2 .

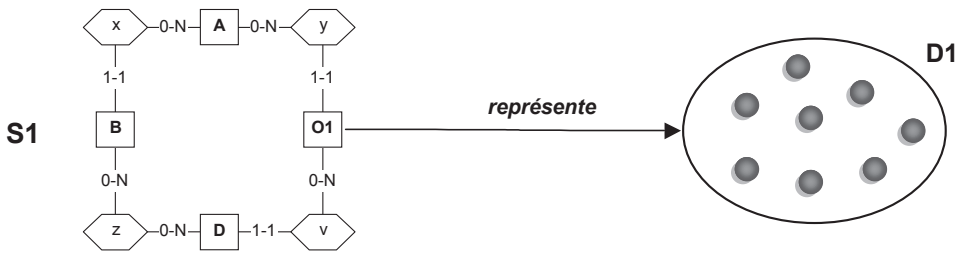


Figure 18.54 - Relation de représentation entre un objet O1 du schéma et une partie du domaine d'application

Ce lien, que nous appellerons **correspondance**, exprime la relation qui existe entre les ensembles D1 et D2 du domaine d'application (figure 18.55). Ces ensembles sont-ils toujours identiques ? L'un inclut-il toujours l'autre ? Sont-ils toujours disjoints ?

On retiendra sept relations entre D1 et D2, cinq positives, l'*égalité*, la *complémentarité*, la *compréhension*, l'*union* et la *dérivation*. On y ajoute la correspondance négative de *différence*. De ces relations, on dérive les correspondances de même nom.

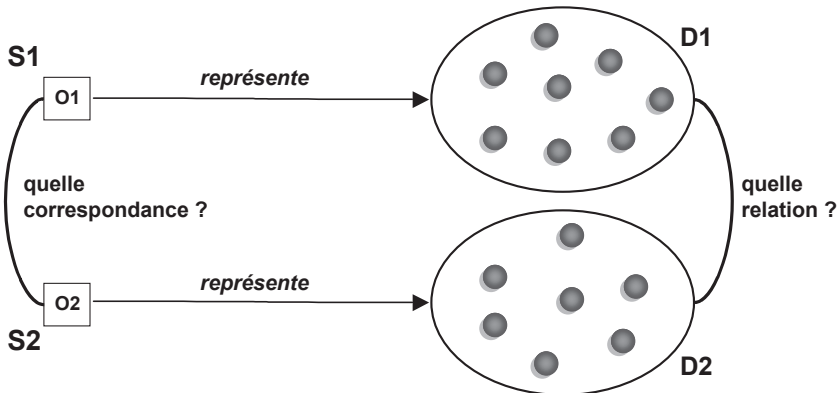


Figure 18.55 - La correspondance entre O1 et O2 représente la relation entre les ensembles D1 et D2

- **Correspondance d'égalité (=)**

Une correspondance d'égalité, notée =, existe entre O1 et O2 si D1 = D2. L'égalité indique que les objets O1 et O2 ont même sémantique et qu'il est inutile de conserver les deux. Si les objets sont de même nature, l'égalité correspond généralement à une synonymie.

Dans la figure 18.56, on compare les types d'entités CLIENT et PATIENT, appartenant respectivement aux schémas S1 et S2. Il apparaît que les ensembles d'entités représentés sont identiques (*client* est un synonyme de *patient*). On trace donc une correspondance d'égalité entre CLIENT et PATIENT. Descendant à présent vers leurs attributs, on établit que NumEnreg est un synonyme de Numéro et que

Adresse est un autre nom de Contact. Quand aux attributs Nom, ils représentent la même propriété. Portant le même nom, ils sont en correspondance d'égalité par défaut.

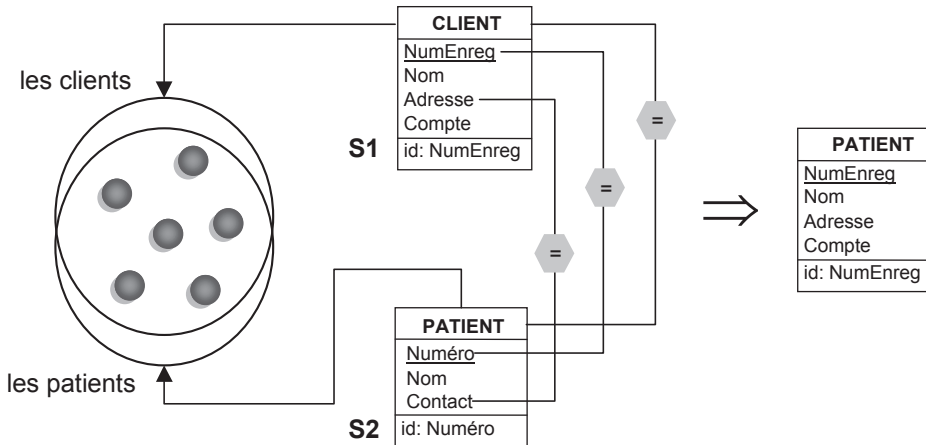


Figure 18.56 - CLIENT et PATIENT sont en correspondance d'égalité

En revanche, l'attribut Compte n'a pas d'équivalent dans PATIENT. Le type d'entités PATIENT à droite de la figure représente le résultat de l'intégration des deux types d'entités sources²¹.

- **Correspondance de complémentarité (cp)**

Une correspondance de complémentarité, notée *cp*, existe entre O_1 et O_2 s'il existe une bijection $b: D_1 \longleftrightarrow D_2$, indiquant que chaque objet de D_2 est un *complément* ou un *aspect* d'un objet de D_1 . Les objets O_1 et O_2 doivent être conservés, unis par un type d'associations 1:1 ou 1:N.

Dans la figure 18.57, on compare les types d'entités CLIENT et HISTORIQUE. Il apparaît que les historiques sont distincts des clients, mais qu'il en constituent des compléments. On trace donc une correspondance de complémentarité entre CLIENT et HISTORIQUE. L'intégration de ces types d'entités consiste à les conserver et à créer un type d'association entre les deux. L'attribut Numero de HISTORIQUE, qui permet de faire le lien avec CLIENT peut être supprimé, étant désormais redondant avec le nouveau type d'associations.

- **Correspondance de compréhension ($\supseteq$)**

Une correspondance de compréhension, notée $\supseteq$, existe entre O_1 et O_2 si $D_1 \supseteq D_2$. Une correspondance de compréhension entre types d'entités indique généralement que l'objet O_1 doit être considéré comme un surtype de O_2 . Les attributs communs ne sont repris que dans le surtype, pour autant qu'ils soient en correspondance d'égalité ou de compréhension. Il en est de même des rôles et des contraintes communs.

21. Selon une logique ultralibérale, on aurait évidemment choisi le nom CLIENT.

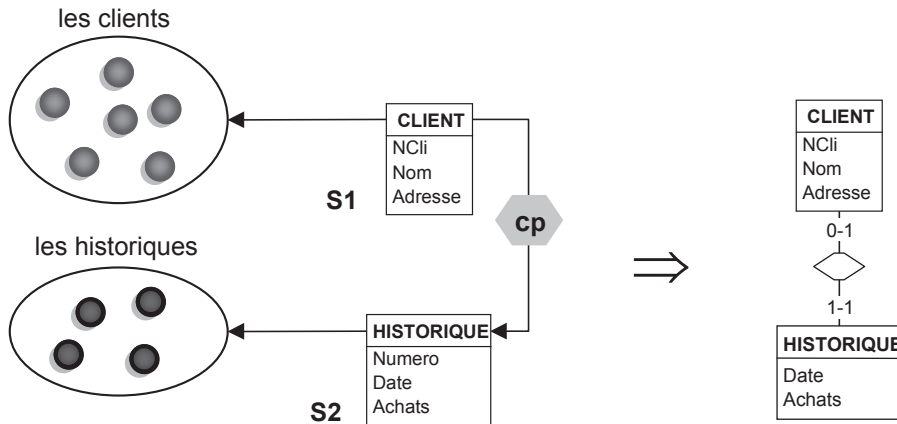


Figure 18.57 - HISTORIQUE est en correspondance de complémentarité avec CLIENT

Dans la figure 18.58, on observe que les ouvriers forment structurellement (c'est-à-dire à tout instant), un sous-ensemble des salariés.

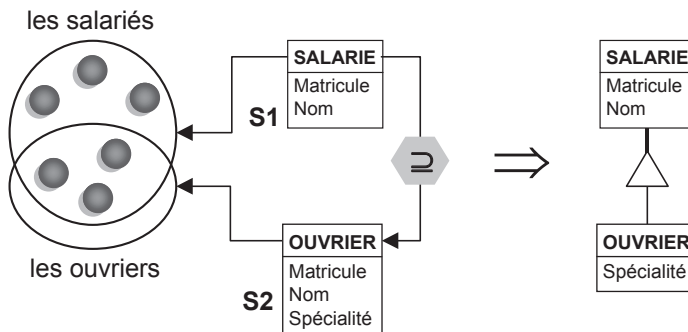


Figure 18.58 - SALARIE est en correspondance de compréhension avec OUVRIER

- **Correspondance d'union ($\cup$)**

Une correspondance d'union, notée $\cup$, exprime le fait que, pour les objets en correspondance (au moins deux), il existe un objet O_3 pertinent, désignant l'ensemble D_3 , et tel que $D_3 \supseteq D_1$, $D_3 \supseteq D_2$, etc. L'union indique généralement la nécessité de créer un nouvel objet O_3 et de le déclarer surtype de O_1 , O_2 , etc.

Dans la figure 18.59, on observe que les médecins et les infirmiers partagent des caractéristiques communes et qu'ils sont tous membres du personnel médical. On leur donne un surtype commun, **PERSONNEL**, qui reprend les attributs communs, pour autant que ceux-ci soient en correspondance d'égalité.

- **Correspondance de dérivation (C)**

Une correspondance de dérivation, notée C, existe entre O_1 et O_2 s'il existe une fonction de calcul c : $D_1 \rightarrow D_2$.

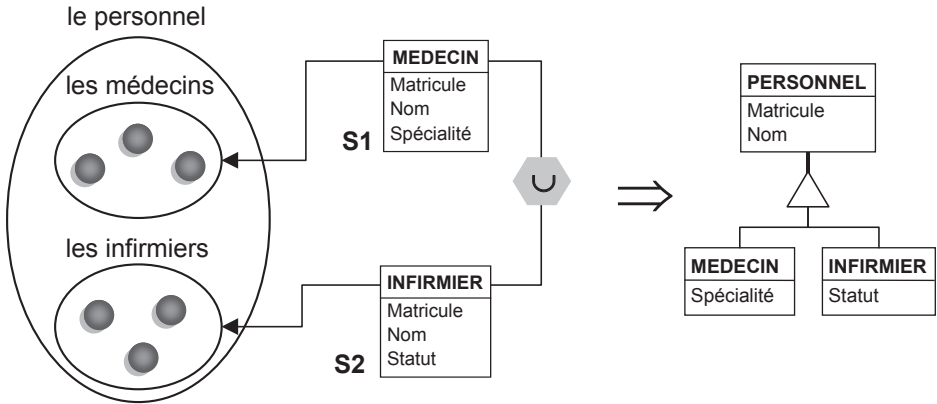


Figure 18.59 - MEDECIN et INFIRMIER sont en correspondance d'union

Cette fonction peut être exprimée comme une contrainte d'intégrité ou par une fonction procédurale de forme quelconque. La dérivation suggère de ne pas conserver l'objet O2 dans le schéma intégré.

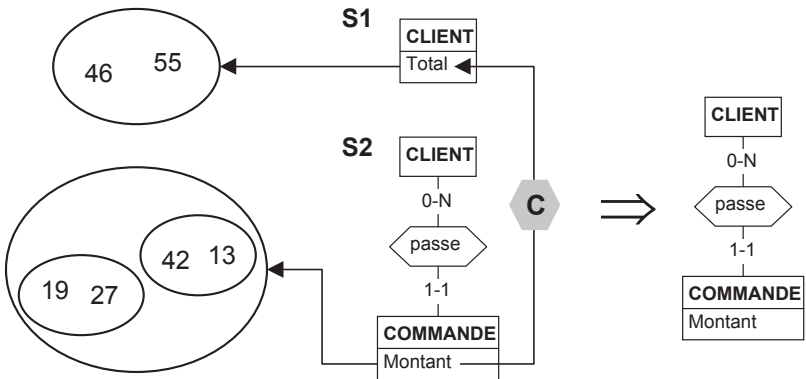


Figure 18.60 - Les valeurs de l'attribut Montant sont calculables à partir de celles de Total

- **Correspondance de différence ($\neq$).**

Une correspondance de différence, notée $\neq$, existe entre O1 et O2 si, *contrairement à ce que suggèrent les schémas*, il n'existe aucune relation sémantique entre ces objets. La différence sera notamment utilisée lorsque deux objets indépendants portent le même nom (homonymie). Cette correspondance, contrairement aux autres, est *négative* (elle affirme l'inexistence d'une propriété apparente). On se contentera bien sûr d'indiquer les différences qui peuvent prêter à confusion. Tel est le cas de l'exemple de la figure 18.61, dans lequel la notion d'*unité* est comprise différemment dans les deux schémas.

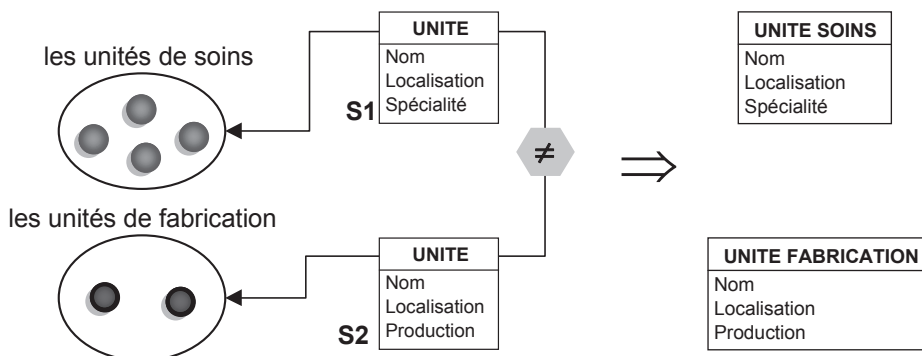


Figure 18.61 - Les deux types d'entités UNITE sont distincts

On ajoute trois règles générales à respecter lors de la définition de correspondances.

- *Règle propre aux attributs* : une correspondance peut être établie entre deux attributs s'il existe une correspondance positive entre leurs parents respectifs, éventuellement via leurs surtypes.
- *Règle propre aux types d'associations* : une correspondance peut être établie entre deux types d'associations s'il existe des correspondances positives entre leurs rôles ou les types d'entités qui jouent ces rôles, éventuellement via leurs surtypes.
- On convient qu'il existe une correspondance d'égalité implicite entre deux objets de même nature, de même nom et, pour des attributs, de parents en correspondance d'égalité. Les correspondances implicites ne sont pas notées (figure 18.53).

18.9.4 Repérage des objets en correspondance

L'identification des objets à mettre en correspondance dans deux schémas à intégrer demande une connaissance approfondie du domaine d'application. Cependant, certaines propriétés des schémas peuvent nous aider dans cette recherche. Citons trois techniques complémentaires.

- **L'analyse des noms** des objets peuvent souvent nous mettre sur la piste. Idéalement, ces noms sont identiques ou proches linguistiquement : CLIENT et CLIENTS, ORDRE et COMMANDE, ARTICLE et REPORTAGE, NCom et NumCommande.
- **L'analyse structurelle** des schémas peut mettre en évidence des sous-structures analogues : deux types d'entités possèdent certains attributs communs, sont soumis à des contraintes similaires et jouent certains rôles identiques. Ou encore, deux types d'associations ont les mêmes membres et les mêmes contraintes de cardinalité.
- Des techniques plus avancées peuvent être développées à partir d'**ontologies** et de leurs techniques d'alignement. Une ontologie est une représentation formalisée de l'ensemble des concepts d'un domaine d'application et de leurs associations. Bien que cette définition soit analogue à celle du schéma conceptuel,

L'ontologie relève d'une discipline distincte, à la base du *web sémantique*. Une ontologie décrivant les concepts d'une organisation pourra par exemple nous apprendre qu'un EMPLOYE est une PERSONNE et que AUTEUR est un rôle qu'une PERSONNE peut jouer. Elle nous indiquera aussi qu'une PERSONNE a généralement un nom, un prénom et une date de naissance.

Ces techniques sont décrites plus en détail à l'annexe A18.

18.9.5 Unification : le cas des correspondances hétérogènes

Nous avons jusqu'ici considéré que les correspondances étaient établies entre objets de même nature : entre deux types d'entités, deux attributs ou deux types d'associations. De telles correspondances sont qualifiées d'**homogènes**. Nous avons vu que leur exploitation pour intégrer deux schémas est simple et intuitive.

Il est fréquent cependant qu'on mette en correspondance deux objets de nature différente, telles qu'un type d'entités et un attribut. Il s'agit d'une correspondance **hétérogène**. Plusieurs auteurs ont publié des règles d'intégration basées sur des correspondances hétérogènes. Cependant, l'approche la plus simple consiste à **unifier** les deux schémas par la **transformation** des objets en correspondance de manière telle qu'il n'existe plus que des correspondances homogènes.

Nous examinerons, pour trois couples hétérogènes, attribut/type d'entités, attribut/rôle et type d'associations/type d'entités, les opérations recommandées pour unifier deux schémas en correspondance hétérogènes.

a) Correspondance entre un attribut et un type d'entités

Dans la figure 18.62, l'attribut COMMANDE.Objet est mis en correspondance de compréhension avec le type d'entités ARTICLE. Transformons cet attribut en un type d'entités OBJET par représentation des valeurs. La correspondance $\supseteq$ est à présent définie entre deux types d'entités et devient homogène. Leurs identifiants sont mis en correspondance d'égalité. Après intégration, on observera que OBJET est un sous-type *faiblement spécifique* de ARTICLE. La normalisation proposera sans doute de fusionner ces deux types d'entités, selon la règle de la figure 18.30.

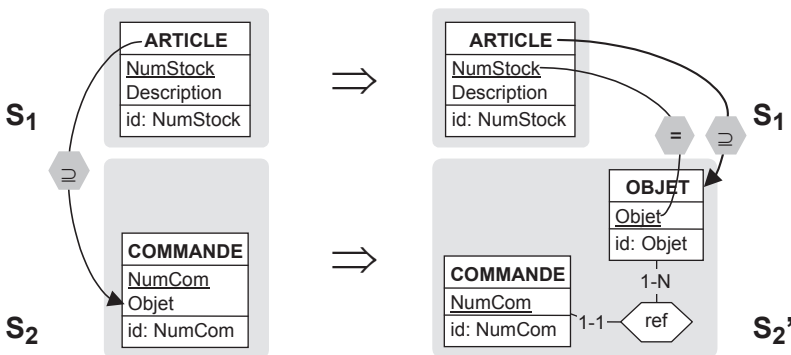


Figure 18.62 - Unification en cas d'une correspondance attribut / type d'entités

b) Correspondance entre un attribut et un rôle

Dans la figure 18.63, l'attribut VEHICULE.vendeur de S₂ est mis en correspondance avec le rôle vendu.SOCIETE dans S₁. Ce pattern est similaire au précédent mais la correspondance est établie avec une variante seulement du type d'entités. On transforme l'attribut vendeur en un type d'entités VENDEUR par représentation des valeurs. SOCIETE et VENDEUR sont mis en correspondance de compréhension mais ceci est insuffisant : il faut encore mettre en correspondance vendu dans S₁ avec le nouveau type d'associations. Nous appelons donc ce dernier vendu, ce qui établit une correspondance d'égalité implicite entre eux. Lors de la fusion, VENDEUR sera sans doute considéré comme un sous-type faiblement spécifique et sera supprimé.

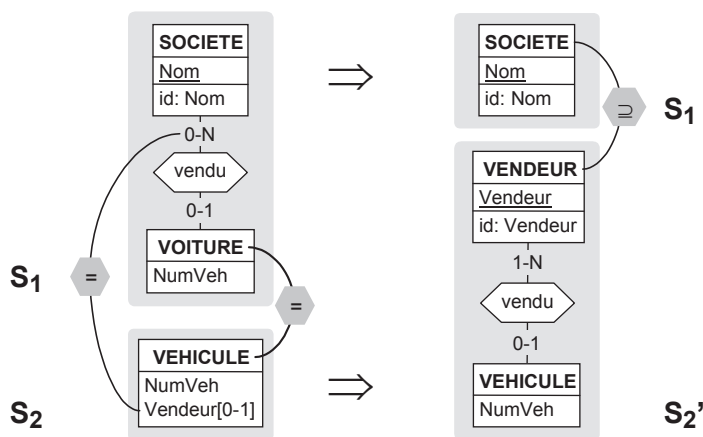


Figure 18.63 - Unification en cas d'une correspondance attribut / rôle

c) Correspondance entre un type d'associations et un type d'entités

Dans la figure 18.64, le type d'associations fabrique du schéma S₁ est en correspondance d'égalité avec le type d'entités PRODUCTION du schéma S₂, ce qui donne à ce dernier le statut de *type d'entités association* (on vérifie qu'il en satisfait les pré-conditions selon 18.7.2).

Les types d'entités ARTICLE et PRODUIT d'une part et UNITE et USINE d'autre part sont en correspondance d'égalité. Le type d'associations fabrique est transformé en un type d'entités FABRICATION et deux types d'associations fonctionnels. La correspondance peut dès lors être établie entre FABRICATION et PRODUCTION. Des correspondances sont établies entre les nouveaux types d'associations et ceux du schéma S₂.

18.9.6 Normalisation et résolution de conflits sémantiques

Les phases de recherche des correspondances et d'unification ont pour effet secondaire l'identification et la résolution de certains conflits : conflits de *nom* (synonymie, homonymie, erreurs typographiques ou orthographiques),

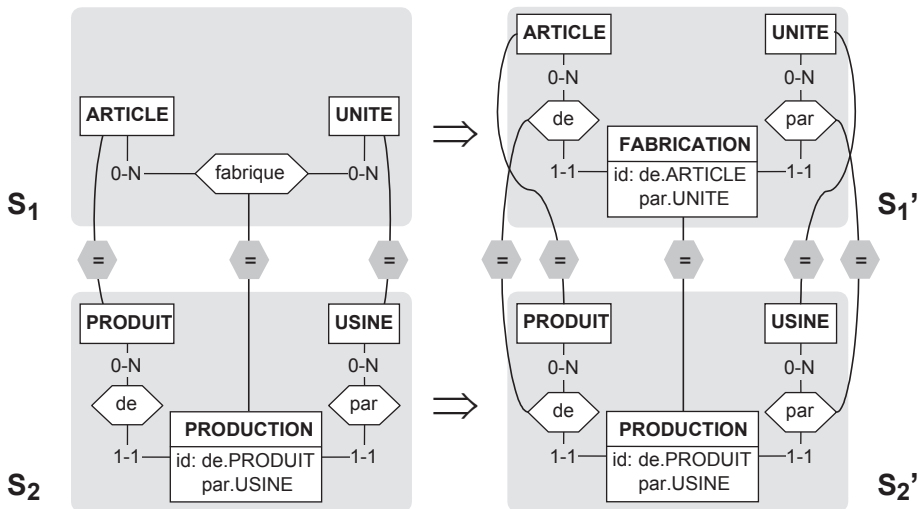


Figure 18.64 - Unification en cas d'une correspondance *type d'entités / type d'associations*

conflits de *modélisation* (les mêmes types de faits sont représentés par des constructions différentes mais équivalentes), conflits de *perspective* (vues plus étroites ou plus générales des mêmes types de faits). Certains de ces problèmes seront résolus par les techniques de normalisation décrites en 18.7.

Il existe cependant des contradictions entre deux schémas qu'il n'est pas possible de réduire par de simples manipulations de schémas comme nous l'avons fait jusqu'ici : il s'agit des **conflits sémantiques**. Ceux-ci exigent de **corriger** un ou les deux sous-schémas conceptuels à intégrer.

Une première source de conflits sémantiques résulte d'un excès de divergence de *perspective*. Alors qu'on peut sans problème réconcilier les cardinalités [0-5] et [0-10] d'attributs en correspondance d'égalité en [0-10], il faut être plus prudent lorsque le conflit porte sur le caractère *obligatoire ou facultatif*, ou sur le caractère *mono-valué ou multivalué*. De même, des types d'associations 1:N et N:1 en correspondance ne se traduisent pas en N:N ou 1:1 sans validation de la part des utilisateurs.

Une seconde source de conflits sémantiques, plus grave, est celle des constructions incorrectes (section 18.6) : la plupart de celles-ci résultent de l'union de deux constructions en elles-mêmes non problématiques. La figure 18.65 montre quatre exemples de conflits sémantiques qui interdisent l'intégration des schémas. Ces exemples sont inspirés de la section 18.6. Le lecteur en imaginera aisément d'autres. Quand convient-il de résoudre les conflits sémantiques ? La littérature suggère de les traiter soit lors de l'unification, soit lors de la fusion proprement dite.

18.9.7 Compléments

L'étude du processus d'intégration que nous avons présentée dans cette section est évidemment loin d'épuiser la question. Signalons quatre prolongements utiles.

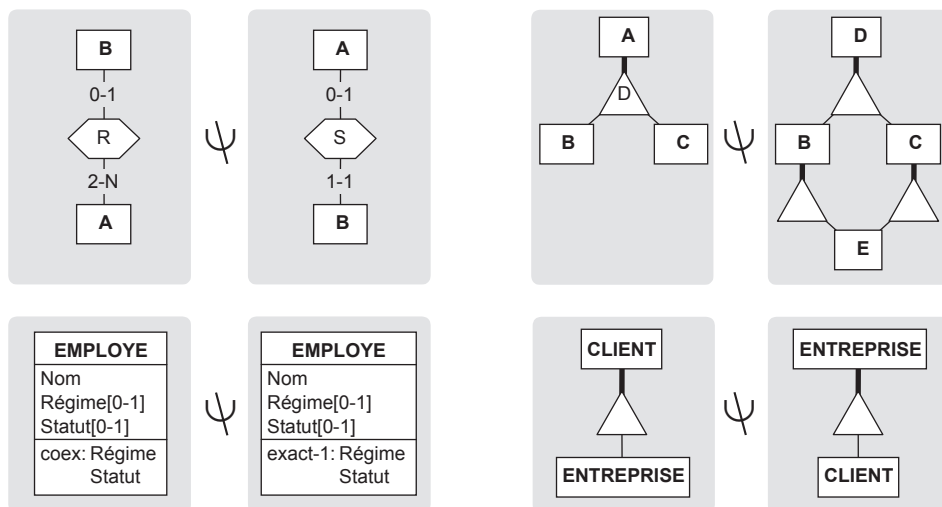


Figure 18.65 - Quatre exemples de conflits sémantiques : les schémas sources ne sont pas intégrables sans modification de la sémantique externe de l'un d'eux

- *Intégration de schémas à faible expressivité.* La richesse du modèle Entité-association implique naturellement une plus grande complexité des règles d'intégration. On pourrait dès lors suggérer de transformer au préalable les schémas à intégrer dans un modèle plus pauvre et donc moins expressif (celui du chapitre 12 par exemple), de procéder à leur intégration puis de normaliser le résultat de manière à reconstituer les constructions expressives à l'aide des techniques étudiées en 18.7.2. De nombreux travaux actuels, qui utilisent des structures de données intrinsèquement *arborescentes*, telles qu'XML ou les ontologies, s'inscrivent (inconsciemment) dans cette direction.
- *Auto-intégration.* L'intégration de deux schémas réduit la redondance des structures entre ceux-ci. Rien n'interdit de procéder à l'intégration de deux parties distinctes d'un même schéma de grande taille. Les techniques étudiées dans cette section sont donc applicables à l'élimination des redondances dans le processus de normalisation.
- *Analyse et intégration.* L'analyse de documents (section 18.2 et plus particulièrement 18.2.4) en vue d'en tirer un schéma conceptuel est intrinsèquement un processus d'intégration. En effet, il s'agit d'enrichir le schéma courant d'une construction élémentaire exprimant la sémantique d'un fragment de texte ou d'un formulaire. Cet enrichissement constitue en soi une intégration du schéma en cours et de la construction élémentaire. Dans ce contexte, les opérations de *restructuration* décrites à la section 13.5.5 ne sont rien d'autre qu'une version simplifiée du processus d'unification.
- *Intégration de bases de données.* Nous avons étudié le problème de l'intégration de schémas relatifs à une base de données en projet, c'est-à-dire auxquels ne correspondent pas encore d'instances. On peut également intégrer des schémas

conceptuels de bases de données existantes, obtenus par exemple par rétro-ingénierie. L'existence de données change la problématique de manière importante. D'une part, les hypothèses sur la nature des correspondances peuvent être validées par une analyse des données. Par exemple, la comparaison des populations de deux objets des schémas permet de décider si une correspondance est d'égalité, de compréhension, d'union ou de différence. D'autre part, le but de l'intégration de bases de données est en général d'intégrer non seulement les schémas mais aussi les données. On est alors confronté à des problèmes nouveaux, liés à la qualité des données existantes.

18.9.8 Étude de cas

On se propose d'intégrer les trois schémas de la figure 18.66. Le schéma 2 semble le plus riche et le plus expressif. Nous le choisirons comme schéma de départ. Les schémas seront pris en compte dans l'ordre

<Schéma 2, Schéma 1, Schéma 3>.

L'analyse comparative des schémas 2 et 1 fait apparaître les correspondances suivantes (figure 18.67), validées par les utilisateurs :

- 1. PUBLICATION $\supseteq$ RAPPORT ; un *rapport* est un cas particulier de *publication*.
- 2. EXEMPLAIRE = COPIE ; termes synonymes.
- 3. de(PUBLICATION, EXEMPLAIRE) $\supseteq$ de(RAPPORT, COPIE ; conséquence de 1 et 2.
- 4. EXEMPLAIRE.CodeExemplaire = COPIE.Numéro ; termes synonymes attachés à des objets synonymes.

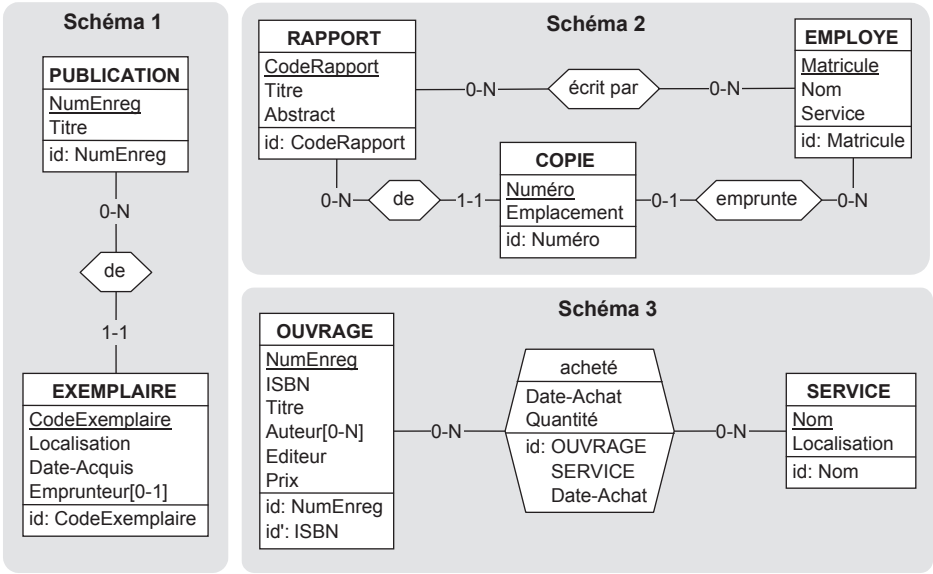


Figure 18.66 - Trois schémas à intégrer

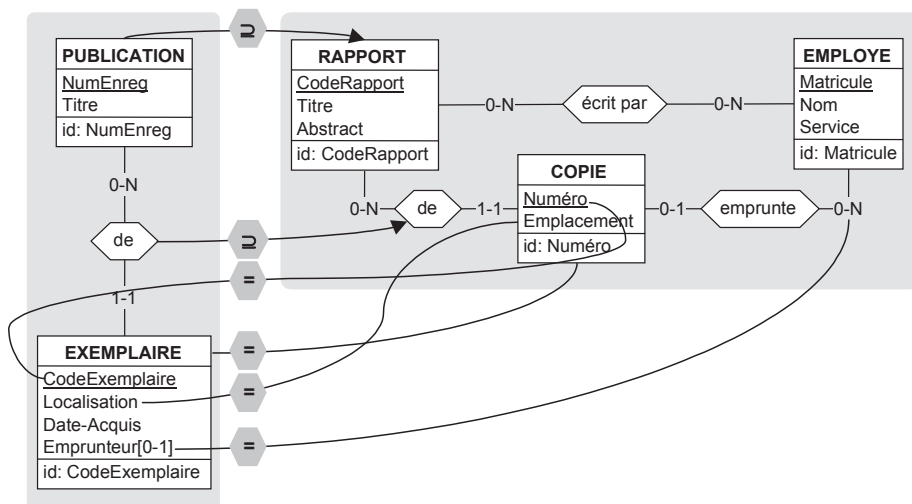


Figure 18.67 - Correspondances entre les schémas 1 et 2

5. EXEMPLAIRE.Localisation = COPIE.Emplacement ; termes synonymes attachés à des objets synonymes.
6. EXEMPLAIRE.Emprunteur = emprunte.EMPLOYE ; une valeur d'Emprunteur représente une instance d'EMPLOYE jouant un rôle dans emprunte.

Ces correspondances sont homogènes, à l'exception de l'égalité entre l'attribut Emprunteur de EXEMPLAIRE et le rôle emprunte.EMPLOYE. L'attribut Emprunteur est transformé en type d'entités EMPRUNTEUR, connecté à EXEMPLAIRE par le nouveau type d'associations emprunte. EMPRUNTEUR est en relation de compréhension avec le type d'entités EMPLOYE, dont il devient un sous-type faiblement spécifique. De ce fait, on normalise en fusionnant EMPLOYE et EMPRUNTEUR.

Nous pouvons à présent compléter le schéma 2' et le schéma 1' transformé. On observe une construction incorrecte dans le schéma final : l'identifiant primaire propre de RAPPORT entre en conflit avec celui, hérité, de PUBLICATION. Il devient secondaire (figure 18.68).

L'intégration des schémas 21 et 3 va introduire une correspondance plus complexe entre deux objets. On relève cinq correspondances (figure 18.69) :

1. PUBLICATION $\supseteq$ OUVRAGE ; OUVRAGE est une sorte de PUBLICATION.
2. OUVRAGE.Auteur $\cup$ écrit.EMPLOYE ; OUVRAGE.Auteur représente une information *similaire* à écrit.EMPLOYE (on rappelle que OUVRAGE et RAPPORT sont deux sortes de PUBLICATION).
3. acheté.Date-Achat = EXEMPLAIRE.Date-Acquis ; il s'agit d'une correspondance hétérogène en raison de la nature de la correspondance entre leurs parents.
4. SERVICE $\supseteq$ EMPLOYE.Service.
5. Il existe entre EXEMPLAIRE et acheté une correspondance qui ne rentre pas dans les catégories étudiées et que nous notons ?? ; nous l'étudierons plus en

détail dans l'annexe A18 sous le nom de *dérivation mutuelle*.

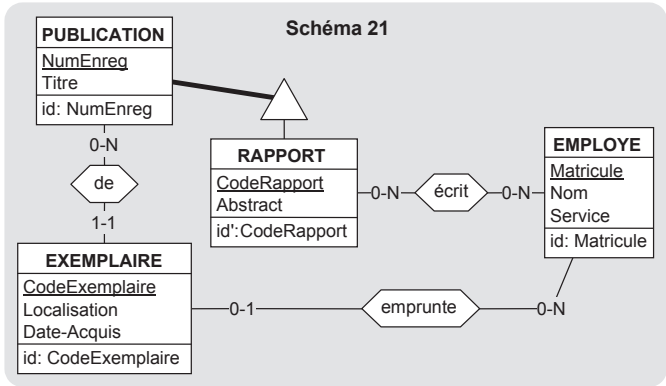


Figure 18.68 - Le schéma 21, résultant de l'intégration des schémas 2 et 1

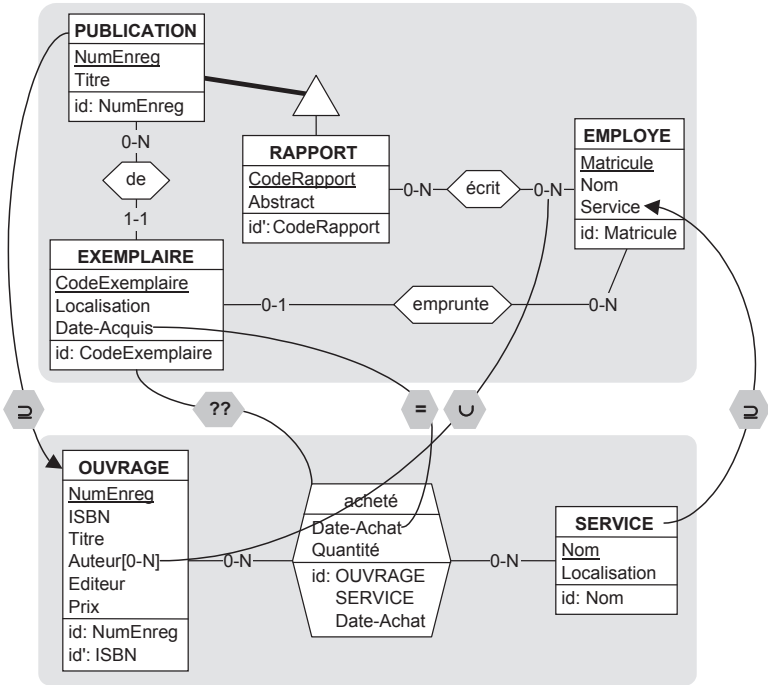


Figure 18.69 - Correspondances entre les schémas 21 et 3

L'unification des schémas 21 et 3 s'effectue comme suit :

1. OUVRAGE est déclaré sous-type de PUBLICATION.
2. L'attribut OUVRAGE.Auteur est d'abord transformé en un type d'entités AUTEUR, identifié par l'attribut Nom, et un type d'associations écrit entre OUVRAGE et AUTEUR; le nouveau type d'entités AUTEUR et EMPLOYE héri-

tent de la correspondance d'union. Par conséquent, un surtype PERSONNE est créée pour les sous-types EMPLOYE et AUTEUR.

3. L'attribut EXEMPLAIRE.Date-Achis est renommé Date-Achat.
4. La correspondance entre EMPLOYE.Service et SERVICE étant hétérogène, on introduit dans le schéma 21 un type d'entités SERVICE et un type d'associations occupe entre SERVICE et EMPLOYE. Il reste alors à fusionner les deux types d'entités SERVICE.
5. La correspondance entre EXEMPLAIRE et acheté est plus complexe que celles que nous avons rencontrées jusqu'ici. Considérons une association (**ouv,ser,dat,qté**) de acheté(OUVRAGE,SERVICE,Date-Achat,Quantité). Elle nous dit que **qté** exemplaires de l'ouvrage **ouv** ont été achetés par le service **ser** à la date **dat**. Ces informations existent déjà dans le schéma 21, à l'exception du service qui a effectué l'achat. Si nous ajoutons au schéma 21 un nouveau type d'associations acheté associant EXEMPLAIRE à SERVICE (introduit à l'étape précédente) alors toute l'information exprimée par acheté du schéma 3 peut être recalculée par agrégation les information du schéma 21.

La figure 18.70 montre le résultat de la fusion des schémas 21 et 3 unifiés. Les caractéristiques communes des sous-types ont été remontées dans le surtype. Il reste quelques questions à régler, notamment les contraintes de sous-types. On propose la *partition* pour PUBLICATION et la *totalité* pour PERSONNE. Cependant, le fait que AUTEUR et EMPLOYE possèdent chacun un identifiant primaire pose problème : certaines entités PERSONNE vont avoir *deux identifiants primaires*. On change donc le statut de l'identifiant de AUTEUR de primaire en secondaire.

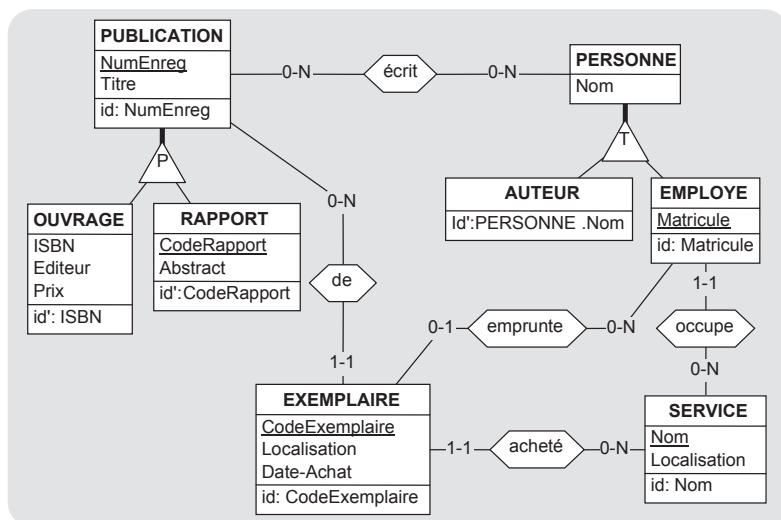


Figure 18.70 - Le schéma global intégré

Note.Un développement plus détaillé du processus d'intégration est proposé à l'annexe A18.

18.10 ANALYSE CONCEPTUELLE À L'AIDE D'UN ATELIER

L'analyse conceptuelle est intrinsèquement un processus créatif basé sur une compréhension approfondie du domaine d'application. Néanmoins, un atelier logiciel peut apporter une aide considérable dans certains aspects complexes ou fastidieux du processus.

L'atelier DB-MAIN offre plusieurs outils utiles à l'analyse conceptuelle. Nous en décrirons brièvement trois.

18.10.1 Transformation de schémas

DB-MAIN propose trois modes de transformation de schémas selon le degré de précision et de contrôle désiré : *transformations élémentaires*, *transformations globales* et *scripts de transformation*.

Transformation élémentaire. Une transformation élémentaire est **un** opérateur appliqué à **un** objet du schéma courant. L'atelier contient une boîte à outils comprenant une trentaine de transformations élémentaires, dont la plupart de celles qui sont utilisées dans cet ouvrage. Leur utilisation est très simple : l'utilisateur sélectionne à l'écran l'objet à transformer, puis choisit la transformation à appliquer (figure 18.71). Des boîtes de dialogues permettent de choisir certains paramètres ainsi que les noms des nouveaux objets.

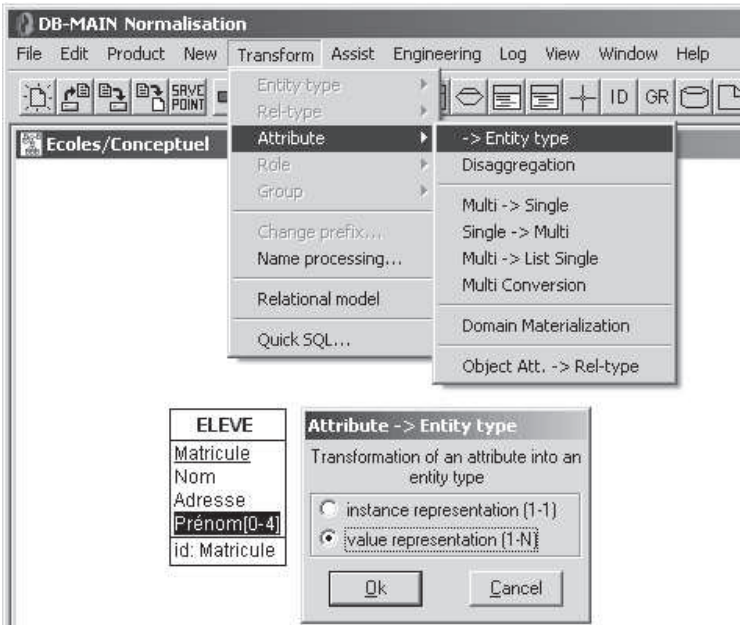


Figure 18.71 - Transformation élémentaire de l'attribut *Prénom* en type d'entités

Transformation globale. Une transformation globale est **un** opérateur appliqué à **tous** les objets du schéma courant qui vérifient une condition. On exécute ces opérateurs via un assistant de transformation qui permet de choisir (1) le type d'objets, (2) les conditions que les objets doivent satisfaire et (3) la transformation à appliquer. DB-MAIN offre deux assistants de transformation globale : l'assistant de base est très simple mais de puissance limitée (figure 18.72). L'assistant avancé est en revanche beaucoup plus riche et plus puissant. Il sera décrit à la section 19.12.

Scripts de transformation. Un script de transformation est une suite de transformations globales dont on peut demander l'exécution pour transformer un schéma. Le développement, la gestion et l'exécution des scripts sont assurés par deux assistants de transformation. DB-MAIN offre pour chacun une collection de scripts prédéfinis pour les processus les plus fréquents. Le script illustré à la figure 18.72 comporte cinq transformations de **normalisation** : (1) transformer les *types d'entités association* en types d'associations, (2) transformer les *types d'entités attribut* en attributs, (3) agréger en un attribut composé les attributs qui font l'objet d'une contrainte de coexistence, (4) ajouter un identifiant primaire technique aux types d'entités dont l'identifiant comporte plus de trois composants, (5) désagréger les attributs composés d'un seul composant.

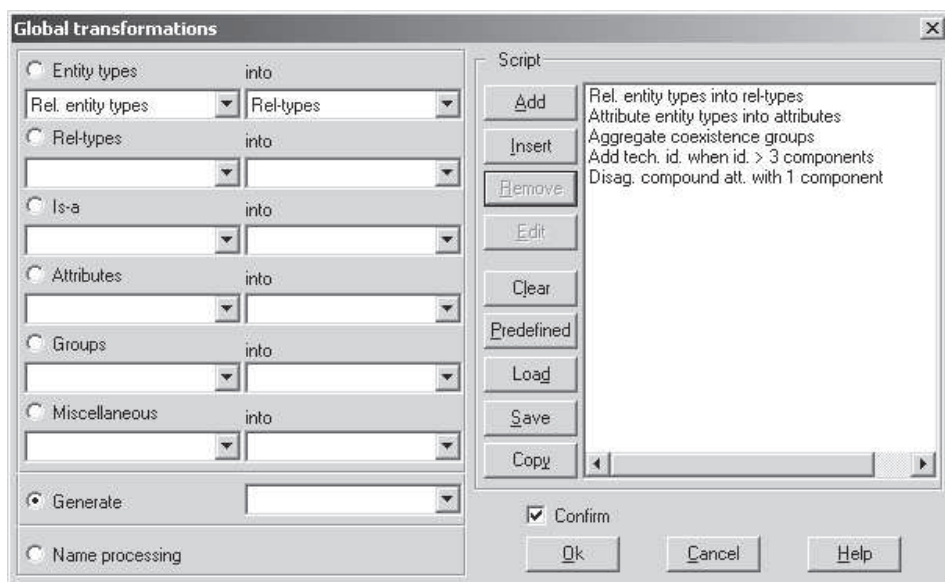


Figure 18.72 - L'assistant de transformation de base. La partie de gauche permet de choisir un type de structure (*Rel[ationship] entity type*) et d'y associer une transformation à exécuter (*[transform into] Rel[ationship] type*). La partie de droite permet de construire, gérer et exécuter des scripts de transformation.

18.10.2 Analyse de schémas

L'analyse d'un schéma admet deux objectifs inverses : (1) rechercher dans le schéma courant toutes les constructions qui **vérifient** des conditions définies et (2) rechercher dans un schéma toutes les constructions qui **violent** des conditions définies. La première variante consiste à **interroger** le schéma via un langage de requête tandis que la seconde permet de valider un schéma par rapport à un ensemble de conditions structurelles. DB-MAIN comporte un assistant d'analyse de schémas offrant les deux modes de fonctionnement. Il permet par exemple de définir sous forme de conditions les principales classes de constructions **erronées** et à **normaliser** étudiées dans ce chapitre et de les identifier dans le schéma courant

18.10.3 Intégration de schémas

Deux outils sont disponibles : l'assistant d'*intégration d'objets* et le processeur de *fusion*. Le premier est interactif et travaille sur deux objets de même nature (figure 18.73) en proposant diverses techniques d'intégration. Le processeur de fusion travaille sur deux schémas unifiés et ajoute au premier les concepts du second qu'il ne contient pas. Les transformations permettent d'unifier les schémas à intégrer.

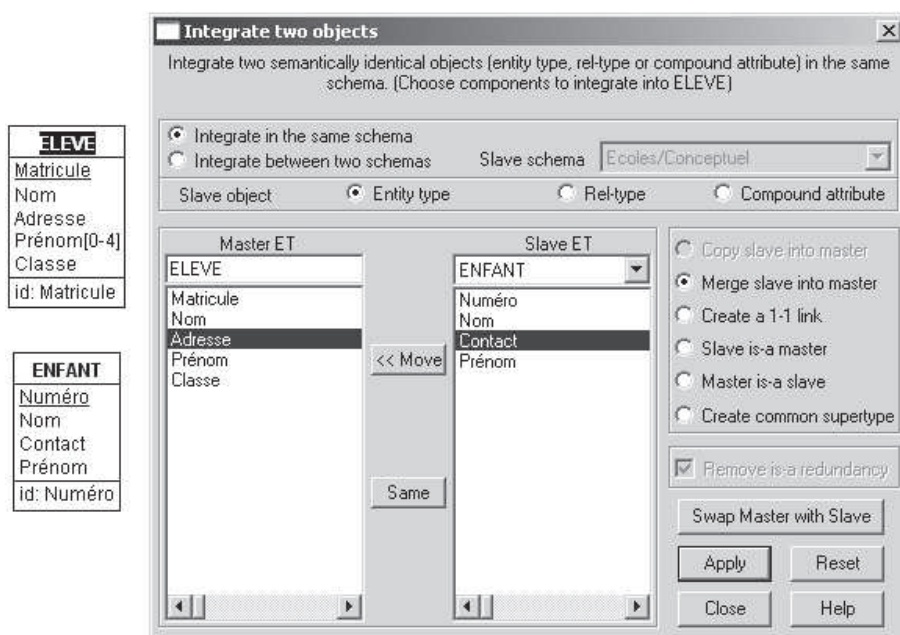


Figure 18.73 - L'assistant d'intégration d'objets

18.11 QUE RETENIR ?

Le processus d'analyse conceptuelle consiste à traduire les besoins des utilisateurs en une spécification formelle précise et correcte. Celle-ci est appelée **schéma conceptuel** du domaine d'application (et aussi de la future base de données).

On propose une stratégie comportant d'abord une analyse des besoins (exprimés sous la forme de textes et de formulaires notamment) produisant des **sous-schémas conceptuels bruts**. Ces derniers sont alors corrigés, normalisés, évalués puis fusionnés (intégrés) en un schéma conceptuel global.

L'**analyse des besoins** exprimés sous la forme de **textes** utilise une analyse linguistique de ceux-ci. Les énoncés sont décomposés en propositions élémentaires qui sont, l'une après l'autre, traduites en objets du modèle EA. On construit ainsi progressivement le schéma des concepts sous-jacents à l'énoncé. Une proposition peut exprimer l'existence d'un concept (proposition unaire) ; une proposition binaire exprime une classification ou une association entre deux concepts, ou encore une propriété d'un concept ; une proposition n-aire exprime une relation entre plus de deux concepts.

L'analyse de **formulaires**, d'écrans ou de pages web fournit des indications sur des assemblages pertinents d'attributs, desquels on peut extraire des schémas de concepts. Ces différents schémas doivent ensuite être intégrés.

La **correction** d'un schéma consiste d'abord à identifier les constructions erronées, et en particulier les erreurs syntaxiques, les constructions non satisfiables et les constructions incohérentes. Ces constructions sont ensuite corrigées.

La **normalisation** consiste à éliminer ou transformer certaines parties maladroites ou non conformes à des critères de qualité d'un schéma de manière à améliorer, par exemple, sa lisibilité, son expressivité, son évolutivité ou encore éliminer des redondances.

La **validation** d'un schéma est à charge des utilisateurs. Elle consiste à vérifier qu'un schéma respecte les besoins de ces utilisateurs. On utilisera notamment des techniques de paraphrase et de prototypage.

L'**intégration** d'un ensemble de schémas consiste à produire un nouveau schéma qui exprime toute la sémantique des premiers, ni plus ni moins, et ce, sans redondance. Pour intégrer deux schémas, on identifie les correspondances entre leurs concepts, on résout les conflits éventuels puis on fusionne leurs composants. Par itération sur l'ensemble des schémas, on produit le schéma global.

18.12 POUR EN SAVOIR PLUS

La littérature relative au processus d'analyse conceptuelle est extrêmement riche et toujours très active, bien qu'elle s'étende sur plus de trois décennies. Aux problèmes classiques qui méritent toujours qu'on s'y intéresse (la question de l'intégration reste ouverte encore aujourd'hui) s'ajoutent de nouveaux défis issus à la fois des nouvelles technologies et des nouveaux contextes d'utilisation et de développement des bases de données. Parmi les ouvrages généralistes qui ont traité cette question, on citera en particulier [Batini, 1992], [Bodart, 1994] (pour la méthode IDA),

[Nancy, 1996] (pour la méthode Merise), [Blaha, 1998] (méthodologie appropriée aux modèles UML), [Akoka, 2001] et [Halpin, 2008], mais il en existe beaucoup d'autres. Les actes des conférences internationales *Entity-relationship* (renommées *International Conference on Conceptual Modeling*) et *Conference on Advanced Information System Engineering* (CAiSE) contiennent de nombreuses références aux processus étudiés dans ce chapitre.

Quelques pistes destinées au lecteur désireux d'approfondir certaines thématiques spécifiques :

- *Utilisation du langage naturel*. Le langage naturel peut être utilisé pour exprimer les besoins des utilisateurs [Chen, 1983], [Gomez, 1999] mais aussi pour valider des spécifications conceptuelles via, notamment, leur paraphrase [Danialis, 1992]. Certains modèles conceptuels s'inspirent directement de la linguistique, tels que ORM (*Object-Role Model*) [Halpin, 2008]. Voir aussi [Storey, 2004], qui étudie spécifiquement l'interprétation des verbes dans le modèle Entité-association. Les interactions entre bases de données et langage naturel font l'objet d'une série de conférences organisée depuis 1995, *Applications of Natural Language to Data Bases* ou *NLDB*, dont on consultera les actes.
- *Analyse de formulaires*. On retrouvera l'idée d'analyser le contenu informationnel de formulaires dans [Choobineh, 1992], [Batini, 1992], [Vilz, 2006], [Lukovic, 2007]. [Ramdoyal, 2011] décrit une méthode outillée basée entre autres sur des formulaires dessinés par les utilisateurs.
- *Patrons de conception*. Lire par exemple [Blaha, 2010].
- *Qualité des schémas conceptuels*. On consultera notamment [Moody, 1998] [Calero, 2002], [Genero, 2002], et, plus spécifiquement, [Vigna, 2004] pour les problèmes de satisfiabilité et [Bowers, 2003] pour la redondance de types d'associations. Une approche basée sur des classes de patterns défectifs est décrite dans [Lemaitre, 2011].
- *Intégration de schémas*. Quelques références de base : [Parent, 1998], [Rahm, 2001], [Noy, 2004], [Shvaiko, 2005]. On trouvera une riche bibliographie à l'adresse <http://www.ontologymatching.org/publications.html> (juin 2015) ainsi que dans [Shvaiko, 2005].
- *Méthodologies spéciales*. Lors de l'analyse d'un domaine d'application, l'accent peut être mis sur certains aspects spécifiques, réclamant ainsi des modèles et des méthodes adaptées. Citons par exemple les données à dimension temporelle [Theodoulidis, 1991], [Lee, 1997], [Detienne, 2001], les données spatio-temporelles [Parent, 2006], des entrepôts de données [Vaisman, 2014] et les systèmes d'information réactifs [Ceri, 1997].
- *Approches transformationnelles*. Nous avons à plusieurs reprises utilisé des techniques de transformation de schémas. On trouvera dans [Hainaut, 2006] une introduction à ces techniques. L'annexe A25 propose un catalogue étendu comprenant notamment les transformations de schémas utilisées dans cet ouvrage.

- *Méthodes agiles*. Des méthodes plus légères, analogues à celles qui sont développées pour la production des programmes (*extreme programming*, *agile programming*), ont également été proposées récemment. [Proper, 1997] et [Ambley, 2007] notamment suggèrent que le développement d'une base de données soit organisé comme une évolution continue.

18.13 EXERCICES

- 18.1 *Toute personne possède deux parents biologiques, lesquels sont des personnes*. Discuter cette proposition dans le contexte de l'analyse conceptuelle.

- 18.2 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). À 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.

- 18.3 Proposer un schéma conceptuel traduisant les concepts de l'énoncé suivant.

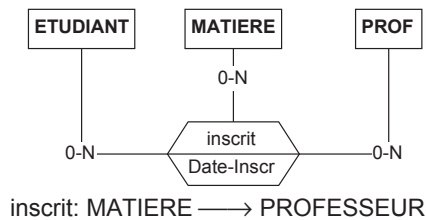
Dans un réseau de bibliothèques, chacune d'elles commande des exemplaires d'ouvrages repris dans une liste de référence. Chaque commande est numérotée (de manière à être identifiable) et reprend la quantité, la date et l'ouvrage concerné. Une bibliothèque n'envoie pas plus d'une commande d'un même ouvrage le même jour.

Un ouvrage peut être publié par un éditeur pendant une certaine période. À tout moment, l'éditeur d'un ouvrage peut céder ses droits de publication à un

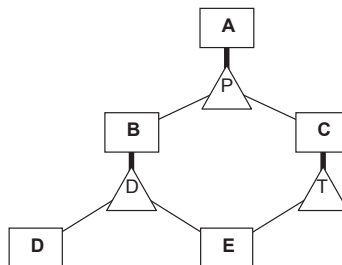
autre éditeur (il peut même les réacquérir plus tard). D'un ouvrage, un éditeur qui en possède les droits peut tirer, à une certaine date, un nombre déterminé d'exemplaires. Si, après au moins 3 mois, les ventes s'avèrent meilleures que prévu, l'éditeur effectuera un nouveau tirage (et ainsi de suite). Un ouvrage est caractérisé par un numéro ISBN qui l'identifie, son titre et sa langue. Un ouvrage est soit un original (auquel cas il est caractérisé par un genre), écrit par un ou plusieurs auteurs, soit une traduction d'un original. Dans ce dernier cas, l'ouvrage est traduit par un ou plusieurs auteurs (pour simplifier, vrais auteurs et traducteurs sont considérés comme des auteurs). Une bibliothèque est identifiée par son nom et sa localité. Elle a une adresse et un responsable.

Un éditeur est identifié par son nom et caractérisé par son adresse et son pays d'origine. Cependant, son adresse peut changer au cours du temps. L'adresse est accompagnée d'un numéro de téléphone. Un auteur porte un code unique et est caractérisé par son nom et son prénom.

18.4 Normaliser, si nécessaire, le type d'associations ci-dessous.



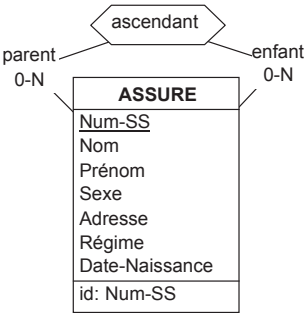
18.5 En utilisant le concept de *satisfiabilité*, simplifier le schéma ci-dessous :



18.6 Proposer un schéma conceptuel minimal intégrant les schémas ci-dessous

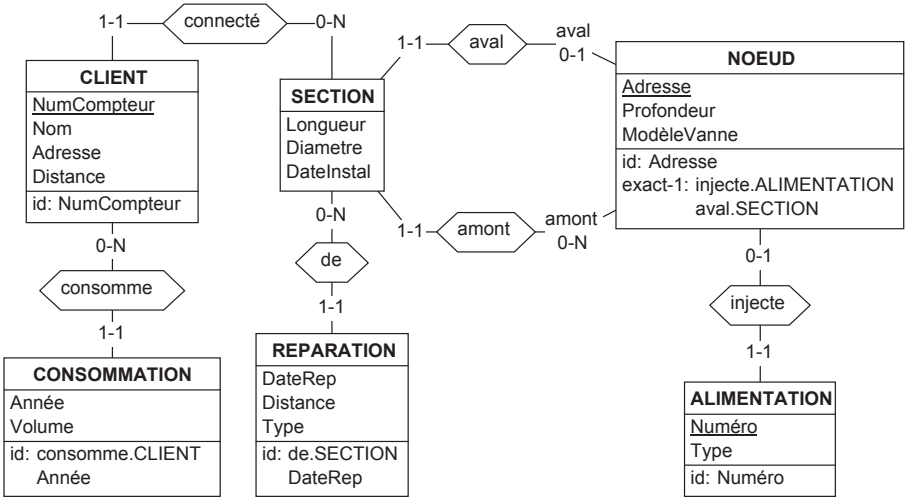
ELEVE
<u>Num-Inscript</u>
Nom
Adresse
Prénom
Date-Naissance
Ecole
Année-Scolaire
Père[0-1]
Mère[0-1]
id: Num-Inscript

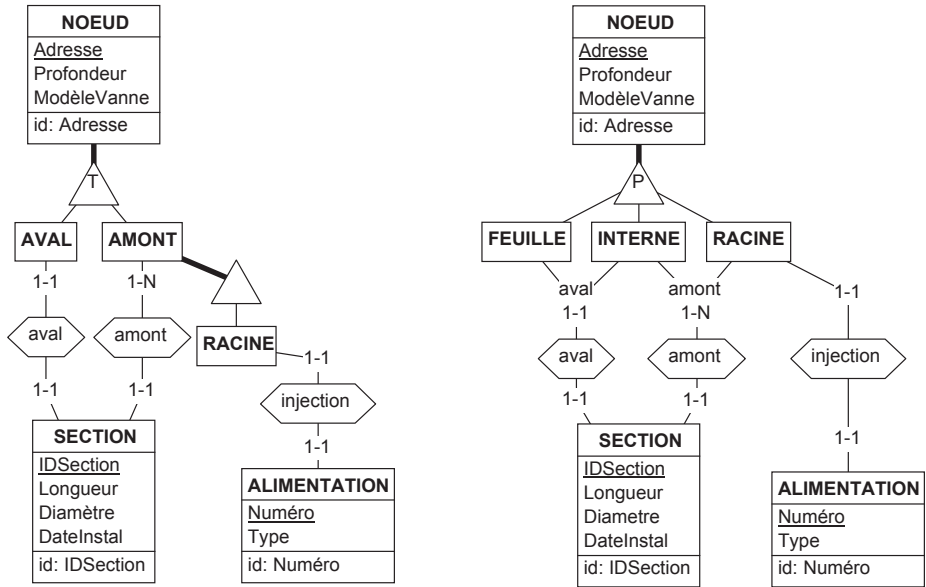
EMPLOYE
<u>Matricule</u>
Nom
Prénom
Adresse
Service
Date-Engagement
Date-Naissance
Prénom-Enfant[0-10]
id: Matricule



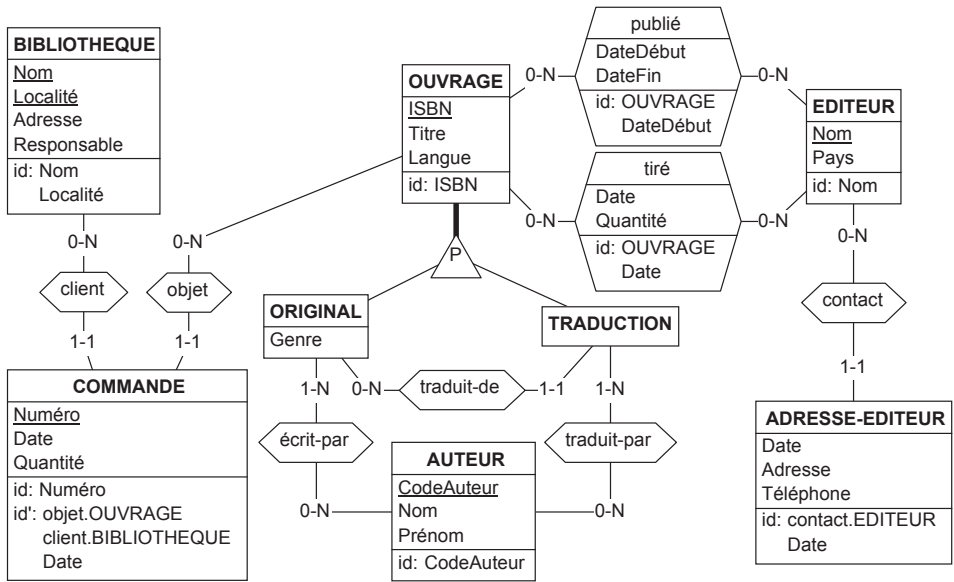
Exercices corrigés

- **Exercice 18.1.** Introduire dans le schéma, comme une analyse naïve pourrait le suggérer, un rôle enfant de cardinalité [2-2] rendrait le type d'entités PERSONNE **non satisfiable** (section 18.6.2/c). On se souviendra qu'un schéma conceptuel ne représente pas à proprement parler un domaine d'application, mais plutôt la connaissance qu'on en a. Comme on ne connaît pas les parents biologiques de chaque personne, on dégradera la contrainte en [0-2].
- **Exercice 18.2.** Une solution simplifiée suivie de deux variantes plus précises.





• **Exercice 18.3.**

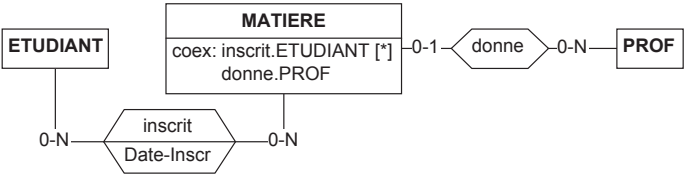


• **Exercice 18.4.** La sémantique interne du type d'associations inscrit est la suivante, compte tenu de son identifiant implicite :

inscrit(ETUDIANT, MATIERE, PROF, Date-Inscr)

inscrit: MATIERE → PROF

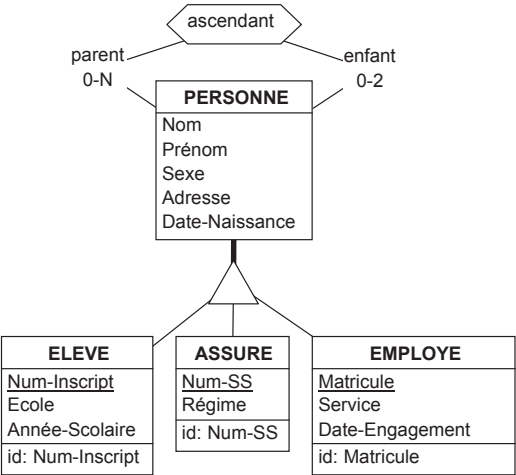
Cet identifiant n'est pas minimal : on peut lui retirer PROF. Cette relation n'est pas normalisée. La décomposition de ce schéma sous 3^e forme normale correspond au schéma ci-dessous. La contrainte de coexistence disparaît si on admet que *toute matière est dispensée par un professeur*.



- **Exercice 18.5.** En raison de la contrainte **P** sur A, le type d'entités E est non satisfiable. Les populations de C et E sont les mêmes (contrainte **T** sur C). C est donc lui aussi non satisfiable. Par conséquent, les populations de A et B sont les mêmes. On peut donc fusionner A et B en AB, qui regroupe leurs attributs, rôles et contraintes. Il vient donc :



- **Exercice 18.6.**



Des matériaux complémentaires ainsi que des exercices supplémentaires sont disponibles à l'annexe A18.

Chapitre 19

Conception logique d'une base de données relationnelle

La conception logique consiste à produire un schéma qui soit conforme au modèle d'une famille de SGBD et équivalent à un schéma conceptuel source. On réalise ce processus par une séquence de transformations appliquées au schéma conceptuel, jusqu'à ce que le résultat soit conforme au modèle cible. Ce chapitre est consacré à la production de schémas relationnels (SQL2) et relationnels objet (SQL3) à partir d'un schéma conceptuel Entité-association.

19.1 INTRODUCTION

Le procédure décrite au chapitre 14 comporte deux tâches distinctes : la conversion du schéma conceptuel en un schéma de tables et la traduction de ce dernier dans le langage SQL DDL. La simplicité du modèle conceptuel utilisé nous autorisait à regrouper ces tâches en un seul processus, d'ailleurs assez direct.

On peut comprendre que l'usage d'un modèle Entité-association plus riche va nécessiter des règles de traduction plus complètes et inmanquablement plus complexes. Nous constaterons qu'il y a souvent plus d'une manière de traduire une construction conceptuelle en structures de tables. De même, un schéma de tables pourra se traduire en SQL de plusieurs manières, qu'il nous faudra comprendre et évaluer afin que nous puissions faire des choix judicieux. La décomposition du processus de production du schéma de tables en processus plus spécialisés est donc naturelle dès lors qu'on se propose de produire d'une manière professionnelle des

bases de données de taille réaliste. Comme annoncé au chapitre 15, nous procéderons comme suit :

- le schéma conceptuel est d'abord transformé en un *schéma logique* (ici un schéma de tables) équivalent par le processus de **conception logique**, objet du présent chapitre ;
- le schéma logique est ensuite complété de constructions physiques qui conféreront à la future base de données de bonnes performances ; le résultat s'appellera le *schéma physique* et sera produit par le processus de **conception physique** (chapitre 20) ;
- le schéma physique sera enfin traduit dans le langage du SGBD en un *texte SQL DDL* par le processus de **production de code** (chapitre 21).

De même qu'un schéma conceptuel s'exprime dans un formalisme approprié, en l'occurrence le modèle Entité-association, le schéma logique sera défini selon un modèle logique qui correspond aux structures de données du SGBD choisi. Nous avons jusqu'ici admis l'usage exclusif d'un SGBD relationnel. Dans la pratique, nous pourrions également choisir de construire une base de données relationnelle objet, orientée objet, XML, orientée document ou tout simplement un ensemble de fichiers classiques. À chaque famille de SGBD correspond ce qu'on appelle un *modèle logique*.

Le processus de conception logique peut être défini assez simplement (figure 19.1) : il consiste à transformer toutes les constructions du schéma conceptuel qui ne sont pas conformes au modèle du SGBD, en constructions compatibles avec ce modèle, et ce, tout en préservant toutes les informations du schéma conceptuel, c'est-à-dire sa sémantique. Comme nous l'avons vu au chapitre 14, un type d'associations 1:1 ou 1:N sera transformé en une clé étrangère et un type d'associations N:N en une table association munie de deux clés étrangères.

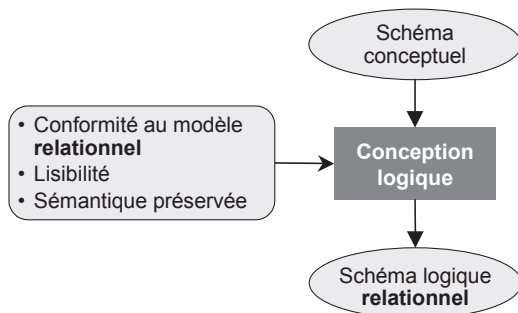


Figure 19.1 - Le processus de conception logique selon le modèle cible relationnel

Dans la section suivante, nous décrirons d'une manière générale les structures de données du modèle logique relationnel. Nous examinerons ensuite comment transformer chacune des nouvelles constructions introduites dans le modèle Entité-association étendu en structures relationnelles. Nous proposerons alors une procédure

globale de transformation d'un schéma conceptuel en un schéma relationnel. Le schéma qui nous servira d'illustration est celui de la figure 16.42.

L'annexe A19, propose une méthode de conception logique adaptée au modèle *relationnel objet*. Nous en reprendrons certaines techniques dans ce chapitre.

19.2 LE MODÈLE LOGIQUE RELATIONNEL

Le lecteur pourrait s'étonner qu'on revienne sur un modèle, non seulement relativement simple, mais en outre largement étudié dans les chapitres qui précèdent. C'est que les raisonnements que nous allons développer vont s'appuyer sur une approche très en vogue aujourd'hui, bien que très ancienne : l'approche *transformationnelle*.

L'idée est simple : tout schéma est le résultat de l'application d'opérations de **transformation** aux objets d'un schéma source. Si le jeu d'opérateurs est convenablement choisi, alors le schéma résultant contient, sous une autre forme, toute l'information du schéma source. Une transformation consiste à remplacer un ensemble de constructions par un autre. Ce remplacement n'est pas arbitraire : on remplace une construction qui ne respecte pas des conditions, ou contraintes, spécifiques par d'autres qui satisfont ces conditions. Par exemple, un schéma conceptuel ne respecte pas les contraintes des schémas relationnels. En particulier, il comporte des types d'associations 1:N, constructions inconnues dans le modèle relationnel. En appliquant une transformation qui remplace chaque type d'associations 1:N par une clé étrangère (figure 14.2), on rapproche ce schéma du modèle relationnel. Entre le schéma source et le schéma relationnel final, il existe une suite de schémas *hybrides*, qui sont en partie *Entité-association* et en partie *relationnels*. D'où la nécessité d'un modèle permettant de représenter ces schémas hybrides. La figure 14.1 montre qu'une simple adaptation du modèle Entité-association fera l'affaire : à l'exception de l'ombrage dans le schéma de droite, purement cosmétique, le graphisme de la table est identique à celui du type d'entités.

19.2.1 Le modèle relationnel de base

Ces observations nous permettent de préciser ce qu'on entend par un *schéma relationnel*. Un schéma est dit **relationnel** s'il respecte les contraintes de la figure 19.2.

Modèle relationnel de base

1. le schéma comporte des types d'entités (appelés *tables*),
2. tout type d'entités possède au moins un attribut (appelé *colonne*),
3. un attribut est monovalué et atomique ; il est obligatoire ou facultatif,
4. les seules contraintes sont celles qui sont induites par les identifiants (*identifiant primaire* ou *secondaire*) et les attributs de référence (*clés étrangères*),
5. le schéma relationnel ne contient pas d'autres constructions,
6. les noms des objets respectent la syntaxe SQL.

Figure 19.2 - Définition du modèle relationnel de base¹

Tout schéma Entité-association qui satisfait ces contraintes peut être qualifié de relationnel. Il ne contient que des constructions directement exprimables en SQL DDL, de sorte que sa traduction en SQL est, en principe, un jeu d'enfant².

19.2.2 Le modèle relationnel enrichi

Le modèle relationnel est évidemment très pauvre comparé au modèle Entité-associations. Les constructions et les contraintes du schéma conceptuel qui ne peuvent se traduire en structures relationnelles seront ajoutées au schéma logique sous la forme d'annotations. Comme nous le verrons au chapitre 21, ces contraintes additionnelles seront rigoureusement traduites en SQL via des prédicats (*check*), des déclencheurs (*triggers*) ou des procédures SQL (*stored procedures*).

Nous nous permettrons cependant, pour des raisons pragmatiques, d'ajouter au modèle relationnel un petit nombre de contraintes qui, soit sont d'une importance majeure, soit sont assez communes et particulièrement aisées à traduire en SQL. Nous en retiendrons deux.

a) Les contraintes référentielles totales (clés étrangères *equ*)

Il s'agit de contraintes assez communes mais délicates, que nous avons déjà rencontrées au chapitre 3, lorsque nous avons décrit les contraintes d'inclusion (section 3.5). Nous y avons rencontré une situation soumise à deux contraintes inverses, l'une spécifiant qu'il n'existait *pas de détail sans commande* et l'autre qu'il n'existait *pas de commande sans détail* :

DETAIL[NCOM] ⊆ COMMANDE[NCOM]
COMMANDE[NCOM] ⊆ DETAIL[NCOM]

Ces deux propriétés se traduisaient par une contrainte d'égalité :

COMMANDE[NCOM] = DETAIL[NCOM]

Nous traduirons dans le modèle logique cette propriété par une *clé étrangère totale*, qui sera notée **equ** (*equal*) au lieu de **ref** (figure 19.3).

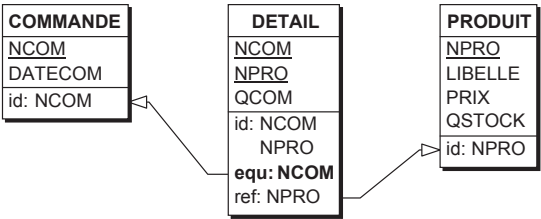


Figure 19.3 - Clé étrangère totale (equ)

1. Les particularités communes des modèles Entité-association et relationnel ne sont pas reprises. Tel est le cas de la propriété : *tous les composants d'un identifiant primaire sont obligatoires*.
2. Nous reviendrons au chapitre 21 sur la notion de *jeu d'enfant* dans ce contexte.

b) Les contraintes d'existence

Nous rencontrerons ces contraintes notamment dans la traduction des relations *is-a*. Leur prise en compte explicite dans le modèle se justifie d'autant plus qu'elle pourront se traduire aisément par des prédicats *check*.

On en déduit ainsi le *modèle relationnel enrichi* de la figure 19.4. C'est ce modèle que nous utiliserons dans la suite de ce chapitre.

Modèle relationnel enrichi

1. le schéma comporte des types d'entités (appelés *tables*),
2. tout type d'entités possède au moins un attribut (appelé *colonne*),
3. un attribut est monovalué et atomique; il est obligatoire ou facultatif,
4. les seules contraintes sont celles qui sont induites par les identifiants (*identifiant primaire* ou *secondaire*), les attributs de référence (*clés étrangères ref*) et les attribut de référence totaux (*clés étrangères equ*) ainsi que les contraintes d'existence (*coex*, *excl*, *at-least-1*, *exact-1*, *if*),
5. le schéma relationnel ne contient pas d'autres constructions,
6. les noms des objets respectent la syntaxe SQL.

Figure 19.4 - Définition du modèle relationnel enrichi

À l'inverse, un schéma **n'est pas relationnel** dès qu'on y trouve des constructions du modèle Entité-association qui ne sont **pas reprises** dans le modèle relationnel. Les principales constructions non valides sont reprises à la figure 19.5.

Constructions non relationnelles

1. les relations *is-a*,
2. les types d'entités sans attributs,
3. les types d'associations,
4. les attributs multivalués,
5. les attributs composés,
6. les contraintes autres que celles des identifiants, des attributs de référence (éventuellement totaux) et d'existence,
7. les noms d'objets non conformes à la syntaxe SQL.

Figure 19.5 - Les constructions non relationnelles à transformer

Le processus de conception logique consiste à transformer dans un schéma conceptuel les constructions non conformes du schéma 19.5 en constructions strictement comprises dans le modèle relationnel (figures 19.2 ou 19.3). Nous étudierons la transformation de chacune des constructions non conformes puis nous intégrerons ces transformations en une procédure globale sous la forme d'un **plan de transformation** (*recette de cuisine* pour les concepteurs pressés).

19.3 REPRÉSENTATION DES ATTRIBUTS

On examinera la représentation des attributs simples (atomiques et monovalués) puis celle des attributs composés et/ou multivalués.

19.3.1 Représentation d'un attribut simple

Un attribut simple de niveau 1 se traduit par une colonne dont le type SQL est plus proche de son type conceptuel. On examinera à la section 19.3.5 la représentation des contraintes de type et de domaine auxquelles un attribut peut être soumis.

19.3.2 Représentation d'un attribut composé

On traite ici le cas d'un attribut monovalué. Il existe trois familles de techniques dont on évaluera les avantages et les inconvénients. Nous les appliquerons à l'attribut composé Adresse du type d'entités EMPRUNTEUR (figure 19.6).

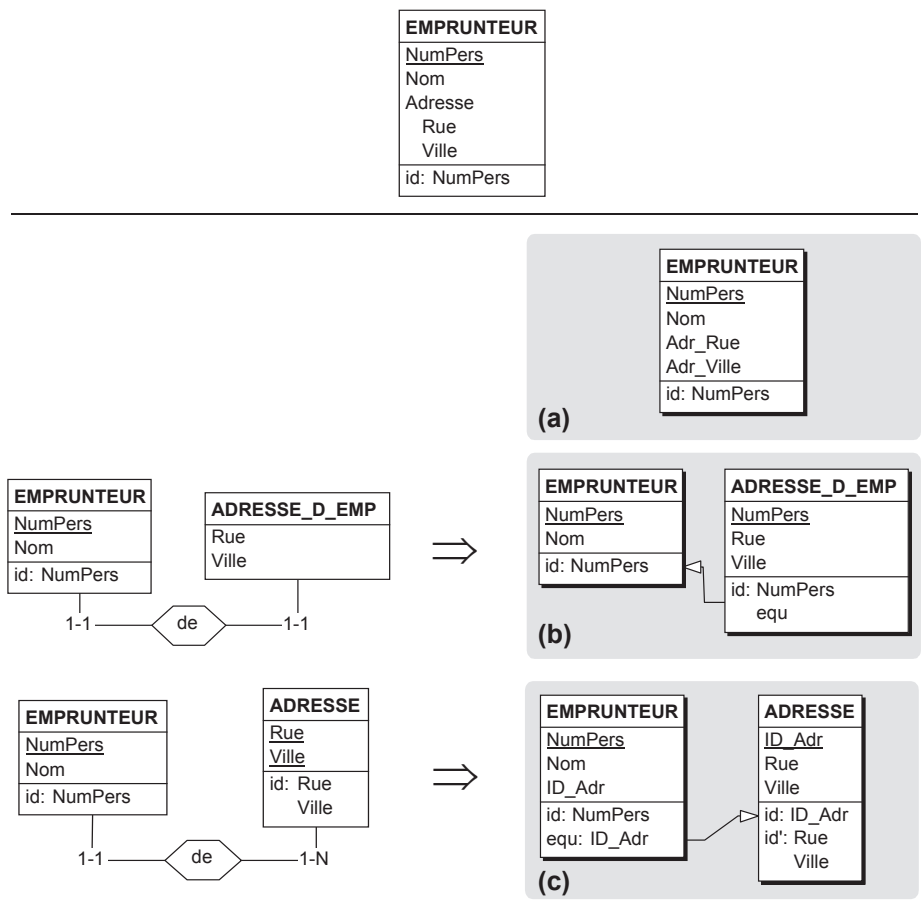


Figure 19.6 - Trois représentations relationnelles de l'attribut composé Adresse

On notera que les deux dernières techniques ne sont que des cas particuliers de celles qui seront proposées pour les attributs multivalués.

- *Désagrégation.* La technique la plus simple consiste à désagréger d'attribut composé, lequel est remplacé par ses composants. On rappelle l'origine commune de ceux-ci en préfixant leur nom d'une abréviation de l'attribut d'origine. L'attribut Adresse est représenté par deux colonnes représentant ses composants Rue et Ville (schéma **a**). Si l'attribut source est facultatif, les composants le sont également et sont soumis à une contrainte de coexistence.
- *Transformation en type d'entités (représentation des instances).* L'attribut est extrait de son type d'entités d'origine pour être attaché à un nouveau type d'entités. Le type d'associations 1:1 qui relie les deux types d'entités est transformé en une clé étrangère identifiante. Chaque entité du nouveau type représente **une instance** de l'attribut source. L'attribut Adresse est représenté par la table ADRESSE_D_EMP qui reprend toutes les **instances** d'Adresse (schéma **b**).
- *Transformation en type d'entités (représentation des valeurs).* Ici encore, l'attribut est extrait de son type d'entités d'origine pour être attaché à un nouveau type d'entités. Le type d'associations 1:N qui relie les deux types d'entités est ensuite transformé en une clé étrangère basée sur un identifiant technique. Chaque entité du nouveau type représente **une valeur distincte** de l'attribut source. L'attribut Adresse est représenté par la table ADRESSE qui reprend toutes les **valeurs distinctes** d'Adresse (schéma **c**). La contrainte référentielle forte equ peut être assouplie en ref en admettant la cardinalité [0-N] du rôle de ADRESSE (voir sur ce point la section 19.7.6).

Le cas des attributs composés facultatifs

Si un attribut composé **facultatif** est transformé par désagrégation, tous les composants qui le remplacent deviennent également facultatifs mais doivent être soumis collectivement à une contrainte de coexistence. S'il est transformé en type d'entités, alors le caractère facultatif est transmis au rôle que joue le type d'entités source : [0-1] au lieu de [1-1].

La question se complique lorsque certains³ composants de l'attribut composé sont eux-mêmes facultatifs (figure 19.7). Si la transformation en type d'entités reste intuitive (schéma de gauche), il n'en va pas de même de la désagrégation, qui réclame un peu plus d'attention. On raisonne comme suit : les composants obligatoires de Téléphone se comportent de manière normale (ils sont facultatifs et coexistants), en revanche, CodePays, également (et doublement) facultatif, ne peut exister que si les deux autres sont aussi présents. Il s'agit donc d'une contrainte d'implication :

$$\text{existe}(\text{CodePays}) \Rightarrow \text{existe}(\text{PréfixeRégional}) \wedge \text{existe}(\text{Numéro}),$$

ce qu'on traduit par la contrainte d'existence if (schéma de droite)⁴.

3. Certains mais pas tous. Qu'en est-il lorsque **tous** les composants de l'attribut composé facultatif sont eux-mêmes facultatifs ?

Grâce à la contrainte de coexistence, la contrainte d'implication peut être réduite à :

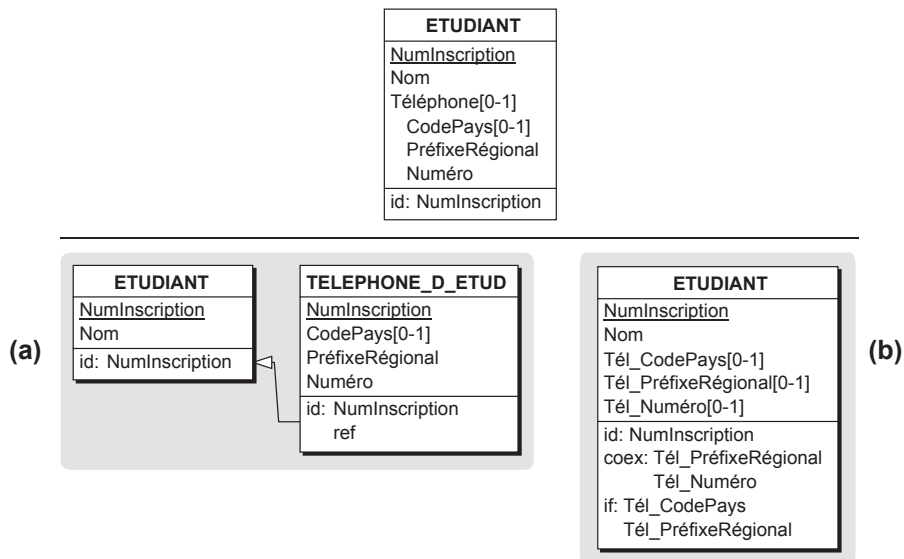
$$\text{existe}(\text{CodePays}) \Rightarrow \text{existe}(\text{PréfixeRégional})$$


Figure 19.7 - Représentation d'un attribut composé facultatif dont certains composants sont facultatifs

Évaluation des techniques

- **Désagrégation.** Cette technique produit un résultat **simple et concis**. Elle favorise la minimalité du schéma (pas de table additionnelle) mais réduit son expressivité (on perd la notion d'adresse), et son évolutivité. Il sera difficile dans l'avenir de donner au concept d'adresse un statut indépendant lui permettant de recevoir de nouveaux attributs ou d'être associé à d'autres type d'entités. Contrairement aux deux autres, cette technique est applicable à tout attribut, quel que soit son niveau.
- **Type d'entités / représentation des instances.** Le schéma résultant est moins concis (il comporte une table additionnelle) mais **plus expressif** que celui produit par la technique précédente. Si le concept d'adresse est plus visible, il n'est cependant représenté que comme un composant d'emprunteur. La table ADRESSE_D_EMP n'offre pas de valeur en dehors d'un complément à la table EMPRUNTEUR. Cette technique correspond à une division verticale de la table EMPRUNTEUR. Elle ne s'applique qu'aux attributs de niveau 1.
- **Type d'entités / représentation des valeurs.** Le résultat est **le plus expressif** et **le plus évolutif**. Il est facile d'ajouter des composants au concept d'adresse et de

4. Chose curieuse : la contrainte if ne reprend qu'une partie de l'implication. Certainement une erreur de l'auteur. Qu'en pensez-vous ?

l'associer à d'autres types d'entités (par ajout de colonnes ou de tables). Cette table possède une réelle valeur intrinsèque, indépendante de la table EMPRUNTEUR. Cette technique ne s'applique qu'aux attributs de niveau 1.

19.3.3 Représentation d'un attribut multivalué

Cette question a déjà été discutée, et d'ailleurs résolue, bien que dans un tout autre contexte, au chapitre 13, section 13.5.6. Représenter une propriété multivaluée dans un modèle qui n'admet que des attributs monovalués est en effet un problème similaire à celui qui nous occupe ici. Le lecteur relira cette section, et en particulier les figures 13.13 et 13.14 qui décrivent les deux techniques que nous allons reprendre pour traiter, à titre d'illustration, l'attribut MotClé de DOCUMENT (figure 19.8)

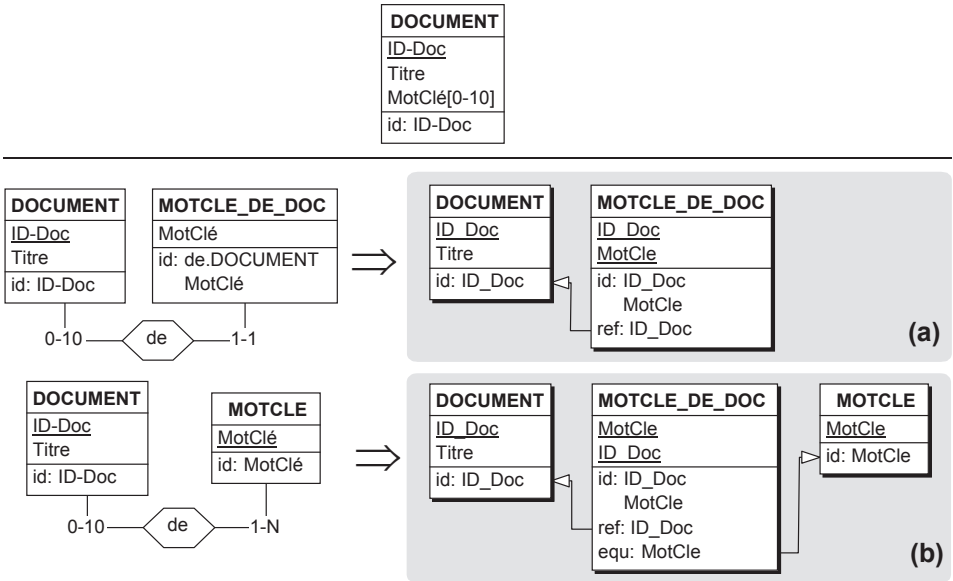


Figure 19.8 - Deux transformations relationnelles de l'attribut multivalué MotClé

- **Transformation en type d'entités (représentation des instances).** L'attribut MotClé est extrait de son type d'entités d'origine DOCUMENT pour être attaché à un nouveau type d'entités MOTCLE_DE_DOC. Le type d'associations 1:N qui relie les deux types d'entités est ensuite transformé en une clé étrangère. Chaque entité du nouveau type représente une instance de l'attribut source MotClé. Ce dernier est représenté par la table MOTCLE_DE_DOC qui reprend toutes les **instances** de MotClé (schéma a).
- **Transformation en type d'entités (représentation des valeurs).** L'attribut MotClé est extrait du type d'entités DOCUMENT pour être attaché à un nouveau type d'entités MOTCLE. Le type d'associations N:N qui relie les deux types d'entités est ensuite transformé en un *type d'entités association* accompagné de deux types d'associations 1:N (section 13.5.5, figure 13.6), lesquels sont transformés en clés

étrangères (section 14.4.3). Chaque entité du nouveau type représente une valeur distincte de l'attribut source MotClé. Ce dernier est représenté par la table ADRESSE qui reprend toutes les **valeurs distinctes** de MotClé (schéma **b**). La contrainte référentielle forte **equ** peut être assouplie en **ref** en admettant la cardinalité [0-N] moins contraignante du rôle de MOTCLE.

Évaluation des techniques

- **Type d'entités / représentation des instances.** Le schéma résultant est **concis** (une seule table additionnelle, pas de redondance) mais d'expressivité limitée. La table MOTCLE_DE_DOC n'offre pas de valeur propre en dehors d'un complément à la table DOCUMENT. Cette technique, tout comme la seconde, ne s'applique qu'aux attributs multivalués de niveau 1.
- **Type d'entités / représentation des valeurs.** Le résultat est le **moins concis** (ajout de deux tables) mais aussi le **plus expressif** et le **plus évolutif**. Il est facile d'ajouter des propriétés au concept de *mot clé* (traduction dans d'autres langues ou référence à une ontologie par exemple) et de l'associer à d'autres types d'entités (pour constituer un thésaurus). La table MOTCLE possède une réelle valeur intrinsèque (le répertoire des mots clés), indépendante de la table DOCUMENT. On remarquera que le schéma supérieur est entièrement inclus dans le second. Les tables comporteront donc un certain degré de redondance qu'il faudra gérer. L'expressivité et l'évolutivité ont un prix !

19.3.4 Représentation d'un attribut composé multivalué

Cette configuration d'attribut est assez rare dans un schéma conceptuel normalisé construit avec méthode et discipline. Lorsqu'un tel attribut apparaît, il est en général assorti d'un identifiant propre. On le traitera d'abord comme un attribut multivalué, puis, dans son nouveau type d'entités où il est devenu monovalué, comme un attribut composé. La figure 19.9 montre comment l'attribut Détail après transfert vers le nouveau type d'entités DETAIL (représentation des instances), a été désagrégé. Une discussion du rôle des identifiants dans les attributs complexes a été proposée à la section 16.10.3. On s'y reportera utilement.

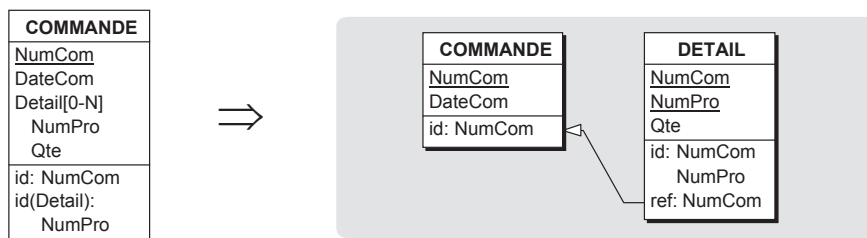


Figure 19.9 - Représentation relationnelle d'un attribut composé multivalué

19.3.5 Traduction des types et domaines de valeurs

Les types de valeurs des attributs sont convertis en fonction du type de données SQL le plus proche. Pour ce qui concerne les types définis par l'utilisateur (TDU), il faut se souvenir que seuls les types atomiques sont admis en SQL2 (section 6.6) alors que SQL3 admet des types composés et/ou multivalués (section 9.6.1). La traduction des TDU atomiques est donc immédiate.

Pour les TDU complexes (section 16.7.4) on suggère, soit de les traduire en TDU SQL3, soit de procéder à leur *matérialisation*, c'est-à-dire de remplacer l'attribut par la définition de son TDU, puis de procéder à la transformation de l'attribut complexe comme décrit en 19.3. La figure 19.10 illustre la matérialisation (partielle) des domaines matricule et adresse empruntés à la figure 16.6, suivie de leur désagrégation, selon le procédé de désagrégation (figure 19.6). Cette dernière nous suggère d'ailleurs deux autres techniques (non illustrées) basées sur la transformation de l'attribut en type d'entités.

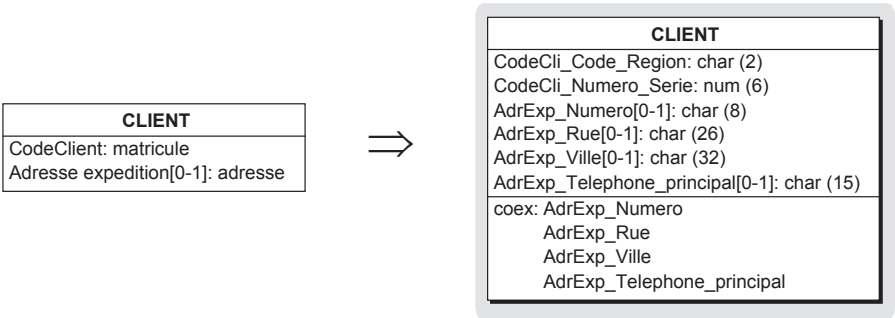


Figure 19.10 - Matérialisation d'un TDU complexe

L'attribut peut être défini sur un *domaine énuméré*, défini par l'ensemble des valeurs admissibles (section 16.11.1). Si ces valeurs sont peu nombreuses et invariables (exemple : Sexe = {'M', 'F'}) on associera à l'attribut ou à son domaine une contrainte sous la forme d'un prédicat check. Si, en revanche, cet ensemble est de grande taille et est susceptible d'évoluer, il sera préférable de le représenter par un *type d'entités domaine*, beaucoup plus aisé à gérer qu'une contrainte et par ailleurs partageable par d'autres attributs similaires. La figure 19.11 montre la représentation de l'attribut Pays, dont le domaine de valeurs comprenait 192 éléments en 2010.

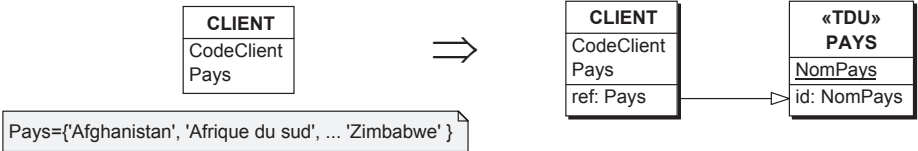


Figure 19.11 - Représentation d'un type ou domaine énuméré évolutif et de grande taille

19.3.6 Compléments

Deux compléments utiles aux techniques de base qui viennent d'être développées.

a) Expression des cardinalités spéciales

Un examen attentif de la figure 19.8 montre que l'équivalence conceptuel/logique n'est pas complètement assurée. Nous avons en effet perdu la limite de 10 de la contrainte de cardinalité [0-10] de l'attribut MotCle. Rien n'empêche en effet d'associer 11 lignes de MOTCLE_DE_DOC à une ligne de DOCUMENT. Le modèle relationnel, même enrichi, ne permet pas d'exprimer une telle contrainte. Elle sera donc exprimée comme une contrainte additionnelle au moyen d'une annotation (figure 19.12). C'est au niveau du codage que nous pourrons la traduire.

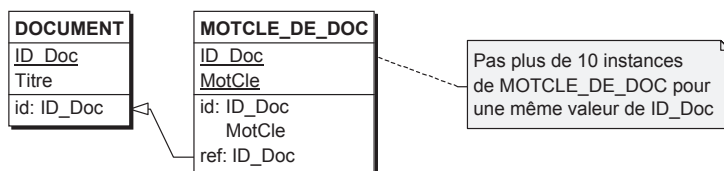


Figure 19.12 - Contrainte de cardinalité sous forme d'annotation

b) Attention : transformations déconseillées !

Les techniques de représentation que nous venons de décrire ne sont évidemment pas les seules qui soient appliquées en pratique. Nous en mentionnerons deux qui, malgré leur usage fréquent, ne sont pas à conseiller : la concaténation et l'instanciation (figure 19.13). La **concaténation** consiste à traduire un attribut composé ou multivalué par une simple colonne dont chaque valeur est obtenue par la concaténation⁵ des valeurs de l'attribut d'origine. Nous avons déjà évoqué ces deux techniques à la figure 13.15 en les qualifiant de *représentations à éviter*. Ces techniques réduisent considérablement l'expressivité et l'évolutivité⁶ des structures et compliquent l'exploitation de la base de données (écrire par exemple une requête SQL qui recherche les emprunteurs habitant à Courtrai, en évitant bien sûr ceux qui habitent "rue de Courtrai"; écrire ensuite une requête qui modifie la rue d'un emprunteur). L'**instanciation** consiste à représenter chaque instance de valeur d'un attribut multivalué par une colonne indépendante.

Le principal problème de ces deux techniques appliquées à un attribut multivalué est qu'elles introduisent des aspects indésirables délicats à gérer. Il est par exemple difficile de garantir l'unicité des mots clés pour un document. En outre, les valeurs

5. Concaténer deux chaînes de caractères consiste à en former une troisième en mettant bout à bout les deux premières. "Jeanne Marine" est la conténation de "Jeanne" et " Marine".

6. Certains spécialistes pourraient affirmer, non sans raison, que la concaténation peut favoriser dans une certaine mesure les évolutions futures. Une telle discussion dépasserait les objectifs de cet ouvrage.

sont ordonnées et il est possible que des valeurs soient manquantes (les valeurs de MotCle_1 et MotCle_3 sont présentes alors que MotCle_2 est null)⁷.

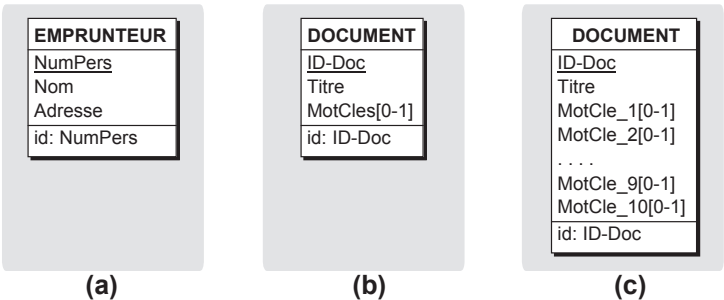


Figure 19.13 - Concaténation (a et b) et instantiation (c) : deux transformations fréquentes mais déconseillées

19.4 RETOUR AUX TYPES D'ASSOCIATIONS FONCTIONNELS

Nous avons vu que les types d'associations *fonctionnels* (1:1 ou 1:N) se traduisent par une clé étrangère (sections 14.4.1 et 14.4.2). Cependant, rien ne nous empêche de les traiter comme les types d'associations N:N, c'est-à-dire de les représenter par une *table association*. La figure 19.14 illustre cette technique.

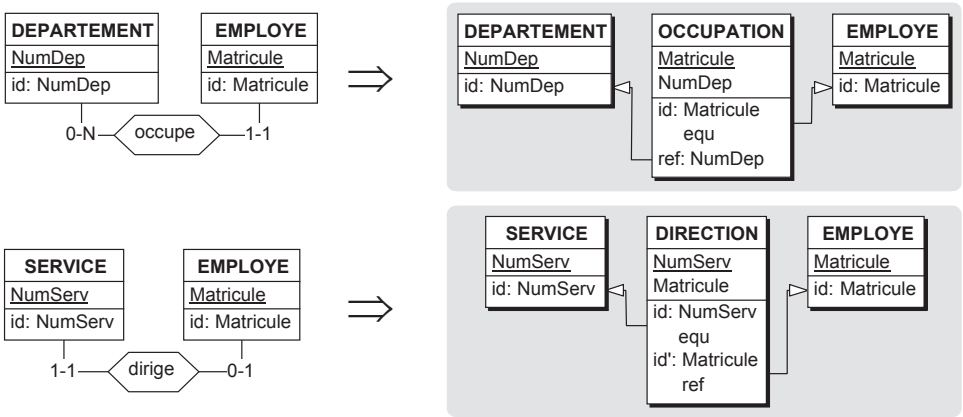


Figure 19.14 - Représentation d'un type d'associations fonctionnel favorisant l'évolution

La technique semble inutilement lourde, puisque qu'elle exigera deux jointures au lieu d'une pour associer les lignes des deux tables en relation. En revanche, elle permet une évolution ultérieure beaucoup plus aisée. Un type d'associations binaire 1:1 ou 1:N pourra évoluer vers la classe N:N, ou être doté d'attributs, sans que le

7. Dit d'une manière plus technique, ces deux transformations représentent un ensemble de valeurs par un tableau de valeurs.

schéma relationnel et les programmes d'application doivent être remaniés en profondeur. La principale modification consistera à modifier l'identifiant de la table association. On notera cependant l'apparition de contraintes *equ* dont la traduction en SQL pourra s'avérer délicate. Nous reviendrons sur ce point à la section 19.7.6.

Noms des colonnes d'une clé étrangère

Par défaut, nous avons donné aux nouvelles colonnes formant une clé étrangère le nom de leur correspondant dans l'identifiant cible (NUMCLIENT vers CLIENT dans CONTRAT). Pour améliorer la lisibilité du schéma, on envisagera de leur donner des noms plus explicites : nom de la table cible (CLIENT) ou nom du rôle que joue cette table cible (SIGNATAIRE). Lorsque la clé étrangère est formée de plusieurs colonnes, on prefixera leur nom de manière significative : CONTRAT_SIGNATAIRE, CONTRAT_NUMCTR. On identifie ainsi plus facilement ce groupe de colonnes.

19.5 REPRÉSENTATION DES TYPES D'ASSOCIATIONS COMPLEXES

On qualifiera de *complexe* un type d'associations n-aire, et/ou doté d'attributs, éventuellement soumis à des contraintes telles que des identifiants, ainsi que les types d'associations dotés d'un rôle polymorphe.

Les types d'associations complexes vont réclamer la définition d'une *table association* dont chaque ligne représentera une association. Cette technique a déjà été utilisée pour les types d'associations N:N (section 14.4.3), de sorte que sa généralisation aux structures complexes ne doit pas poser de problèmes majeurs.

19.5.1 Types d'associations n-aires, avec attributs et/ou contraintes

Le type d'associations emprunte est *complexe* selon la définition précédente. Pour limiter la discussion à l'essentiel, il est repris à la figure 19.15, EXEMPLAIRE étant doté d'un identifiant simplifié. Dans une première étape, le type d'associations est transformé en un type d'entités (EMPRUNT) et d'autant de types d'associations fonctionnels (dont le nom importe peu) qu'il possède de rôles. Les attributs sources et les identifiants éventuels sont transférés vers le nouveau type d'entités. Dans une seconde étape, chacun de ces types d'associations est transformé en clé étrangère. Les contraintes de cardinalités spéciales se traitent comme décrit précédemment.

19.5.2 Représentation d'un type d'associations à rôle polymorphe

Les règles habituelles de traduction en clé étrangère ne sont plus valables pour les rôles polymorphes. La transformation illustrée à la figure 19.16 permet de traiter le cas le plus courant : type d'associations binaire et rôle opposé de cardinalité [0-1] ou [1-1]. Le type d'associations source est dupliqué pour chaque type d'entités du rôle polymorphe avant d'être transformé en clé étrangère facultative. Une contrainte *excl* (pour un rôle [0-1]) ou *exact-1* (pour un rôle [1-1]) est imposée au type d'entité opposé.

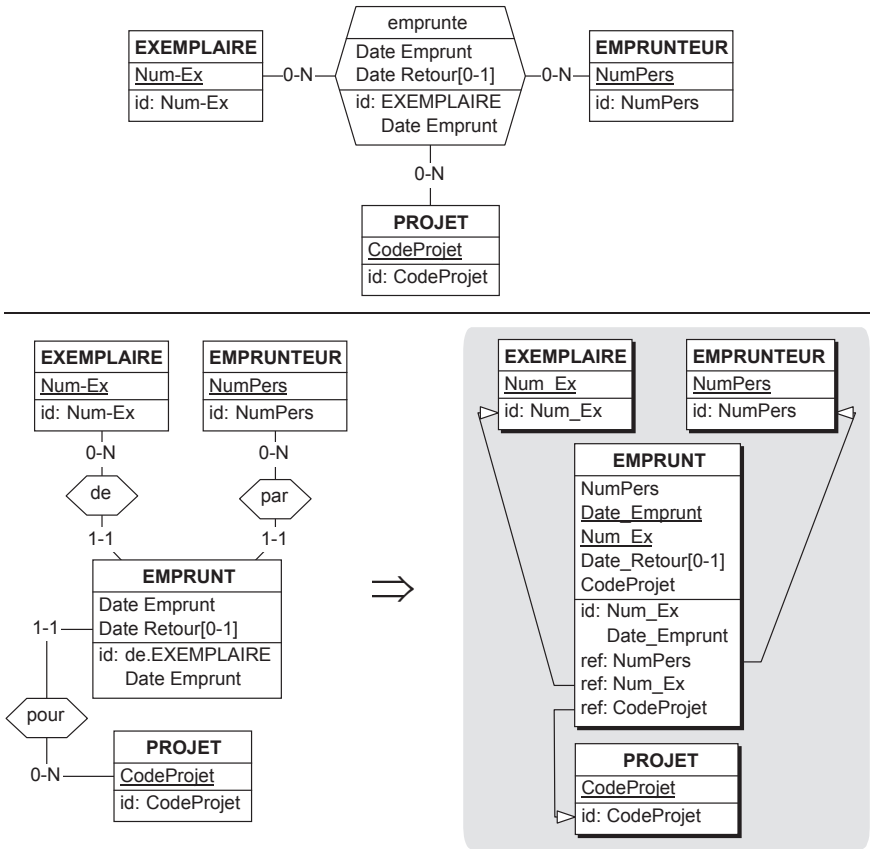


Figure 19.15 - Représentation d'un type d'associations complexe

Lorsque le rôle polymorphique intervient dans un identifiant, celui-ci sera dupliqué pour chaque type d'associations sous la forme d'un identifiant secondaire (puisque'un de ses composants est facultatif).

La représentation d'un type d'associations N:N, n-aire ou possédant **plusieurs rôles polymorphiques** peut s'approcher de plusieurs manières. La première applique le principe de multiplication qu'on vient de traiter à chacun des rôles polymorphiques. La seconde passe par la transformation préalable du type d'associations en type d'entités. Les rôles polymorphiques sont alors transférés aux types d'associations binaires, qui se traitent ensuite comme décrit précédemment. Une troisième approche consiste à définir pour chaque groupe de types d'entités jouant un rôle polymorphique un surtype d'entités, puis à faire jouer ce rôle par ce surtype. L'exercice A16.9 de l'annexe A16 développe et compare ces trois solutions.

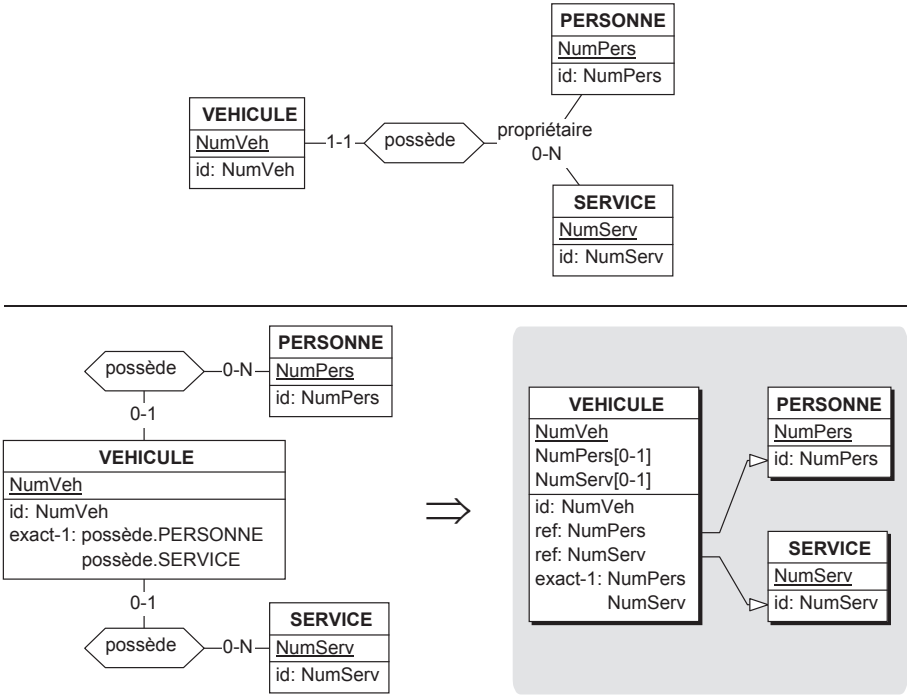


Figure 19.16 - Représentation d'un type d'associations à rôle polymorphe

19.5.3 Clé étrangère et identifiant absent

La représentation d'un type d'associations fonctionnel par une clé étrangère suppose que le type d'entités qui se traduira par la table référencée soit doté d'un identifiant, primaire de préférence, constitué, *in fine*, d'un ensemble d'attributs.

Il peut arriver que ce ne soit pas le cas, soit que ce type d'entités ne dispose pas d'identifiant, soit qu'il n'est pas possible, sans intervention extérieure, de réduire ses composants rôles en attributs (examiner attentivement les identifiants cycliques de la section A16.8 de l'annexe A16 par exemple). Dans un tel cas, on ajoutera à la table référencée une colonne technique qui en constituera l'identifiant primaire.

La figure 19.17 montre le traitement du type d'associations écrit, dont la traduction nécessite l'ajout à AUTEUR d'un identifiant technique *ID_Auteur*.

19.6 REPRÉSENTATION DES RELATIONS IS-A

Nous étudions dans cette section la représentation des relations *is-a* entre un surtype d'entités et ses sous-types d'entités dans le modèle SQL2 qui n'admet pas de représentation explicite de ces relations. Les extensions relationnelles objet de SQL3 seront étudiées à la section 19.11.3.

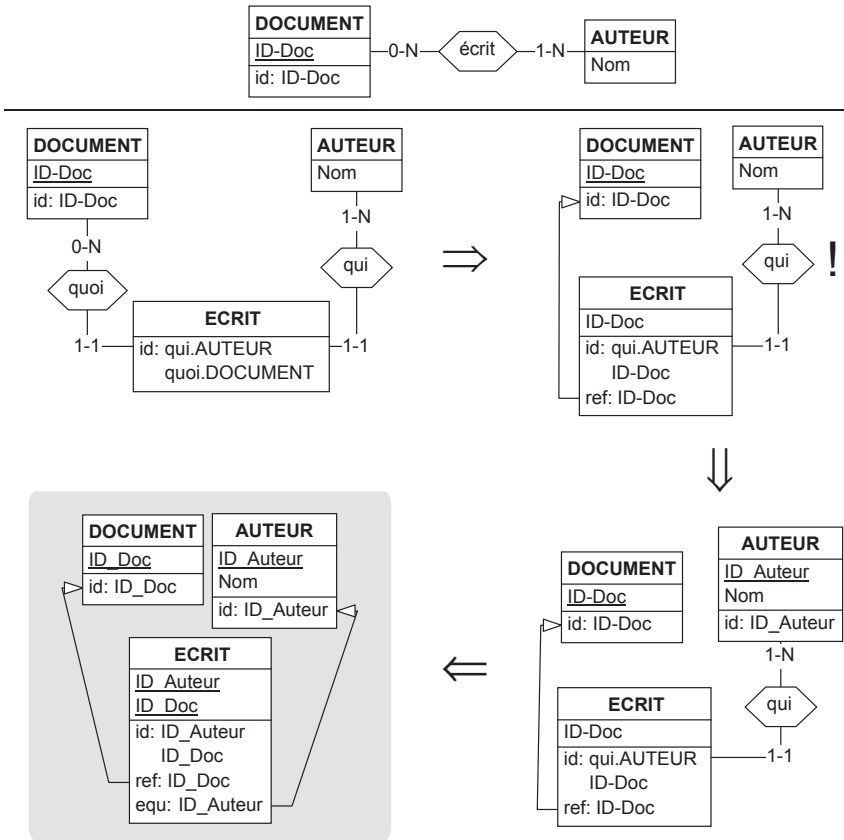


Figure 19.17 - Représentation d'un type d'associations nécessitant l'ajout d'un identifiant technique

Nous travaillerons principalement sur la structure DOCUMENT/RAPPORT/OUVRAGE de la figure 19.18. Dans un premier temps, nous présenterons succinctement trois familles de techniques, puis nous les détaillerons individuellement.

19.6.1 Les trois techniques de base

Les trois familles de représentation de relations *is-a* se distinguent selon les types d'entités sources (surtypes, sous-types, surtypes et sous-types) qui sont explicitement représentés [Hainaut, 1996].

- La première, dite par **matérialisation**, conserve tous les types d'entités source (DOCUMENT, RAPPORT, OUVRAGE), qu'ils soient surtype ou sous-types et *matérialise* les relations *is-a* sous forme de types d'associations 1:1.
- La deuxième technique ne représente que les sous-types (RAPPORT, OUVRAGE). Les composants du surtype sont au préalable diffusés vers chaque sous-type selon le principe de l'**héritage descendant** (section 16.9.3) ; s'il existe des entités du

- surtype qui n'appartiennent à aucun sous-type, on les rassemble dans un sous-type spécifique, de même format que le surtype (DOC_SEUL).
- La troisième technique ne représente que le surtype (DOCUMENT). Celui-ci reprend les composants (attributs, rôles, contraintes) de chacun de ses sous-types, selon le principe de l'*héritage ascendant* (section 16.9.3).

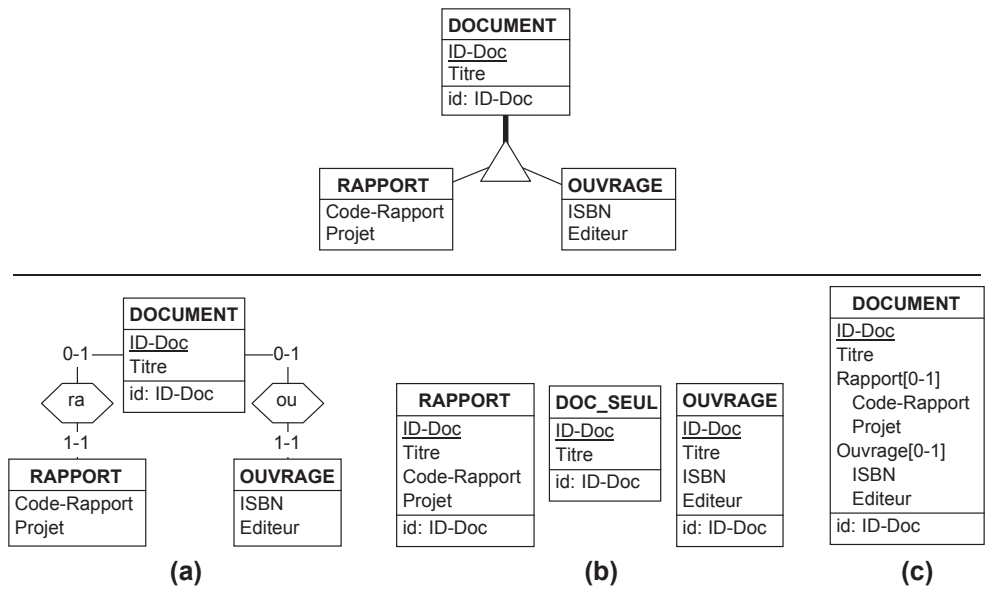


Figure 19.18 - Les trois familles de transformations : (a) matérialisation, (b) héritage descendant et (c) héritage ascendant.

19.6.2 Représentation de relations *is-a* par matérialisation

Cette technique est certainement celle que le concepteur adoptera naturellement. Elle préserve en effet les types d'entités du schéma conceptuel d'origine. Elle traite en outre chaque type d'entités de manière égale, sans assigner au surtype ou aux sous-types un statut privilégié.

a) Représentation des relations *is-a*

La figure 19.19 montre, sous chaque configuration de contraintes, le résultat de la représentation par matérialisation. Les contraintes de sous-types se traduisent par des contraintes d'existence appliquées au surtype.

b) Représentation des types d'associations 1:1

Chaque type d'associations 1:1 est ensuite transformé en une clé étrangère associée à chaque sous-type et référençant le surtype.

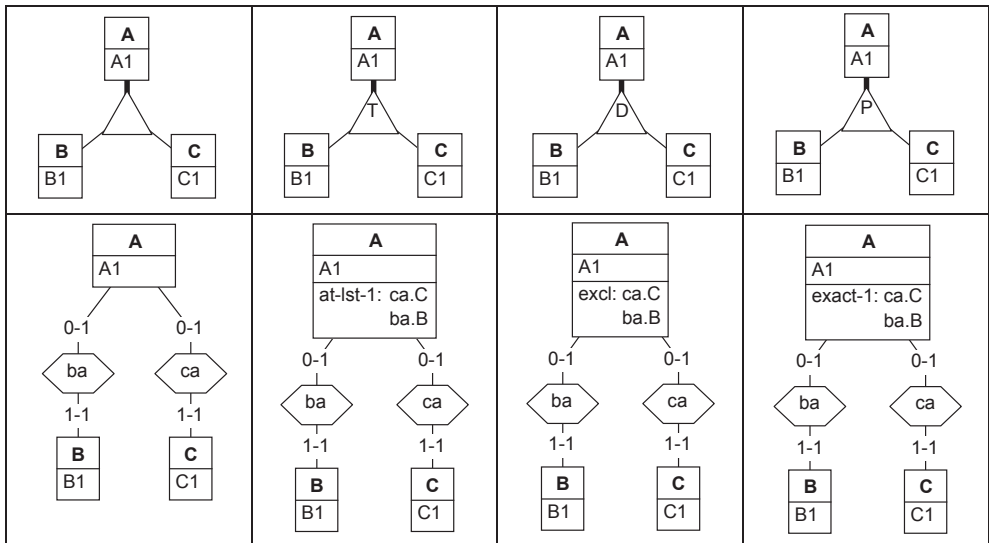


Figure 19.19 - Représentation par matérialisation - Traduction des contraintes de sous-types

Si cette traduction convient parfaitement à la première configuration (*libre* = non-disjonction et non-totalité), elle présente en revanche pour les trois autres une difficulté que nous allons analyser. Celle-ci concerne l'adaptation de chacune des contraintes d'existence aux structures relationnelles. On observe en effet que cette contrainte, qui est attachée à la table du surtype, concerne la présence de lignes dans les tables des sous-types ! Ces contraintes d'existence ne sont plus valides, ce qui nous oblige à traiter le problème d'une autre façon. La référence [Hainaut, 1996] propose trois représentations, dont il apparaît qu'aucune n'est totalement satisfaisante, puisqu'elles s'accompagnent toutes d'un jeu de contraintes additionnelles dont la validation n'est pas évidente. Nous décrirons l'une d'elles, qui exige que deux conditions soient satisfaites.

1. Le surtype possède un **identifiant primaire**, propre ou hérité. Si tel n'est pas le cas, on lui assigne un identifiant technique.
2. Le surtype possède un **indicateur de sous-type** (section 16.9.8). Si celui-ci est absent, on l'ajoute. Nous choisissons d'ajouter pour chaque sous-type un indicateur facultatif et dont le nom est celui du sous-type. Son type est quelconque (par exemple 1 caractère). Pour chaque entité du surtype, l'indicateur possédant une valeur (non null) indique que l'entité appartient aussi au sous-type correspondant.

La figure 19.20 traite le cas d'une disjonction simple (D), les autres cas (T et P) se résolvant de manière similaire. Tel quel, le schéma est incomplet. En effet, si la contrainte D est bien traduite par la contrainte d'exclusion des indicateurs, il n'existe aucun lien entre l'état de ces indicateurs et l'existence ou l'absence d'une ligne correspondante dans la table du sous-type. Nous devons donc préciser ce lien par une contrainte d'intégrité qui spécifie que, *pour toute ligne D de DOCUMENT*,

D.RAPPORT est not null ssi il existe une ligne *R* de *RAPPORT* telle que *R.ID_Doc* = *D.ID_Doc*. Cette contrainte, qui prend la forme d'une *équivalence* logique entre deux propositions (section A24.4.3, annexe A24), est traduite sous la forme d'une annotation en pseudo-SQL. Chaque sous-type fait l'objet d'une contrainte similaire. Pour traiter une hiérarchie de plus de deux niveaux, on commence par traiter le surtype de plus haut niveau, puis on procède progressivement vers le bas.

L'accès aux composants d'une entité d'un sous-type exigera une jointure de la table du surtype et de toutes les tables des sous-types auxquels l'entité appartient.

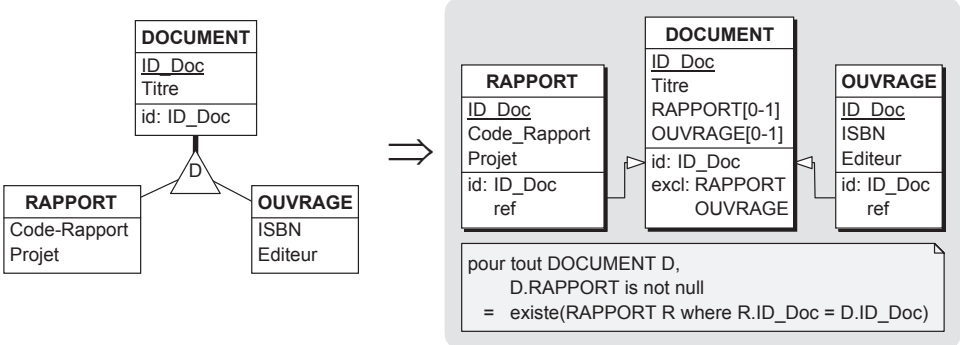


Figure 19.20 - Traduction de relations *is-a* par matérialisation (la contrainte sur *OUVRAGE* est manquante)

c) Représentation des types d'associations 1:1 : une variante

Dans le chapitre 3, nous avons rencontré une forme assez curieuse de clé étrangère, la *clé étrangère non minimale* (section 3.8.5). Celle-ci va servir à définir une autre représentation des relations *is-a* par matérialisation. La figure 19.21 est une variante de la figure 19.20. Les indicateurs de sous-type, nommés *estRAPPORT* et *estOUVRAGE*, sont cette fois obligatoires et peuvent prendre deux valeurs, 1 et 0, selon que la ligne de *DOCUMENT* représente ou non une entité du sous-type désigné.

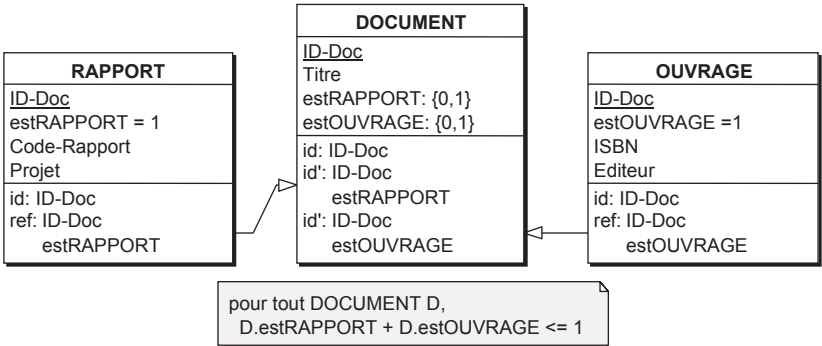


Figure 19.21 - Représentation des relations *is-a* par des *clés étrangères non minimales*. La contrainte additionnelle correspond à une *disjonction*.

Nous associons ensuite à la table DOCUMENT autant d'identifiants secondaires qu'il existe de sous-types possibles. Chaque identifiant reprend l'identifiant primaire ID-Doc ainsi que l'indicateur de sous-type :

(id': ID-Doc, estRAPPORT) et (id': IID-Doc, estOUVRAGE)

Ces identifiants sont évidemment non minimaux et ne joueront qu'un rôle technique. Dans chaque table sous-type, nous ajoutons un indicateur de sous-type qui possède toujours la valeur 1. Enfin, nous déclarons dans chaque table sous-type une clé étrangère non minimale, composée de l'identifiant primaire ID-Doc et l'indicateur de sous-type local, ciblant l'identifiant secondaire correspondant dans DOCUMENT :

(ref: ID-Doc, estRAPPORT) et (ref: IID-Doc, estOUVRAGE)

Il reste à associer à la table DOCUMENT une contrainte qui traduit la contrainte des sous-types. On peut imaginer plusieurs formes possibles. On choisit celle-ci, dont la longueur est proportionnelle au nombre de sous-types, et qui est directement exprimable sous la forme d'un prédicat *check* :

- *disjonction* : un seul indicateur peut être à 1 ($\text{estRAPPORT} + \text{estOUVRAGE} \leq 1$)
- *totalité* : au moins un indicateur doit être à 1 ($\text{estRAPPORT} + \text{estOUVRAGE} > 0$)
- *partition* : un seul indicateur doit être à 1 ($\text{estRAPPORT} + \text{estOUVRAGE} = 1$)

On observe qu'il n'est pas possible de créer une ligne sous-type qui ne respecterait pas les contraintes d'intégrité. Ce schéma est entièrement exprimable en SQL2 natif, y compris les contraintes de sous-type.

Il n'est cependant pas encore complet puisqu'il autorise la suppression d'une ligne sous-type sans mise à jour de l'indicateur *estRAPPORT* dans la ligne surtype. Il reste également une difficulté non encore résolue, liée à la contrainte de *totalité* : on ne peut pas créer une ligne surtype sans qu'il n'existe en même temps une ligne sous-type, et inversement. Ces deux points seront examinés au chapitre 21.

d) Les autres composants

Qu'en est-il des contraintes et rôles des sous-types ? Les contraintes affectant le surtype ou un sous-type sont intégralement transmises aux tables correspondantes. Par exemple, l'identifiant Code-Rapport de RAPPORT se traduira immédiatement par un identifiant secondaire de la table RAPPORT. On observe également qu'un rôle joué par le surtype ou un sous-type se transforme selon les techniques habituelles. Tel serait le cas des types d'associations *réserve*, *de* et *écrit* impliquant les types d'entités DOCUMENT et OUVRAGE. Les contraintes impliquant des composants hérités (section 16.9.6), en particulier des attributs, exigent un traitement un peu particulier. Portant sur des attributs appartenant à des tables différentes, la contrainte ne pourra s'exprimer que si on introduit des copies des attributs étrangers dans la table à laquelle la contrainte est associée. Ces copies feront l'objet de contraintes de redondance (section 16.12.5).

19.6.3 Représentation de relations *is-a* par héritage descendant

On ne représente ici que les sous-types, les composants du surtype étant copiés dans chaque sous-type. Cette technique sera choisie lorsque l'intérêt des utilisateurs et des développeurs se porte presque exclusivement sur les sous-types. Il apparaît que sa gestion correcte peut être relativement complexe, en particulier pour des *sous-types non disjoints*, cas que nous écartons dans notre étude.

a) Représentation des relations *is-a*

Cette représentation s'applique de préférence à une structure de partition (P). Nous complétons donc le schéma source dans ce sens, en ajoutant un sous-type DOC_SEUL, sans composants, représentant les documents qui ne sont ni des rapports ni des ouvrages (figure 19.22).

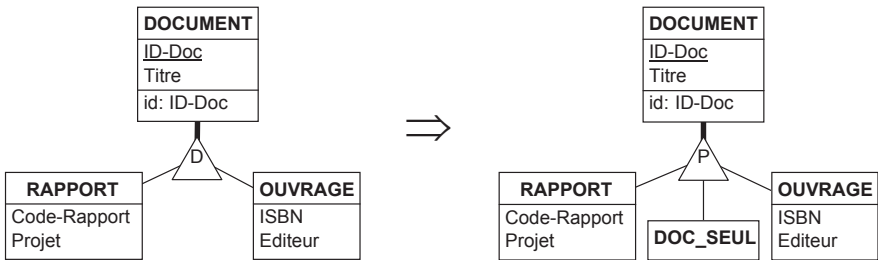


Figure 19.22 - Les sous-types doivent former une partition

On peut dès lors procéder à la copie des composants du surtype puis éliminer celui-ci pour obtenir le schéma (a) de la figure 19.23. Nous avons cependant introduit un problème dans ce schéma : alors que le schéma source précise qu'il ne peut exister deux documents de même valeur d'ID-Doc, le nouveau schéma tolérerait l'existence d'un rapport, d'un ouvrage et d'un document simple de même valeur d'ID-Doc. Pour empêcher que de telles situations se produisent, nous devrions préciser que la colonne ID_DOC est un *identifiant global* dans l'union des tables des sous-types (section 16.10.1), ce qu'on pourrait exprimer par :

id(RAPPORT, OUVRAGE, DOC_SEUL): ID_Doc

Cette contrainte n'ayant pas d'équivalent en SQL, nous indiquons par une annotation dans le schéma (a) que les trois ensembles de valeurs d'ID_Doc sont disjoints.

Malheureusement, la validation de cette contrainte n'est pas des plus simples. Lors de l'insertion d'une ligne de RAPPORT, il faut au préalable vérifier qu'il n'existe ni dans la table OUVRAGE ni dans la table DOC_SEUL de ligne de même valeur de ID_Doc. Pour un ensemble de *n* sous-types, on doit ajouter *n-1* contraintes par sous-type, soit au total *n x (n-1)* contraintes. Si chacune est assez facile à transcrire en SQL sous la forme d'un déclencheur, leur nombre et leur coût devient rapidement prohibitif dès que le nombre de sous-types augmente. C'est cette observation qui incite certains concepteurs à suggérer une structure hybride dans laquelle on introduit une table DOCUMENT, réduite à l'identifiant ID_Doc et aux indicateurs de sous-types (figure 19.23, schéma b). C'est dans le contexte de cette table que la

contrainte de partition est gérée de la même manière que pour la technique de matérialisation. Ici encore, l'usage des *clés étrangères non minimales* pourrait considérablement améliorer cette technique de représentation.

Les composants d'un sous-type sont représentés dans une seule table puisque nous avons fait l'hypothèse de la contrainte de disjonction dans le schéma source. Si cette contrainte n'existait pas, une entité du surtype pourrait être représentée dans plusieurs tables des sous-types. La gestion de la cohérence de ces copies multiples serait particulièrement complexe. D'où notre restriction.

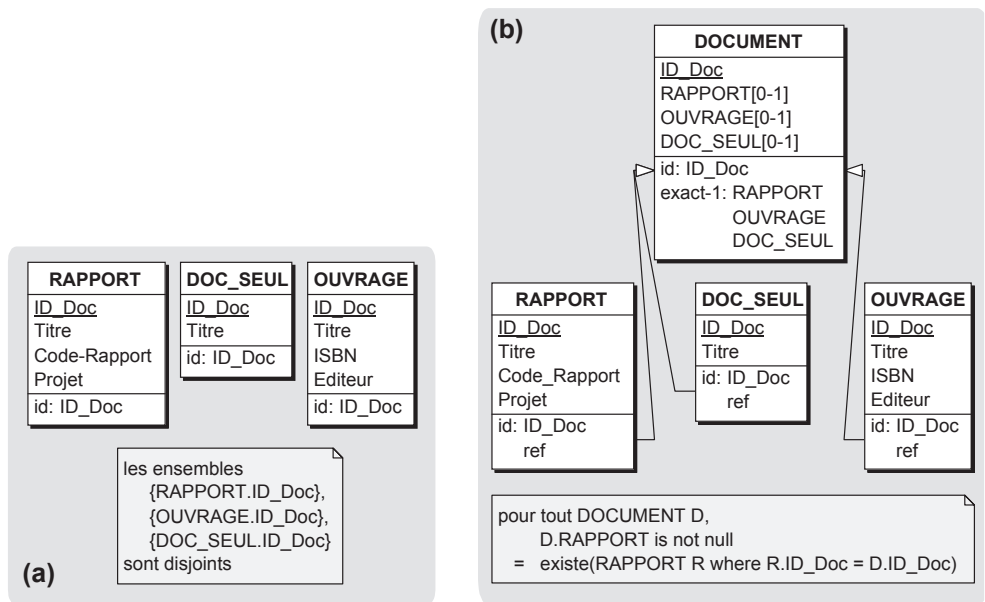


Figure 19.23 - Deux traductions de relations *is-a* par héritage descendant. Le schéma de droite permet de gérer facilement un grand nombre de sous-types. Les contraintes sur DOC_SEUL et OUVRAGE ne sont pas montrées.

b) Les autres composants

Les contraintes et rôles relatifs à un sous-type se traduisent sans difficulté par les techniques décrites dans ce chapitre. Tout rôle joué par le surtype est transformé en rôle polymorphique joué par l'ensemble des sous-types. Il est alors traité comme proposé à la section 19.5.2. Les identifiants secondaires du surtype se traitent comme l'identifiant primaire, ce qui complique le schéma logique. Les autres contraintes imposées au surtype sont distribuées vers chaque sous-type.

19.6.4 Représentation de relations *is-a* par héritage ascendant

Cette technique ne conserve que le surtype, qui recueille tous les composants de tous ses sous-types. Cette technique produit un schéma réduit en nombre de tables, mais les tables de surtypes sont particulièrement complexes. Elle sera utile pour autant que les sous-types soient peu impliqués dans des rôles et des contraintes complexes.

Elle offre l'avantage de présenter à l'utilisateur et au programmeur, en une seule opération, toutes les données relatives à une entité du surtype quels que soient les sous-types auxquels cette entité appartient. Cette technique est la seule qui s'accommode naturellement de la *non-disjonction* des sous-types.

a) Représentation des relations is-a

Les attributs de chaque sous-type sont associés au surtype sous forme d'un attribut composé facultatif dont le nom est celui du sous-type. Une contrainte d'existence (excl, at-least-1, exact-1) traduit la contrainte de sous-type (figure 19.24, schéma central).

b) Représentation des attribut composés

SQL3 admet les attributs composés (array). En SQL2, ces attributs sont désagrégés. Une contrainte de coexistence rassemble les composants de chacun d'eux. Grâce à la contrainte de coexistence, la contrainte d'exclusion ne reprendra qu'un seul composant par attribut composé (figure 19.24, schéma de droite).

c) Les autres composants

Si un sous-type possède des attributs facultatifs, la contrainte de coexistence sur les attributs hérités n'est plus correcte. Elle doit être remplacée par un jeu de contraintes plus complexes selon le canevas étudié à la section 19.3.2.

Les identifiants des sous-types sont réaffectés au surtype, dans lequel ils sont secondaire. Lorsqu'un sous-type joue un rôle, ce rôle est transféré (facultatif) vers le surtype, à partir duquel il sera transformé selon les techniques habituelles. Cette représentation peut cependant entraîner la formulation de contraintes complexes.

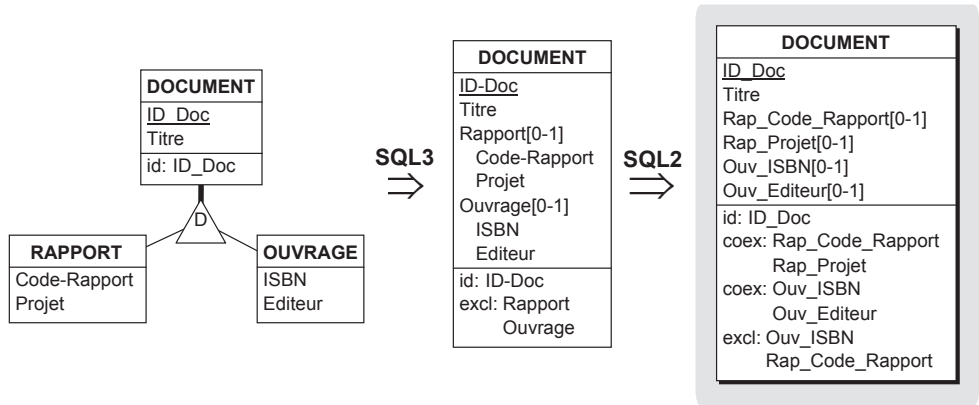


Figure 19.24 - Traduction de relations is-a par héritage ascendant.

19.6.5 Évaluation comparative

Le développement des trois techniques suggère assez clairement les domaines dans lesquels chaque technique est la plus appropriée.

- **Matérialisation.** Le résultat est homogène, chaque type d'entités source gardant son identité. Cette homogénéité rend l'évolution du schéma particulièrement aisée. Ajouter un sous-type, un surtype, un attribut, un rôle, une contrainte ou modifier des objets du schéma conceptuel se propage avec un impact minimum sur le schéma logique. La traduction des attributs, des contraintes et des rôles s'effectue sans difficulté via les techniques habituelles. Le schéma logique est topologiquement proche du schéma conceptuel et donc facile à comprendre (un type d'entités = une table). Cette technique convient aux hiérarchies multiples. Cependant, les contraintes de sous-types sont complexes à gérer. En outre l'expression de contraintes impliquant des composants hérités entraîne des redondances. La reconstitution de l'information relative à une entité réclame une jointure des tables contenant les fragments de cette entité.
- **Héritage descendant.** Cette technique est envisageable lorsque le surtype est d'une importance sémantique mineure, soumis à peu de contraintes et jouant peu de rôles. Le schéma logique favorise les sous-types, pour lesquels il présente toute l'information en une seule table. La gestion des contraintes issues du surtype peut s'avérer complexe, conduisant à une structure hybride peu lisible. Cette technique exige en pratique une contrainte de disjonction. Étant donné l'intrication des représentations des sous-types et du surtype, le schéma logique supporte difficilement certaines modification d'évolution conceptuelle. On appliquera l'héritage descendant principalement aux types d'entités racines d'une hiérarchie (réduction par le haut). L'application de cette technique aux hiérarchies multiples est théoriquement possible mais entraîne un haut degré de complexité.
- **Héritage ascendant.** Technique envisageable lorsque le surtype supporte l'essentiel de la sémantique et que les sous-types apportent un complément d'information peu complexe : peu de contraintes et peu de rôles. L'utilisateur s'intéresse prioritairement au surtype, dont les données lui sont livrées en une seule table, quels que soient les sous-types des entités. Cette technique supporte toutes les contraintes de sous-types, dont la gestion peut en général être traduite sous la forme de contraintes SQL locales (via des prédicats *check*). La traduction des contraintes et rôles des sous-types peut conduire à des contraintes complexes sur la table du surtype. Cette technique convient exclusivement aux hiérarchies simples. On appliquera l'héritage ascendant principalement aux types d'entités feuilles d'une hiérarchie (réduction par le bas).

Dans la suite de ce chapitre et dans les chapitres suivants nous donnerons la préférence à la technique de *matérialisation*.

19.6.6 Hiérarchies multiples et clés étrangères liées

La technique choisie pour la représentation des relations *is-a* conduit à introduire dans la table d'un type d'entités autant de clés étrangères que ce dernier a de surtypes. Ces clés étrangères ne sont cependant pas indépendantes. Étudions pour le démontrer la représentation relationnelle du schéma de gauche de la figure 19.25.

La représentation initiale des relations *is-a* en types d'associations (schéma central) s'accompagne d'une contrainte d'intégrité additionnelle qui traduit le fait que le *grand-parent* d'une entité D via B et son *grand-parent* via C sont une seule et même entité A. C'est ce qu'exprime la contrainte additionnelle " $(db*ba)[D,A] = (dc*ca)[D,A]$ " formulée de manière algébrique.

La transformation aveugle des types d'associations db et dc devrait introduire dans la table D finale deux clés étrangères A1 et A1' respectivement vers les tables B et C. La contrainte additionnelle du schéma central se traduirait en une contrainte d'égalité des valeurs de A1 et A1', exprimant ainsi que l'une de ces colonnes est redondante. On supprime donc la colonne A1' et on définit deux clés étrangères sur A1, l'une vers la table B et l'autre vers la table C (schéma de droite). A titre d'exercice, on suggère au lecteur d'appliquer ces principes au schéma de la figure 16.19.

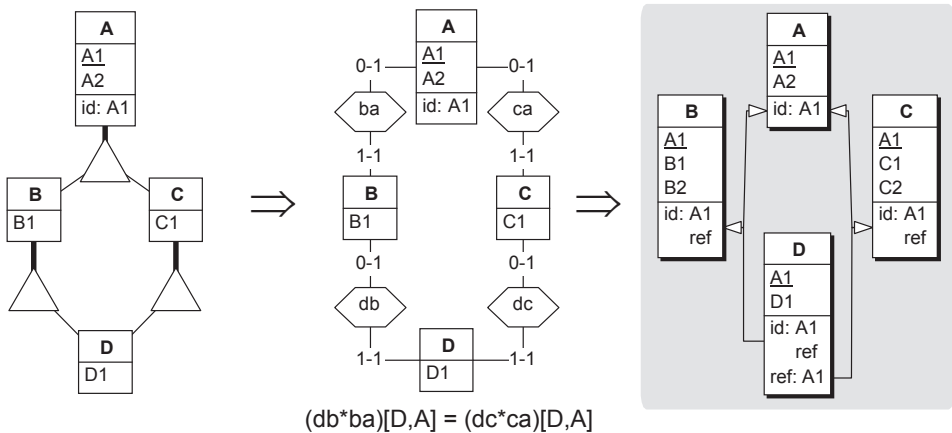


Figure 19.25 - Clés étrangères liées en cas d'héritage multiple

Ce problème peut s'avérer plus complexe lorsque (1) A ne possède qu'un identifiant secondaire et B et/ou C possède(nt) un identifiant primaire, (2) A ne possède pas d'identifiant. On suggère au lecteur de proposer des règles de traduction tenant compte de ces cas particuliers (lire par exemple [Da Silva, 2000]).

19.7 COMPLÉMENTS

Nous décrivons dans cette section quelques constructions moins complexes ou moins critiques qu'il peut être nécessaire de transformer. Nous dirons également quelques mots sur les rôles de cardinalité [1-N].

19.7.1 Les identifiants facultatifs

Un identifiant (secondaire) facultatif est constitué d'attributs facultatifs. S'il comprend plus d'un attribut, ceux-ci doivent être soumis à une contrainte de coexistence. En effet, si tel n'est pas le cas, la signification de l'identifiant est particulière-

ment complexe. Ce qui nous a conduit à le transformer lorsqu'il apparaît dans un schéma conceptuel (figure 18.40).

Les identifiants facultatifs, bien qu'autorisés par SQL, sont interdits par certains SGBD (DB2, InterBase, FireBird) tandis que d'autres ne les gèrent pas de manière correcte, considérant erronément que null = null (SQL Server, Sybase). Il est donc légitime de chercher à les transformer de manière à les rendre obligatoires. La technique la plus appropriée consiste à transformer les attributs de l'identifiant en un type d'entités accompagné d'un type d'associations 1:1. Ce dernier est ensuite transformé en clé étrangère (identifiante bien sûr) vers le type d'entités source. La figure 19.26 illustre cette représentation.

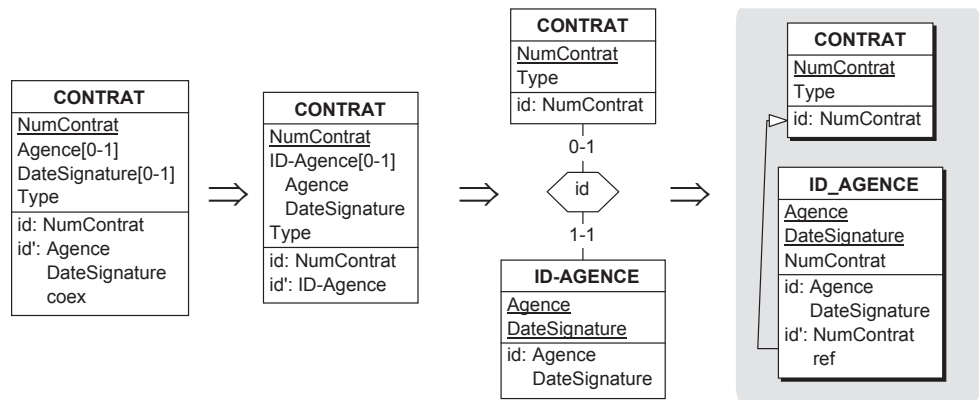


Figure 19.26 - Etapes successives de la représentation d'un identifiant facultatif

19.7.2 Les identifiants primaires complexes

A priori, le nombre de composants d'un identifiant d'un schéma conceptuel est indifférent. Un identifiant primaire constitué d'un trop grand nombre de colonnes risque de poser des problèmes de performances et de lisibilité, notamment lorsqu'il existe des clés étrangères référençant cet identifiant. La technique la plus simple consiste à lui substituer un nouvel identifiant primaire constitué d'un attribut technique de faible longueur. Le premier identifiant devient secondaire (figure 19.27).

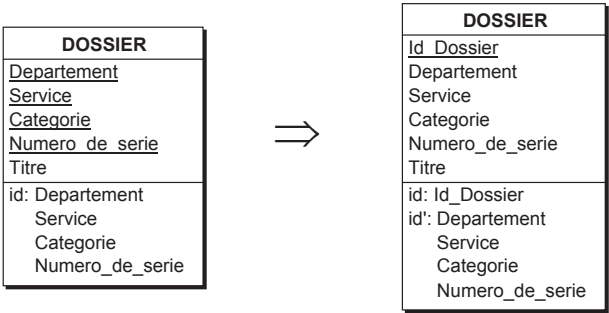


Figure 19.27 - Traitement d'un identifiant primaire trop complexe

19.7.3 Traduction des contraintes conceptuelles

Les cardinalités générales des rôles et des attributs [1-1], [0-1] et [0-N] ont été converties correctement dans les transformations examinées jusqu'ici. Il en est de même des identifiants des types d'entités, des types d'associations et d'attributs. Les contraintes de base étudiées à la section 16.11 (de domaine, entre valeurs, d'existence)⁸ sont, soit préservées dans le schéma relationnel, soit convertibles directement en structures relationnelles (coex, if).

Pour ce qui concerne les contraintes additionnelles (section 16.12), définies sous une forme plus ou moins formelle, par exemple via une annotation, on veillera à les convertir de manière qu'elle s'expriment sur les structures relationnelles. La figure 19.12, relative à la traduction d'une contrainte de cardinalité, en est un exemple.

19.7.4 Noms des objets

L'analyste jouit généralement d'une grande liberté dans le choix des noms qu'il attribue aux objets d'un schéma conceptuel. La syntaxe des noms en SQL est en revanche plus restrictive (relire la section 6.4) : les espaces, les tirets, les signes de ponctuation, les accents et d'autres symboles sont interdits. On évitera aussi les noms réservés, qui ne peuvent servir de noms d'objets dans un schéma relationnel. Même si cette contrainte affecte plus les utilisateurs anglophones que les francophones, des termes usuels tels que *table*, *privilege*, *unique*, *distinct* ou *index* sont proscrits. Signalons enfin que SQL n'est pas sensible à la casse (majuscule ou minuscule) des caractères qui forment les noms d'objets d'un schéma. Il est fréquent que le programmeur SQL assigne aux tables et colonnes des noms en majuscules (figure 19.28).

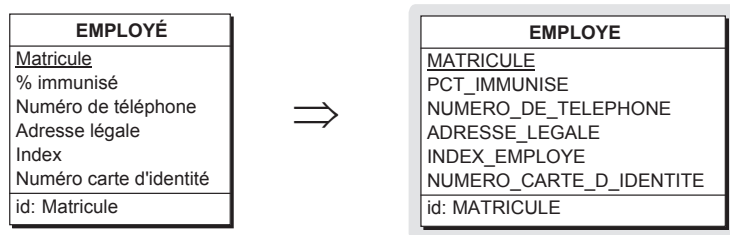


Figure 19.28 - Conversion des noms selon la syntaxe SQL

19.7.5 Delete et update modes des clés étrangères

Le schéma conceptuel décrit *ce qui est*, d'où le terme de *modèle statique* qui le désigne, notamment en UML. Il ne précise pas ce qui se passe lorsque les faits qu'ils décrit apparaissent, disparaissent ou évoluent. Si la question de la dynamique de l'information a pu être ignorée au niveau conceptuel, nous devons pourtant l'aborder lors de la production du code SQL. En particulier (et nous en resterons là),

8. Les dépendances fonctionnelles anormales ont été éliminées lors de la normalisation.

nous devons préciser les modalités de propagation des modifications des données via les clés étrangères, et donc définir les clauses *on delete* et *on update* de chacune.

Conformément aux options habituelles des SGBD, le mode *no action* sera choisi par défaut. Les autres modes seront indiqués par des stéréotypes associés aux clés étrangères. Le schéma 19.29 montre que les *delete* et *update modes* sont *no action*, sauf le *delete mode* de la clé étrangère *DETAIL.NUMCOM* qui est *cascade*⁹.

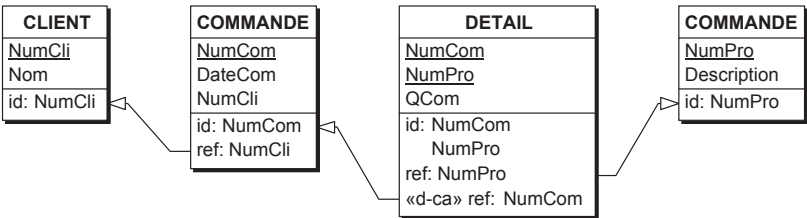


Figure 19.29 - *Delete mode cascade* pour la clé étrangère *DETAIL.NumCom*

19.7.6 Au sujet des rôles de cardinalité [1-N]

Dans la version de base du modèle Entité-association, nous avons explicitement interdit les rôles de cardinalité [1-N] au motif que leur traduction dans un schéma de tables posait des problèmes qui dépassaient les objectifs du chapitre. Or, il apparaît qu'une telle cardinalité est parfois utile, voire nécessaire. Nous l'avons d'ailleurs rencontrée à plusieurs reprises dans les pages qui précèdent. Considérons le schéma de la figure 19.30 (gauche), dans lequel on indique, via la cardinalité [1-N], le fait que *tout accident implique au moins un véhicule*.

Le type d'associations *implique* se traduit par la table *IMPLICATION* incluant deux clés étrangères dont l'une, qui référence *ACCIDENT*, est **totale** (notée *equ*).

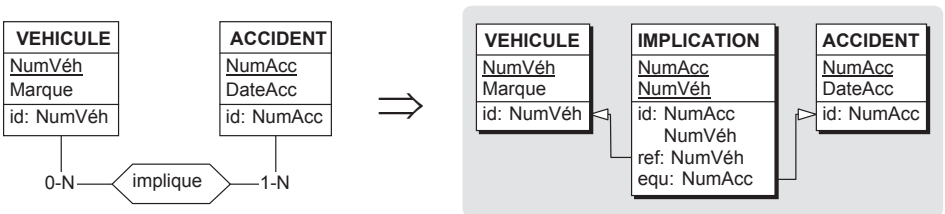


Figure 19.30 - Une cardinalité [1-N] se traduit par une clé étrangère *equ* exprimant l'égalité de deux ensembles de valeurs plutôt que leur inclusion

Une clé étrangère du type *equ* pose de sérieux problèmes qui expliquent pourquoi nous avons écarté les cardinalités [1-N] dans un premier temps.

Le premier problème est qu'il est impossible de traduire complètement ce type de clé étrangère en SQL, même par un jeu de prédicats (*check*) ou de déclencheurs (*trigger*). Le deuxième problème concerne la dynamique d'enregistrement d'un acci-

9. À ce stade, nous ne faisons pas d'hypothèse sur les modes effectivement pris en charge par le SGBD. Un mode absent pourra être émulé par des déclencheurs.

dent. 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 devrait aussi vérifier qu'il existe déjà au moins une ligne d'IMPLICATION de même valeur de NumAcc, ce qui bien sûr est contradictoire. On pourrait donc conclure à l'impossibilité intrinsèque d'insérer des lignes d'ACCIDENT, de sorte que les tables ACCIDENT et IMPLICATION devraient donc rester irrémédiablement vides ! On serait tenté de les déclarer *non satisfiables*, au sens de la section 18.6.2. La solution serait, on s'en doute un peu, de créer **simultanément** une ligne d'ACCIDENT et (au moins) une ligne d'IMPLICATION, soit sous la forme d'une transaction, d'une procédure SQL, d'une vue modifiable ou d'une méthode de table typée. C'est ce que nous examinerons à la section 21.7.

19.8 TRADUCTION D'UN SCHÉMA CONCEPTUEL

Les sections qui précèdent analysent différentes techniques de représentation des constructions non conformes au modèle relationnel, considérées isolément. Ces transformations doivent encore être regroupées et ordonnancées de manière à constituer une sorte de *procédure de transformation* d'un schéma conceptuel quelconque en un schéma relationnel équivalent. Cette procédure constitue d'une certaine manière la *stratégie* du processus de conception logique. Elle prendra la forme d'un **plan de transformation**. Nous construirons ce plan d'une manière systématique et raisonnée de manière à permettre au lecteur de développer ses propres procédures. On identifie deux étapes dans la construction d'un plan de transformation : (1) le choix d'une représentation privilégiée pour chaque type de constructions non conformes et (2) l'assemblage de ces opérations sous la forme d'une procédure.

On cherchera souvent à obtenir un plan *déterministe*, dans lequel le concepteur n'a pas à intervenir, et qui peut être automatisé. Un tel plan est assez facile à implémenter sous la forme d'une procédure dans un AGL programmable. La conception logique prend alors la forme d'une fonction automatique. Dans le cas contraire, le plan de transformation prend simplement la forme de recommandations (*guidelines*) que le concepteur est invité à suivre en appliquant manuellement, ou de manière semi-automatique, les transformations recommandées.

19.8.1 Choix des représentations privilégiées

Nous choisirons une représentation pour chaque type de constructions non conformes ou considérées comme problématiques. D'autres choix conduiraient à d'autres plans de transformation produisant des schémas relationnels différents.

a) *Traitement des relations is-a*

Les relations *is-a* sont transformées par **matérialisation**, les types d'associations 1:1 étant ensuite transformés en clés étrangères. Un identifiant technique est ajouté si nécessaire. Les contraintes de sous-type sont exprimées sous forme d'annotations.

b) Traitement des attributs composés

Les attributs monovalués composés de niveau 1 sont **désagrégés**. Un préfixe constitué d'une abréviation du nom de l'attribut composé est ajouté aux noms des composants. Si l'attribut est facultatif, on imposera aux composants une contrainte de coexistence s'ils sont obligatoires et des contraintes d'implication (if) si certains sont facultatifs.

c) Traitement des attributs multivalués

Les attributs multivalués de niveau 1 sont transformés en **types d'entités par représentation des instances**.

Le type d'associations fonctionnel qui en résulte est transformé en clé étrangère. Un identifiant technique est ajouté si nécessaire.

d) Traitement des types d'associations fonctionnels

Un type d'associations fonctionnel est transformé en **clé étrangère**. Un identifiant technique est ajouté si nécessaire.

e) Traitement des types d'entités sans identifiant

Un type d'entités qui est destiné à devenir une table référencée doit posséder un identifiant. Si tel n'est pas le cas, un **attribut technique** sans signification est ajouté et forme l'identifiant primaire du type d'entités source.

f) Traitement des contraintes non conformes

Les contraintes non conformes sont traduites pour les structures relationnelles et transcrites sous forme d'**annotations**.

g) Conversion des noms en SQL

Les noms conceptuels sont convertis comme suit :

- les accents sont enlevés ("é" devient "e"),
- les caractères séparateurs interdits (" ", "-", ".", ",", etc.) sont convertis en "_",
- les mots réservés SQL sont remplacés,
- les noms sont mis en majuscules (optionnel).

h) Traitement des identifiants facultatifs (optionnel)

Les attributs d'un identifiant facultatif sont extraits du type d'entités source et attachés à un nouveau type d'entités dont ils forment l'identifiant primaire (variante de la transformation d'attributs en type d'entités par représentation des valeurs). Ces deux types d'entités sont liés par un type d'associations 1:1 qui est ensuite transformé en une clé étrangère ajoutée à la nouvelle table. Un identifiant technique est ajouté si nécessaire.

i) Traitement des identifiants primaires complexes

Dans le schéma relationnel, les identifiants primaires composés d'au moins **m** colonnes sont rendus secondaires et un attribut technique devient le nouvel identifiant primaire. Le nombre **m** est un paramètre de la transformation (**m** = 3 est une valeur raisonnable). L'application de cette règle peut être limitée aux tables référencées afin d'éviter les clés étrangères trop complexes.

19.8.2 Construction du plan de transformation

Cette sélection de techniques de représentation ne constitue pas encore une procédure à proprement parler. Nous allons donc en organiser l'exécution de manière logique sur la base des observations suivantes.

- Le traitement des attributs complexes a été limité aux attributs de niveau 1. Lorsque l'attribut possède une structure à plusieurs niveaux, ce traitement doit être appliqué de manière itérative jusqu'à ce que tous les attributs composants aient été transformés.
- On observe que plusieurs transformations produisent des *types d'associations fonctionnels*. Il serait intéressant d'éviter de les transformer dès qu'ils apparaissent et de leur appliquer globalement la représentation sous forme de clé étrangère à la fin de la procédure.
- Ce principe de regroupement ne fonctionne cependant pas pour le traitement des identifiants facultatifs, qui requiert la conversion préalable des identifiants hybrides en groupes d'attributs monovalués atomiques.
- La représentation d'un type d'associations fonctionnel par une clé étrangère est conditionnée au fait que le type d'entités cible est doté d'un identifiant (primaire) exclusivement constitué d'attributs. Si cet identifiant est hybride, il convient au préalable de transformer le type d'associations apparaissant dans cet identifiant. Le schéma de la figure 12.25 montre que ce problème peut nécessiter plusieurs niveaux de transformations : la transformation de *traite* nécessite la transformation préalable de *de* (entre SERVICE et DEPARTEMENT), laquelle ne pourra se faire que quand *de* (entre DEPARTEMENT et DIRECTION) aura été transformé. Pour éviter une procédure récursive, on propose d'appliquer de manière répétée la transformation en clé étrangère à tous types d'associations fonctionnels jusqu'à ce qu'il n'y en ait plus qui puissent être transformés. Dans la cas de la figure 12.25, la première itération traite *de* (entre DEPARTEMENT et DIRECTION), la deuxième traite *de* (entre SERVICE et DEPARTEMENT) et la troisième peut enfin transformer *traite* et occupe. La quatrième itération est sans effet et clôture le traitement.
- Si, après ces transformations itérées, des types d'associations fonctionnels subsistent, c'est qu'il est impossible de les traiter par manque d'identifiants appropriés (absence ou composition hybride) chez des types d'entités cibles. On ajoutera alors à ces derniers un identifiant technique. Ce processus est désigné sous l'expression "ajouter des id. techniques *nécessaires*".

Compte tenu de ces observations, nous pouvons proposer le plan de transformation de la figure 19.31. Les actions font référence aux choix proposés en 19.8.1.

19.8.3 Limitations d'un plan de transformation

Un plan de transformation peut être incomplet pour différentes raisons parfaitement légitimes, telles que la simplicité. Prendre en compte des situations extrêmement rares pourrait conduire à un plan de transformation d'une complexité telle qu'il

risquerait d'être inutilisable. On observera par exemple que le plan proposé ne traite pas, ou pas correctement, les *identifiants partiellement facultatifs*, les *identifiants multicomposants facultatifs* non soumis à une contrainte de coexistence, les *identifiants cycliques*, les *rôles polymorphiques*, les *clés étrangères liées à des hiérarchies is-a multiples*, les *types d'associations génériques* et les *répartitions is-a multiples*¹⁰.

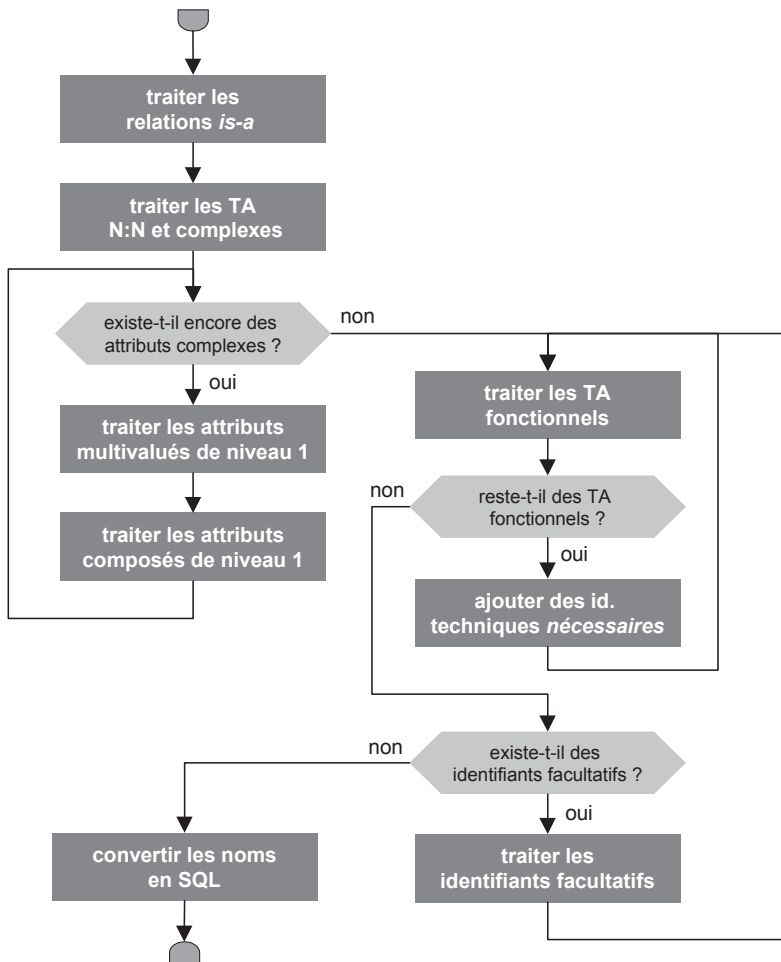


Figure 19.31 - Un plan de transformation pour la conception logique relationnelle

19.8.4 Validation d'un plan de transformation

Pour être valide, un plan de transformation doit être correct. On entend par là qu'il doit réaliser une conception logique effective pour tous les schémas Entité-associations qui respectent ses préconditions. Par analogie avec les algorithmes en

10. L'annexe A25 propose des transformations appropriées à la plupart de ces constructions.

programmation, on s'assurera que le plan de transformation développé respecte notamment trois conditions, pour tout schéma Entité-association qui satisfait à ses préconditions :

- **terminaison** : le plan *se termine* en un nombre fini d'opérations ;
- **conformité** : à la terminaison du plan, le schéma résultant est *conforme* au modèle relationnel ;
- **équivalence** : le schéma résultant est, du point de vue de son contenu en information, *équivalent* au schéma source (ce point est traité dans l'annexe A25).

19.9 DÉFINITION DES VUES

La définition des vues à partir des sous-schémas conceptuels se traite de manière similaire, en appliquant à chaque sous-schéma conceptuel le plan de transformation qui a servi à la production du schéma logique complet. Si des choix de représentation spécifiques ont été effectués, on veillera à les appliquer également aux sous-schémas conceptuels, de manière que les vues logiques et le schéma logique soient compatibles. À titre d'exemple, la figure 19.32 représente la vue relationnelle du sous-schéma 3 de la figure 18.66.

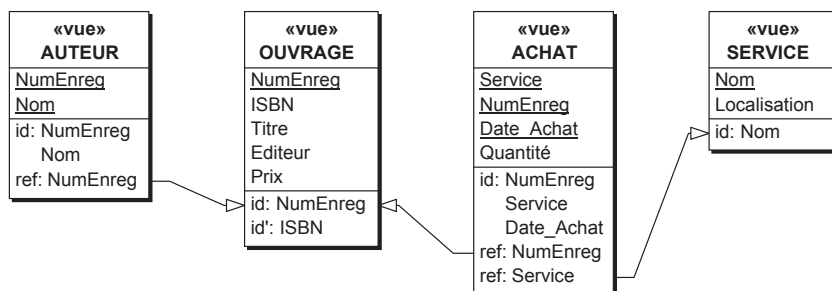


Figure 19.32 - Vue relationnelle du sous-schéma conceptuel n° 3 de la figure 18.66.

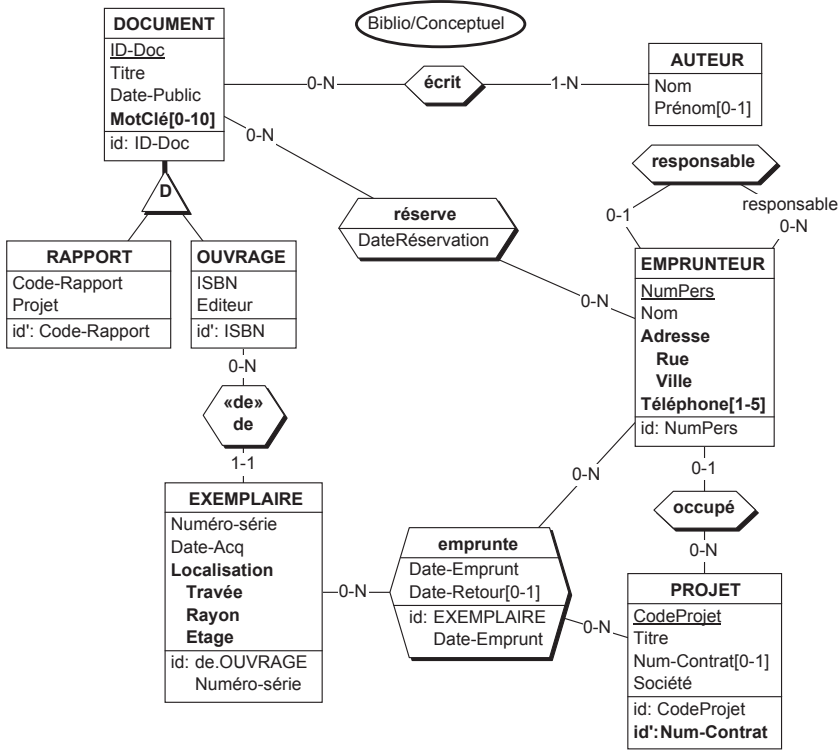
19.10 CONCEPTION LOGIQUE : UN EXEMPLE

Nous illustrerons les principes de la conception logique en les appliquant au schéma de la figure 16.42, repris à la figure 19.33.

Les constructions non conformes y ont été *marquées* (noms en gras, ombrage) :

- relations *is-a* : {RAPPORT, OUVRAGE} is-a DOCUMENT ;
- types d'associations N:N ou complexes : écrit, réserve, emprunte ;
- types d'associations fonctionnels : responsable, de, occupé ;
- attributs composés : Localisation, Adresse ;
- attributs multivalués : MotClé, Téléphone ;
- identifiant facultatif : {Num-Contrat} ;
- noms non conformes à SQL : ID-Doc, MotClé, Prénom, etc.

- contrainte additionnelle.



$$(de * emprunte)[DOCUMENT, EMPRUNTEUR] \cap réserve = \emptyset$$

Figure 19.33 - Schéma conceptuel dont les constructions non relationnelles ont été marquées

Conformément au plan de transformation, les relations *is-a* sont transformées par matérialisation, les types d'associations complexes sont transformés en types d'associations 1:N, les attributs composés sont désagrégés, les attributs multivalués sont transformés en types d'entités. Pour l'instant, nous ne lançons pas encore le traitement des types d'associations fonctionnels. Le résultat est celui de la figure 19.34.

On observe que ce schéma intermédiaire est conforme¹¹ au modèle de base du chapitre 12. Il y reste des constructions non conformes : types d'associations fonctionnels, identifiant nécessaire mais manquant (AUTEUR), identifiant facultatif, cardinalités spéciales ([0-10], [1-5]) et noms non conformes.

La représentation de ces constructions conduit au schéma relationnel de la figure 19.35, conforme au modèle SQL2. Ce schéma inclut des annotations spécifiant les contraintes relatives aux cardinalités spéciales, à la contrainte de disjonction des relations *is-a* et à la contrainte additionnelle. Il resterait à préciser les delete/update modes des clés étrangères, tâche qui fatigue d'avance l'auteur et qui est donc confiée au lecteur.

11. En fait, *presque* conforme. Quelles sont les constructions non conformes au modèle de base ?

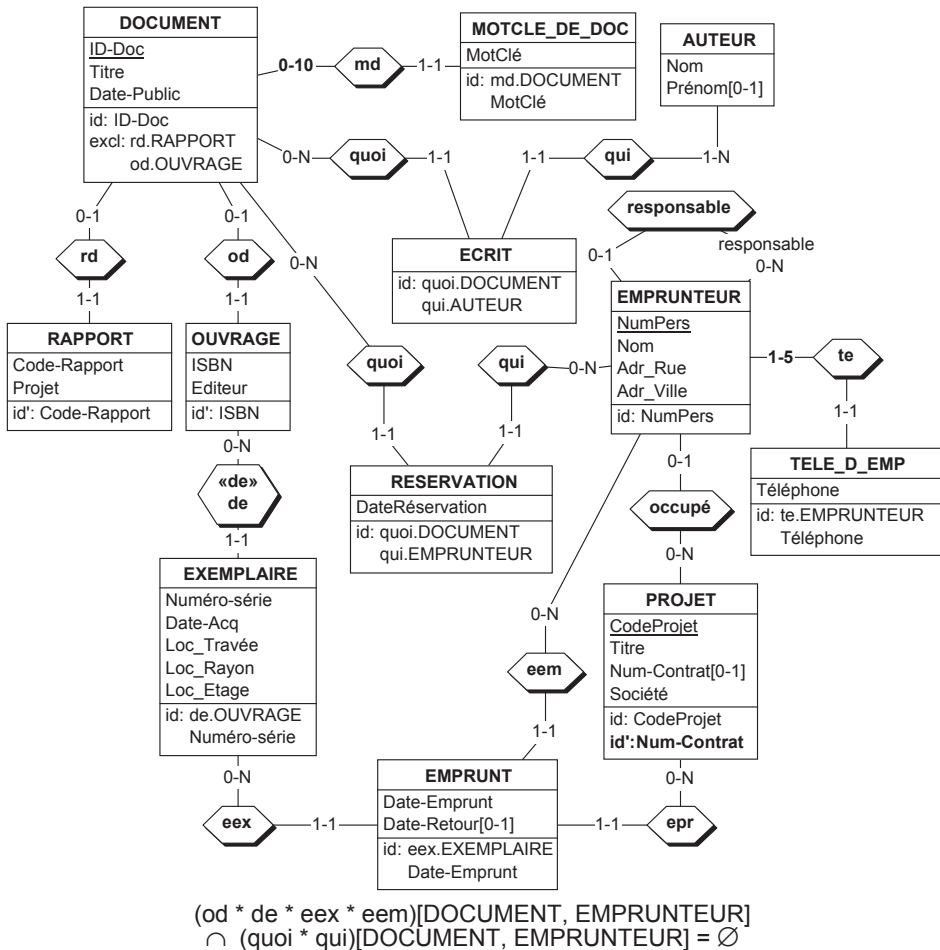


Figure 19.34 - Schéma conceptuel exprimé dans le modèle Entité-association de base. Les constructions non relationnelles sont marquées.

19.11 EXPLOITATION DES CONSTRUCTIONS SQL3

SQL3 ajoute aux structures de données de SQL2 des constructions qu'il peut être utile d'exploiter pour traduire un schéma conceptuel de manière plus expressive. Nous nous concentrerons sur trois des extensions les plus remarquables : les colonnes *composées* (constructeur *row*), les colonnes *multivaluées* (constructeur *array*) et les hiérarchies de tables typées.

Une **colonne composée** est une représentation directe d'un attribut composé d'un type d'entités. Une **colonne multivaluée** pourra être utilisée pour traduire un attribut multivalué. Cependant, la collection de valeurs d'une telle colonne pour une ligne ne forme pas un ensemble, mais bien un tableau.

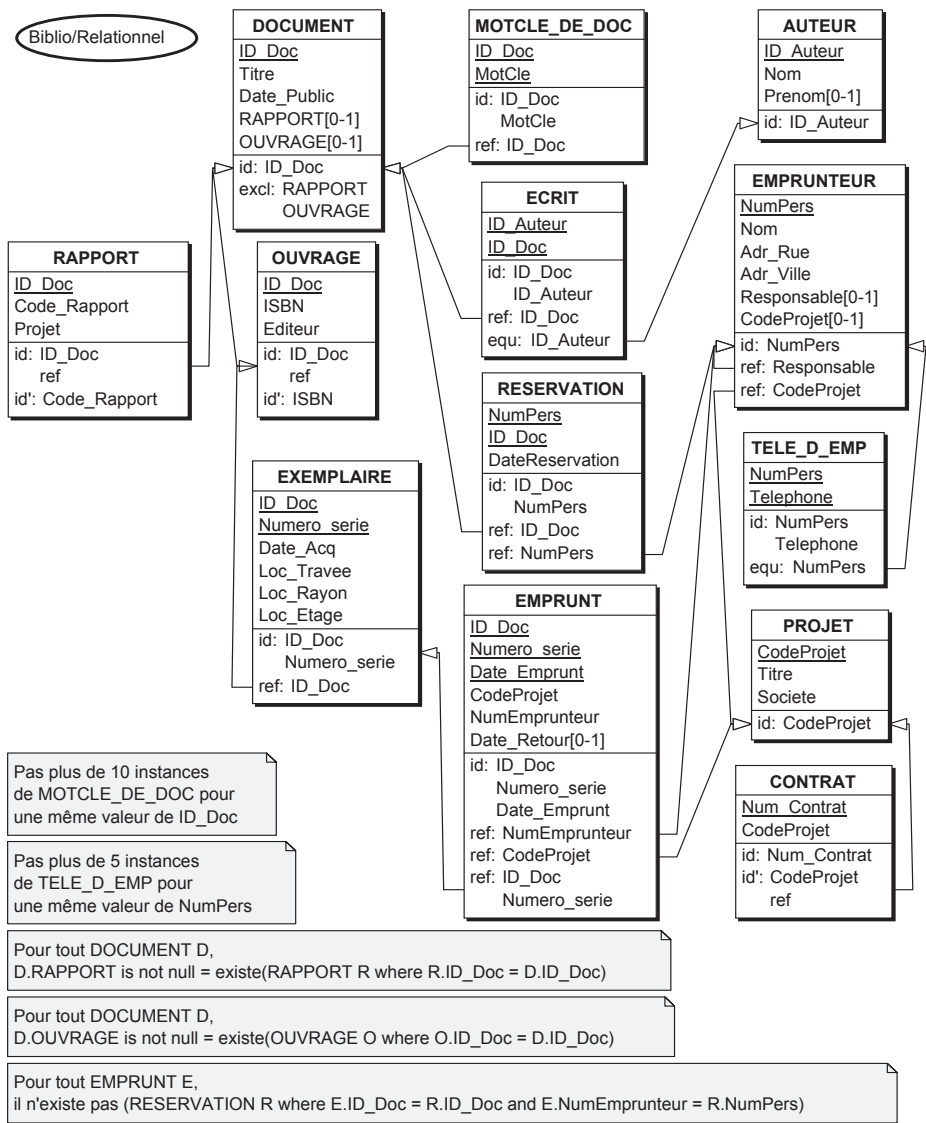


Figure 19.35 - Schéma relationnel SQL2

Comme déjà précisé (section 19.3.6/b), l'utilisation d'un tableau pour représenter des *ensembles* de valeurs n'est que partiellement satisfaisante : un tableau est d'une taille limitée, n'assure pas l'unicité des valeurs qu'il contient, impose un ordre des valeurs¹² et admet les valeurs manquantes. La norme SQL 2003 ajoute le constructeur multiset (multi-ensemble)¹³ qui lève les contraintes sur la taille, l'ordre et les

12. L'ordre des valeurs n'est pas en lui-même une caractéristique négative. Le problème est que l'utilisateur (ou le programmeur) sera tenté d'utiliser cet ordre pour représenter une propriété absente du schéma conceptuel. Exemple : le 1^{er} numéro de téléphone est celui du secrétariat.

valeurs manquantes. L'unicité n'est toujours pas garantie. L'introduction future du constructeur *set* (ensemble) est en discussion.

Les **hiérarchies de types** et les **tables typées** nous permettent de représenter directement des relations *is-a* entre types d'entités. Attention cependant aux limitations de ces constructions imposées par SQL3 : disjonction des sous-types, hiérarchies (surtypes) simples, pas de répartitions multiples.

19.11.1 Le modèle relationnel objet

Tout comme nous l'avons fait pour le modèle SQL2 (section 19.2.2), nous définissons le *modèle relationnel objet* comme l'ensemble des constructions du modèle Entité-association directement exprimables par les structures SQL3 (figure 19.36).

19.11.2 Représentation des attributs complexes

Un *attribut monovalué composé* peut être désagrégé comme en SQL2 ou être traduit sous la forme d'une colonne composée, quel que soit son niveau dans son type d'entité parent et quelle que soit la structure de ses composants.

Un *attribut multivalué composé* sera d'abord traité comme un attribut multivalué. Un *attribut multivalué*, qu'il soit atomique ou composé, peut être soit transformé en type d'entités comme en SQL2, soit être traduit sous la forme d'une colonne multivaluée de type tableau, quel que soit son niveau dans son type d'entité parent et quelle que soit la structure de ses composants s'il est composé. La figure 19.37 illustre ces règles; les contraintes d'unicité y sont spécifiées sous la forme de notes.

Modèle relationnel objet

1. le schéma comporte des types d'entités (appelés *tables*),
2. un type d'entités peut être un sous-type d'un autre type d'entités ; les sous-types d'un type d'entités sont soumis à une contrainte de disjonction,
3. tout type d'entités possède au moins un attribut (appelé *colonne*),
4. un attribut est monovalué ou multivalué (de type tableau), il est atomique ou composé; il est obligatoire ou facultatif,
5. les seules contraintes sont celles qui sont induites par les identifiants (*identifiant primaire* ou *secondaire*), les attributs de référence (*clés étrangères ref*) et les attributs de référence totaux (*clés étrangères equ*) ainsi que les contraintes d'existence (*coex*, *excl*, *at-least-1*, *exact-1*, *if*),
6. le schéma ne contient pas d'autres constructions,
7. les noms des objets respectent la syntaxe SQL.

Figure 19.36 - Définition du modèle relationnel objet

19.11.3 Représentation des hiérarchies *is-a* de types d'entités

Seules les hiérarchies simples sont conformes au modèle relationnel objet SQL3¹⁴.

13. Déjà présent en Oracle sous la forme de nested table.

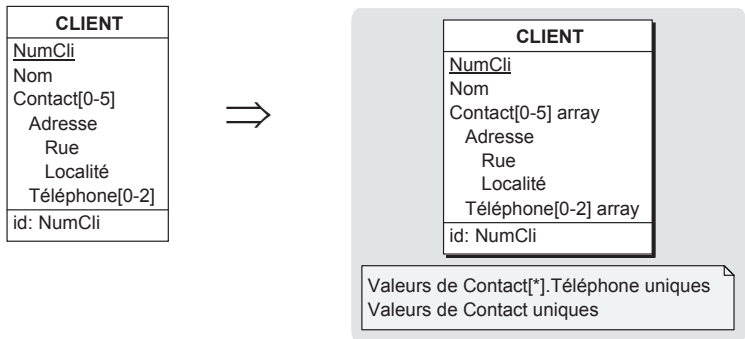


Figure 19.37 - Représentation d'attributs complexes sous forme de colonnes composées et/ou de type tableau

Chaque hiérarchie simple sera représentée par une hiérarchie similaire de TDU puis par une table par TDU. Dans tous les autres cas, on procédera comme pour le modèle SQL2.

Lorsque le schéma comporte des hiérarchies simples et des hiérarchies complexes, on pourrait être tenté de représenter les premières selon la règle de SQL3 énoncée ci-dessus et les dernières selon l'un des procédés proposés pour SQL2 dans la section 19.6. Il en résulterait un schéma irrégulier dans lequel les relations *is-a* seraient traduites selon différentes techniques, schéma qui serait par la suite difficile à utiliser, à maintenir et à faire évoluer. On adoptera donc la règle suivante :

- si le schéma ne comporte que des hiérarchies *is-a* simples, et que cette simplicité ne risque pas d'être remise en cause à moyen terme, ces hiérarchies seront traduites en hiérarchies de tables typées SQL3 ;
- dans tous les autres cas, toutes les relations *is-a* seront traduites en structures SQL2.

19.11.4 Conception logique relationnelle objet : un exemple

La figure 19.38 montre l'application des règles de traduction en SQL3 au schéma conceptuel de la figure 19.33.

19.11.5 Extension de la méthode

La conception logique selon le modèle relationnel objet est traitée plus en détail à l'annexe A26. On y propose de nouvelles techniques de représentation des identifiants, des types d'associations et des hiérarchies *is-a* à sous-types non disjoints. On trouvera aussi à l'annexe A25 un jeu plus complet de transformations permettant notamment de traiter les hiérarchies *is-a* à surtypes multiples et les distributions multiples.

14. Dans certains cas très particuliers, des sous-types non disjoints peuvent être rendus disjoints, mais les préconditions sont très restrictives. Ces cas sont étudiés dans l'annexe A26.

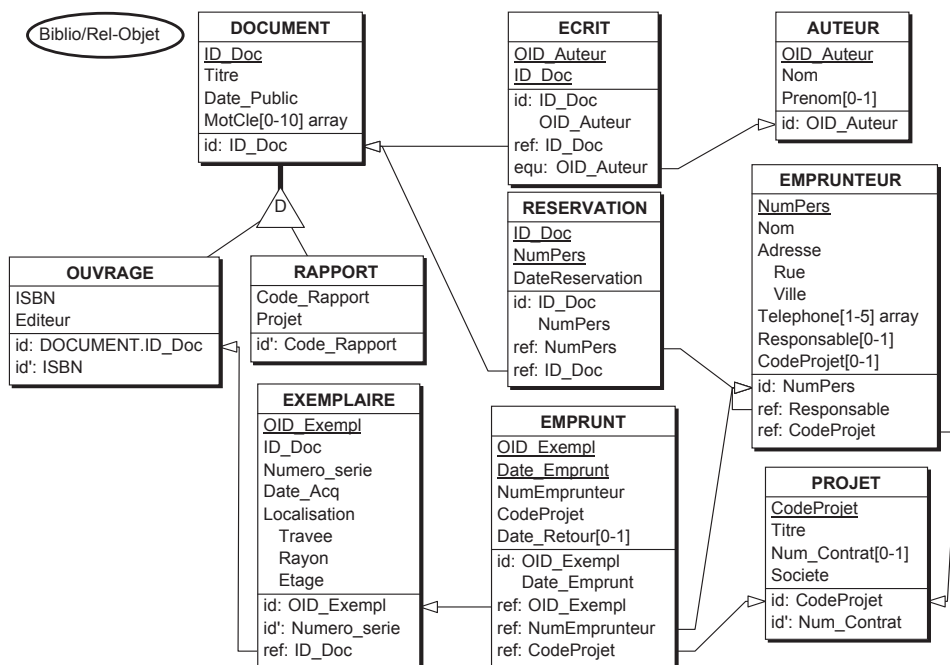


Figure 19.38 - Schéma logique relationnel objet.

19.12 PRODUIRE UN SCHÉMA LOGIQUE À L'AIDE D'UN OUTIL

L'atelier DB-MAIN offre plusieurs outils utiles, généraux ou dédiés à la conception logique d'une base de données à partir d'un schéma Entité-association. Les transformations et l'analyse de schémas constituent l'essentiel de ces outils.

19.12.1 Transformation de schémas

Les outils de transformation de DB-MAIN décrits à la section 18.10 s'appliquent bien sûr au processus de conception logique, particulièrement via les scripts de transformation. La figure 19.39 montre un script de conception logique relationnelle développé à l'aide de l'assistant de transformation avancé. Le langage d'expression de transformations est plus modulaire que celui de l'assistant de base, chaque transformation étant paramétrée par une éventuelle condition de sélection des objets selon la forme **O(C)**, où **O** est un opérateur et **C** la condition que doivent satisfaire les objets auxquels **O** est appliqué (la condition générique ALL correspond à l'absence de condition et peut être omise : O(ALL) s'écrit alors O). Les opérateurs du script de la figure 19.39 s'interprètent comme suit :

- **ISA_into_RT(C)** : transformer les relations *is-a* qui vérifient C en types d'associations ;

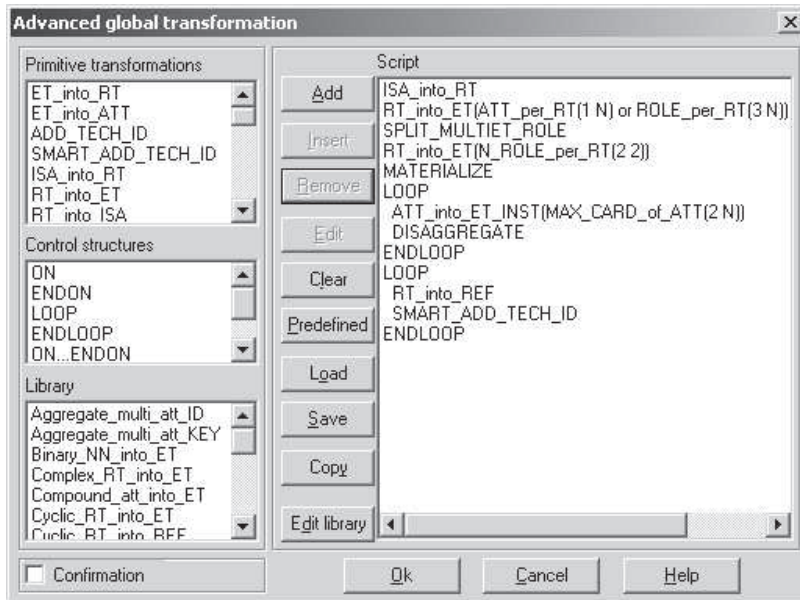


Figure 19.39 - L'assistant de transformation avancé. La partie de droite contient un script de transformation dérivé du plan de transformation 19.31.

- RT_into_ET(C) : transformer les types d'associations qui vérifient C en types d'entités ;
- SPLIT_MULTIET_ROLE(C) : transformer les rôles multi-types qui vérifient C en types d'associations simples ;
- MATERIALIZE(C) : remplacer chaque attribut qui vérifie C par le TDU sur lequel il est défini ;
- LOOP ... ENDLOOP : structure de boucle; les instructions du corps sont exécutées jusqu'à ce qu'elles soient sans effet ;
- ATT_into_ET_INST(C) : transformer tous les attributs qui vérifient C en types d'entités par représentation des instances ;
- DISAGGREGATE(C) : remplacer les attributs composés qui vérifient C par leurs composants ;
- RT_into_REF(C) : transformer les types d'associations qui vérifient C en attributs de référence (clés étrangères) ;
- SMART_ADD_TECH_ID(C) : ajouter un identifiant primaire technique aux types d'entités qui vérifient C *et qui en auront besoin s'ils sont ultérieurement référencés par une clé étrangère*¹⁵.

15. smart = *intelligent*.

Les conditions utilisées dans le script 19.39 sont les suivantes :

- ATT_per_RT(I J) : (types d'associations) qui possèdent de I à J attributs ($N = \infty$) ;
- ROLE_per_RT(I J) : (types d'associations) qui possèdent de I à J rôles ;
- N_ROLE_per_RT(I J) : (types d'associations) qui possèdent de I à J rôles de type *plusieurs* (cardinalité maximum > 1) ;
- MAX_CARD_of_ATT(I J) : (attributs) dont la cardinalité maximale est comprise entre I et J.

La deuxième instruction du script s'interprète comme suit :

RT_into_ET(ATT_per_RT(1 N) or ROLE_per_RT(3 N))

transformer les types d'associations en types d'entités s'ils sont *complexes*; la qualification *complexe* est définie par la condition suivante : le type d'associations possède au moins un attribut (de 1 à N) ou possède au moins trois rôles (de 3 à N).

19.12.2 Analyse de schémas

L'assistant d'analyse de schémas permet, par exemple, de repérer des constructions du schéma courant qui ne satisfont pas la définition du modèle logique relationnel tel que défini à la figure 19.4.

19.13 QUE RETENIR ?

La **conception logique** d'une base de données est le processus qui produit un schéma logique, **conforme au modèle de données** d'une **famille de SGBD**, en l'occurrence, dans ce chapitre, le modèle relationnel ou SQL2.

Le schéma logique est obtenu par des **transformations** successives qui remplacent les constructions du schéma conceptuel non conformes par des constructions relationnelles. L'ordonnancement de ces transformations constitue le **plan de transformation** du processus de conception logique. Le plan proposé dans ce chapitre pour le modèle relationnel SQL2 est organisé comme suit.

1. Chaque type d'entités est représenté par une table et chaque attribut simple par une colonne.
2. Les relations *is-a* sont transformées en types d'associations 1:1, lesquels sont transformés en clés étrangères.
3. Les types d'associations complexes sont transformés en tables association munies de clés étrangères.
4. Les attributs complexes sont transformés itérativement : les attributs composés sont désagrégés et les attributs multivalués sont transformés en tables.
5. Les types d'associations fonctionnels sont transformés en clés étrangères.

6. Les identifiants facultatifs sont transformés en tables.
7. Certaines situations peuvent exiger l'ajout d'un identifiant technique à certaines tables.
8. L'expression des contraintes est rendue conforme aux constructions SQL.
9. Les noms des objets du schéma sont rendus conformes à la syntaxe SQL.

Si on adopte le modèle relationnel objet SQL3, une hiérarchie *is-a* simple pourra aussi se traduire sous la forme de TDU hiérarchisés et de tables typées. Un attribut composé pourra s'exprimer directement en une colonne de type *row* et un attribut multivalué en une colonne de type *array*, du moins si les caractéristiques des tableaux sont admissibles.

Un plan de transformation est une procédure qui doit garantir certaines propriétés : il doit se terminer dans tous les cas, produire un schéma conforme au modèle relationnel et produire un schéma de même contenu informationnel que le schéma source.

On retiendra aussi qu'il existe plusieurs manières de transformer une construction non conforme. Le choix de la technique la plus appropriée peut dépendre des objectifs du concepteur. Citons par exemple la compacité du schéma, la mise en évidence de certains objets du schéma, sa lisibilité, son évolutivité. La conception logique peut donc être réalisée selon plusieurs plans de transformation.

19.14 POUR EN SAVOIR PLUS

Des procédures de production de schémas relationnels ont été publiées depuis de nombreuses années. Parmi les plus connues, citons [Jajodia, 1984] et [Teorey, 1986]. Toutes privilégient un jeu de règles de traduction et présentent peu de variantes. Les approches actuelles se concentrent sur des techniques *transformationnelles et génératives*, selon lesquelles (1) l'essentiel des spécifications est défini à un haut niveau d'abstraction, indépendant de la technologie (le niveau conceptuel pour les bases de données) et (2) il existe un jeu de transformations qui permet de produire le code opérationnel de manière automatique ou semi-automatique. Le développeur est dès lors libéré des soucis liés à l'implémentation. Ces approches, appliquées à tous les composants des applications informatiques et pas seulement les bases de données, sont soutenues par l'OMG, qui a défini un cadre approprié sous le nom de MDA (*Model Driven Architecture*) ou plus généralement MDE (*Model Driven Engineering*). On en trouvera les spécifications sur le site de l'OMG¹⁶. On consultera également l'ouvrage [GTTSE, 2006].

L'approche transformationnelle est très populaire dans le domaine des bases de données et a fait l'objet de développements divers, parfois assez anciens. Citons notamment [Hainaut, 1992], [Rosenthal, 1994], [Fahrner, 1995], [McBrien, 2003] et [Hainaut, 2006]. Le concept de plan de transformation est notamment étudié dans la dernière référence. On y trouvera également une bibliographie sur les transforma-

16. <http://www.omg.org>

tions et le problème de l'équivalence de schémas. Les principales transformations utilisées dans cet ouvrage sont reprises dans l'annexe A25.

Nous avons utilisé des transformations de schémas pour développer des procédures de conception logique. Cependant, leur portée est bien plus large que ce seul processus. En effet, nous avons déjà fait appel aux opérateurs de transformation lors de l'analyse conceptuelle : la construction, la normalisation et l'intégration de schémas réclament en effet de multiples restructurations dont la plupart sont des transformations que nous avons retrouvées dans le présent chapitre. Nous en aurons encore besoin en rétro-ingénierie.

Le concept de *modèle logique* opposé au *modèle conceptuel* est également évoqué dans les travaux de l'OMG dans le cadre d'UML. Comme nous l'avons signalé en 17.16, l'OMG a lancé une initiative [OMG, 2005] destinée à la définition d'un métamodèle pour la modélisation conceptuelle des données, pour SQL et pour XML. En l'absence de propositions concrètes pour la représentation de schémas SQL, on pourra s'inspirer des propositions de S. Ambler [Ambler, 2006] qui rassemblent et unifient plusieurs suggestions déjà publiées.

Sur la question de l'expression relationnelle des relations *is-a*, on lira notamment [Wagner 1989], [Hainaut, 1996] et [Da Silva, 2000].

La littérature sur la conception logique relationnelle objet est en revanche beaucoup plus discrète. On mentionnera par exemple [Soutou, 2012] et, pour le lecteur motivé, [Markos, 2001], [Mok, 2007] et [Mork, 2007]. Rappelons enfin que l'annexe A26 développe plus en détail la conception logique orientée objet.

19.15 EXERCICES

19.1 Transformer le schéma 19.40 en schéma relationnel.

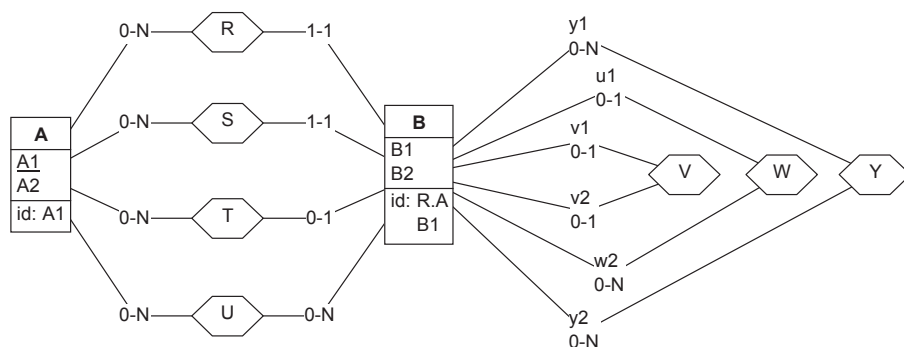


Figure 19.40 - Un schéma conceptuel

19.2 Transformer le schéma 19.41 en schéma relationnel.

19.3 Transformer le schéma 19.42 en structures relationnelles selon l'approche standard (matérialisation). Produire ensuite, en utilisant les représentations par héritage descendant et héritage ascendant, un schéma ne comportant

qu'une seule table (TIERS), un schéma ne comportant que deux tables (CLIENT et FOURNISSEUR) puis un schéma constitué de la représentation exclusive des sous-types. Comparer ces schémas.

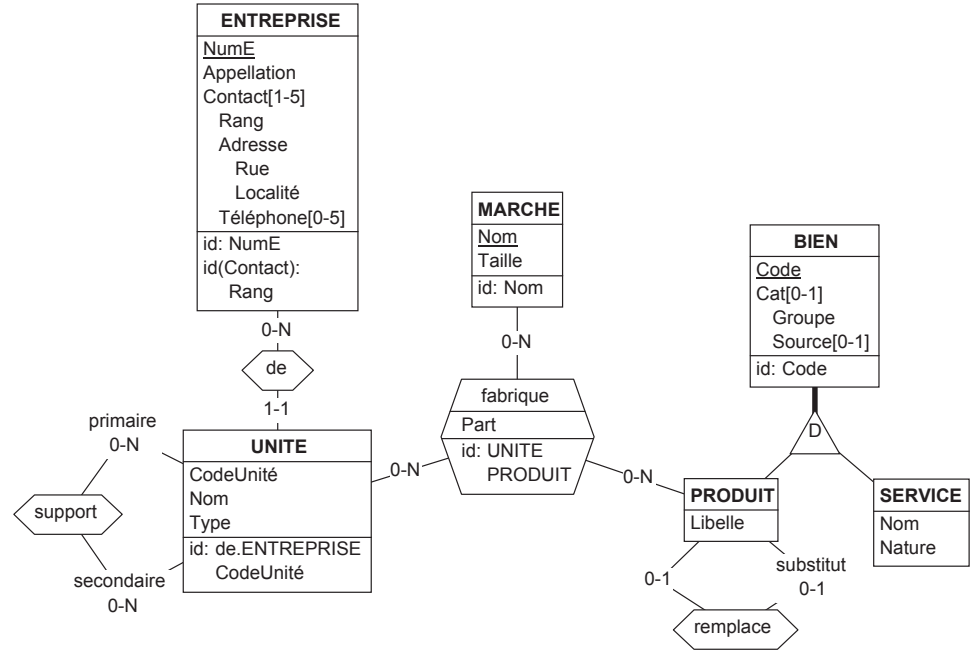


Figure 19.41 - Toujours un schéma conceptuel

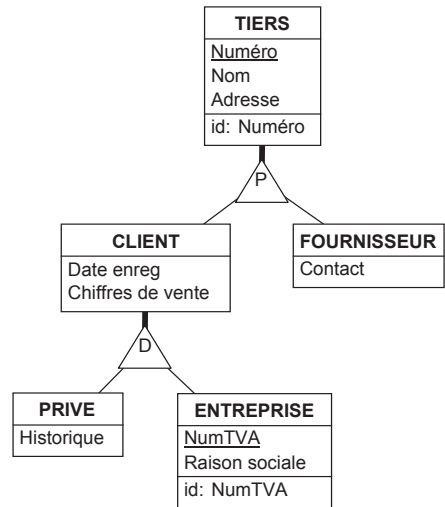
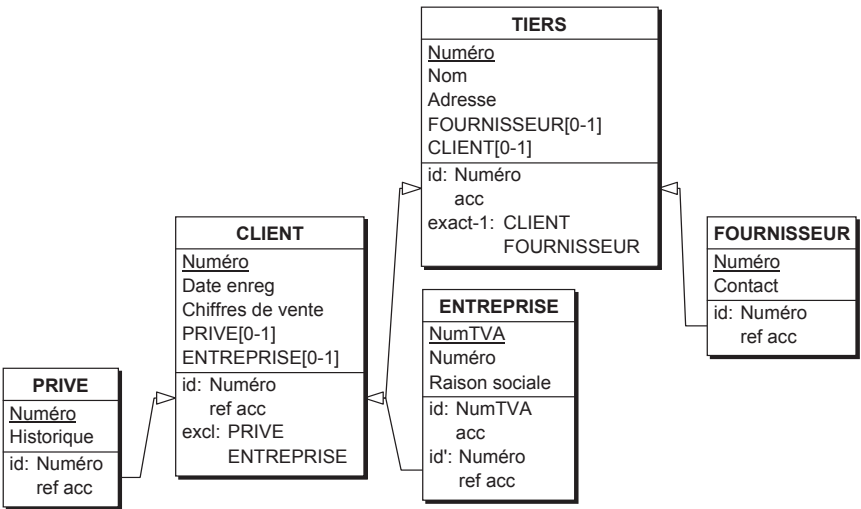


Figure 19.42 - Encore un schéma conceptuel

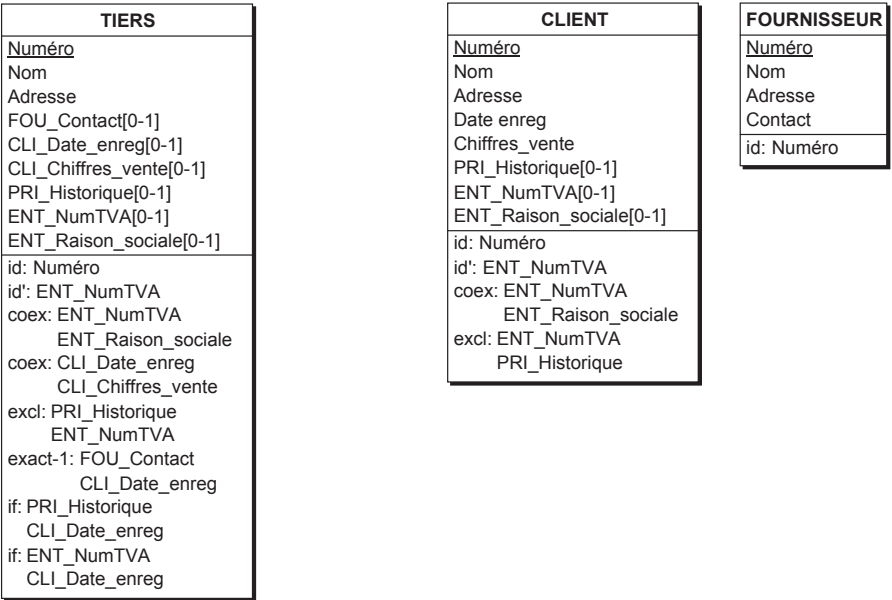
19.4 Exercices de conception logique relationnelle objet : voir annexe A26.

• Exercice 19.3

1. Approche standard (par matérialisation) :



2. Représentation à 1 table (gauche) et à 2 tables (droite) :



3. Représentation exclusive des sous-types :

TIERS_CLIENT_PRIVÉ	TIERS_CLIENT_ENTREPRISE	TIERS_CLIENT_AUTRES	TIERS_FOURNISSEUR
<u>Numéro</u> Nom Adresse Date enreg Chiffres de vente Historique id: Numéro	<u>Numéro</u> Nom Adresse Date enreg Chiffres de vente NumTVA Raison sociale id: Numéro id': NumTVA	<u>Numéro</u> Nom Adresse Date enreg Chiffres de vente id: Numéro	<u>Numéro</u> Nom Adresse Contact id: Numéro

L'expression des contraintes d'intégrité additionnelles dans ces schémas est laissée à l'initiative du lecteur.

Des matériaux complémentaires et des exercices supplémentaires, la plupart accompagnés d'une suggestion de solution, sont disponibles à l'annexe A19.

Chapitre 20

Conception physique d'une base de données relationnelle

La conception physique a pour objectif de produire, à partir d'un schéma logique, une structure de base de données performante spécifique à un SGBD particulier. Dans ce chapitre, on étudiera plus particulièrement la question du choix des index, de leur technologie et des espaces de stockage.

20.1 INTRODUCTION

Le schéma physique d'une base de données est une spécialisation du schéma logique pour un SGBD particulier appartenant à la famille du modèle logique. On parlera de la traduction d'un schéma logique relationnel en un schéma physique Oracle, DB2, SQL Server ou MySQL, dans une version spécifique. Le schéma physique est généralement formé du schéma logique enrichi de constructions techniques telles que les espaces de stockage et les index, ainsi que leurs paramètres. Globalement, la conception physique transforme le schéma logique en un schéma physique qui reprend toutes les constructions du premier (et contient donc la même sémantique), qui est conforme aux particularités du SGBD choisi et qui garantit à la base de données implémentée efficacité et performances (figure 20.1). L'objectif de ce processus est de déterminer le jeu de constructions techniques optimal, celui qui fournira ces performances.

Le chapitre 14 (section 14.8) nous a proposé une (très) courte introduction au concept de conception physique. Pour résumer, on y suggérerait de définir les espaces

de stockage, d'assigner chaque table à l'un d'eux, de définir un index pour chaque identifiant et chaque clé étrangère et de supprimer certains index inutiles.

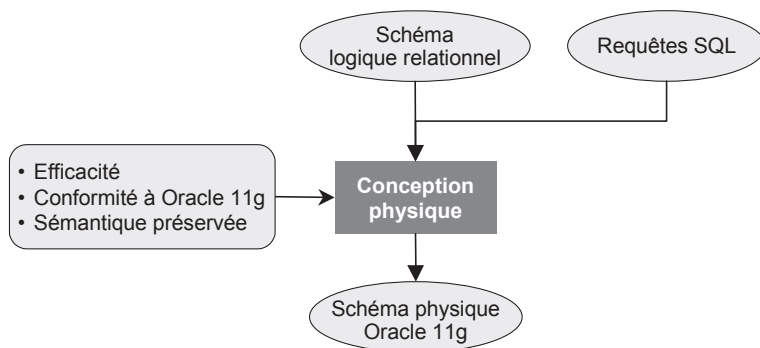


Figure 20.1 - Le processus de conception physique selon un SGBD spécifique

Ces principes restent valables, mais, considérant la taille et la complexité des bases de données développées dans les organisations, et compte tenu des contraintes de performance imposées, il est nécessaire d'approfondir et de compléter ces règles.

Alors que la conception logique est spécifique au modèle de données d'une **famille de SGBD** (chapitre 19), la conception physique est liée à **un SGBD particulier** de cette famille, celui qui sera utilisé pour implémenter la base de données. En principe, la conception physique d'une base de données MySQL est différente de celle d'Oracle, et encore faut-il la particulariser selon la version du SGBD. Cependant, dans le cadre d'un ouvrage introductif, il ne serait pas réaliste de spécialiser ce chapitre pour chacun des SGBD que les lecteurs sont susceptibles d'utiliser. C'est pourquoi nous limiterons notre propos à des principes généraux applicables à la plupart des situations qui se présenteront en pratique. À charge pour le lecteur d'adapter ces principes au SGBD qu'il envisage d'utiliser.

D'autre part, le choix des structures techniques à mettre en œuvre et de leurs paramètres se base généralement sur une comparaison des coûts et des bénéfices qu'entraînent ces structures et leur absence. Par exemple, pour juger de l'utilité d'un index, il est nécessaire d'évaluer le temps de gestion et l'espace qu'il occupe, tant sur disque qu'en mémoire centrale (via les tampons), ainsi que le gain en accès qu'on peut en retirer. L'évaluation a priori de ces coûts et de ce bénéfice est complexe et peu fiable en raison des multiples facteurs dont elle dépend. Bien sûr, le chapitre 4 nous fournit les modèles de calcul des volumes, des temps d'accès et des temps de modification des données pour les principales technologies utilisées par les SGBD. Cependant, des facteurs essentiels tels que les conditions précises du serveur en cours d'exploitation, et notamment son profil de charge, sont impossibles à estimer. On devra donc se contenter d'approximations. Dans le cadre de cet ouvrage, nous nous limiterons à quelques principes intuitifs qui permettront de calibrer des aspects essentiels de la base de données physique au moment de son déploiement. La plupart des paramètres de fonctionnement pourront être ajustés ultérieurement en fonction des résultats et des besoins. Les besoins réels des traitements ne seront pas analysés, sinon de manière superficielle sous la forme d'un jeu

de requêtes SQL accompagnées de leur fréquence (figure 20.1). La procédure conduit à la définition d'un jeu initial d'index, du choix de la technologie de chaque index, des espaces de stockage et du mode de stockage des lignes des tables dans leur espace.

20.2 NOTION D'EFFICACITÉ D'UNE BASE DE DONNÉES

Qu'est-ce qu'une base de données *efficace* ou *performante* ? Globalement, elle doit satisfaire deux critères : occuper un espace minimum et offrir des temps d'accès et de modification réduits, grandeurs que nous avons appris à estimer avec une certaine précision au chapitre 4.

Les volumes

Le premier indicateur physique est l'**espace occupé** par la base de données **en mémoire externe**, sur disque par exemple, le critère à satisfaire étant la minimisation de cet espace. Les modèles de calcul développés au chapitre 4 montrent que le volume d'une table et de ses index dépend de facteurs clairement identifiés. Certains dépendent des conditions d'utilisation, sur lesquelles nous n'avons pas de prise (profil des opérations ou taux d'insertions) tandis que d'autres correspondent à des paramètres que nous pouvons fixer pour optimiser le fonctionnement : technologie, taille des pages, nombre initial de pages, taux d'occupation initial par exemple. Le choix de ces paramètres pour chaque table conditionne le volume total estimé de la base de données et son évolution durant une période déterminée.

Le second indicateur est l'**espace en mémoire centrale** nécessaire au fonctionnement optimal de la base de données pour l'ensemble des opérations projetées sur la base de données. Cet espace est principalement constitué des tampons. On cherchera à minimiser cet espace. Ici encore, le chapitre 4 nous donne les informations nécessaires pour effectuer les choix optimaux, compte tenu d'une taille définie de la mémoire centrale qui doit permettre l'exécution d'un nombre maximum de programmes.

Les temps d'accès

La minimisation des temps d'exécution des principales opérations sur les données, en accès et en modification, constitue le deuxième jeu de critères important devant guider la conception physique. En ce qui concerne la base de données elle-même, c'est avant tout le choix des index et de leur technologie qui seront déterminants. Les temps pourront être estimés pour chaque table et pour chaque index avec les modèles de calcul que nous avons élaborés. Connaissant les principales requêtes SQL que les programmes d'application vont soumettre au serveur ainsi que leur fréquence, on peut estimer (1) le taux d'utilisation des disques (par exemple 6h par jour) et (2) le temps d'exécution d'un programme (réduit aux accès à la base de données). Ces estimations permettent notamment de configurer le serveur de données, notamment le nombre de processeurs, la taille de la mémoire centrale et la l'architecture des mémoires secondaires.

Compromis espace/temps

Les critères de volume et de temps sont généralement conflictuels. Un faible taux d'occupation, un tampon de grande taille et des index conduisent à un espace occupé important mais réduisent les temps d'accès. En ce qui concerne les temps, on observe un autre dilemme : un plus grand nombre d'index conduit à de meilleurs temps d'accès mais aussi à un temps de gestion plus important lors des mises à jour des données et de la réorganisation des tables. Une base de données résulte donc d'un compromis entre de multiples exigences contradictoires.

Autres critères de performance

Il existe d'autres critères qui devront être considérés. Citons-en quelques-uns : fréquence et coût unitaire des réorganisations, coût des sauvegardes (volume et temps), mécanismes de protection contre les incidents (sauvegardes, journaux, redondance de données), coût de restauration en cas d'incident, niveau de parallélisme des transactions (gestion des verrous).

Des critères non techniques sont également essentiels : évolutivité, coût de la maintenance, intégration de données existantes, coûts d'exploitation, facilité de développement des programmes d'application (et donc coûts et délais), compétences nécessaires, sécurité et contrôle d'accès. La satisfaction de ces besoins peut influencer les structures physiques de la base de données.

20.3 CONCEPTION PHYSIQUE D'UNE BASE DE DONNÉES EN 15 RÈGLES

La prise en compte de tous les critères et facteurs décrits ci-dessus conduirait à un processus de conception très complexe, hors de portée d'un ouvrage introductif. Nous proposons donc une approche très simple mais qui conduira à des résultats très satisfaisants dans la plupart des situations pratiques. Si tel n'était pas le cas, on identifiera les parties critiques (les programmes, les tables, les requêtes) pour lesquelles on procédera à une analyse plus approfondie à l'aide des modèles du chapitre 4. Pour des raisons de clarté nous traduirons cette approche sous la forme de quinze règles pratiques regroupées en trois classes : détermination des index, technologie des index et sélection des espaces de stockage.

20.4 DÉTERMINATION DES INDEX

Le choix des index à créer est déterminé par l'accroissement des performances qu'on en attend lors de l'exécution des requêtes de consultation et de modification des données. Nous envisagerons quatre catégories de règles.

20.4.1 Index induits par les structures logiques

On définit ces index sur les structures logiques d'identifiant et de clé étrangère.

Règle 1 On associe un index à chaque identifiant.

Justification. Cet index accélère les opérations de validation des contraintes d'unicité et d'intégrité référentielle lors des insert et de certains update. Il accélère les accès basés sur l'identifiant, majoritaires en mode transactionnel et il accélère certaines jointures. Il favorise par exemple la requête :

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

La plupart des SGBD exigent d'ailleurs que tout identifiant soit accompagné par un index.

Règle 2 On associe un index à chaque clé étrangère.

Justification. Cet index accélère l'exécution et la propagation des delete et des update d'un identifiant cible d'une clé étrangère, il favorise certaines jointures, telle que celle de la requête :

```
select NCOM, DATECOM
from   COMMANDE M, CLIENT C
where  M.NCLI = C.NCLI;
```

Il accélère certains protocoles de gestion des verrous lors des mises à jour (par exemple, verrouillage d'une ligne de CLIENT et des lignes qui en dépendent).

20.4.2 Index induits par l'utilisation des données

Ces index *non identifiants* (les autres sont pris en compte par la règle 1) sont déterminés par les besoins en accès des programmes d'application. Il est donc utile de procéder à une analyse, éventuellement sommaire, des requêtes SQL qui vont être envoyées au serveur par ces programmes. Une telle approche est en toute généralité assez complexe (voir [Garcia-Molina, 2008] par exemple). Elle suppose qu'on connaisse les techniques d'optimisation que le SGBD met en œuvre au moment de l'exécution. Pour cette raison, nous limiterons le raisonnement à quelques règles de bon sens mais que nous justifierons techniquement.

Règle 3 On associe un index à chaque colonne (ou combinaison de colonnes) qui fait l'objet de **comparaisons fréquentes** dans des conditions d'égalité (=) ou d'appartenance (in).

Justification. Cet index peut accélérer la vérification de ces conditions par un accès rapide à l'un des membres de la comparaison, en particulier si l'index comprend un grand nombre d'entrées. Typiquement, on suggère la création d'un index sur la colonne C lorsque la requête "select * from T where C = k " est fréquente. Cette règle, d'une importance majeure, mérite d'être affinée. En effet, on a montré par un exemple (section 4.11) que, dans certains cas, l'accès par index entraine en concurrence avec l'accès séquentiel, qui peut s'avérer le plus rapide. Les règles de choix du meilleur plan d'accès sont décrites à la section 20.9. On y trouvera la description d'une feuille de calcul permettant d'étayer le choix ou le rejet d'un tel index.

Règle 4 On associe un index ordonné à chaque colonne (ou combinaison de colonnes) servant fréquemment de **critère d'intervalle** (between, >, >=, <, <=, like 'DUPONT%').

Justification. Un index ordonné, primaire ou secondaire, sur la colonne **C** permet un accès rapide aux lignes répondant à une requête du type " select * from T where C between k1 and k2 ". Les scénarios et les raisonnements de la règle 3 concernant l'utilité d'un index sont directement transposables ici. Pour les scénarios avec index, le chapitre 4 nous fournit directement les outils nécessaires pour le calcul des temps d'accès par intervalle (sections 4.8.3 et 4.13.3).

Règle 5 On associe un index ordonné à chaque colonne (ou combinaison de colonnes) servant de **critère de groupement** (group by) ou de **tri** (order by) fréquemment utilisé.

Justification. Un index ordonné sur la colonne **C** permet d'accéder aux lignes du fichier de base dans l'ordre des valeurs de **C**. Le calcul des groupes sur **C** (group by **C**) et la sortie du résultat ordonné selon **C** (order by **C**) en sont donc favorisés. Attention cependant, l'usage d'un tel index n'est utile que **si la table est elle-même ordonnée selon C**, c'est-à-dire si l'index est primaire en séquentiel indexé ou secondaire sur fichier ordonné (*clustering index*). En effet, si la table est en vrac selon **C**, alors le SGBD pourra estimer plus efficace de procéder à un tri interne ou externe (voir section 4.6.2 pour le calcul) des lignes sélectionnées.

20.4.3 Index à écarter

Certains index dont les règles précédentes suggèrent la création peuvent être écartés si on peut démontrer leur inutilité.

Règle 6 Des critères tels que l'inégalité (<>), la non-appartenance (not in) ou like '%DUPONT%', ne peuvent, sauf cas très particuliers, profiter de la présence d'un index.

Justification. Ces critères ne peuvent se réduire à l'égalité à une valeur ou à l'appartenance à un intervalle. Un index standard est donc sans utilité.

Règle 7 Si la suite des composants d'un index **K2** est préfixe¹ de celle d'un index **K1**, et si **K1** est un index ordonné (primaire séquentiel indexé ou secondaire), alors il est recommandé de supprimer **K2**. Si l'ordre des composants de l'index le plus complet (**K1**) est indifférent, on réordonne ces composants de manière à faire apparaître en préfixe les composants d'un autre index. Par exemple, on remplace les index {**A,B**} et {**B**} par {**B,A**} et {**B**} de manière à pouvoir supprimer ce dernier (figure 20.2).

Justification. Un index ordonné permet l'accès sur un critère partiel, portant sur le début de la valeur de la clé. Cet accès se traite comme un critère d'intervalle. Tout préfixe de la suite de **K1** constitue bien un début de valeur de clé.

1. Les préfixes de la suite {**A, B, C**} sont les suites {**A**} et {**A, B**}.

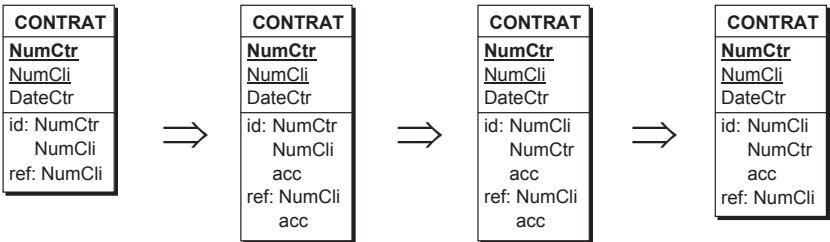


Figure 20.2 - Définition d'un jeu d'index minimum

Règle 8 Si un index non identifiant est défini sur une colonne C de domaine de valeurs très petit (quelques dizaines à quelques centaines de valeurs par exemple), et si les lignes sont rangées aléatoirement selon C, on peut envisager de le supprimer.

Justification. Chaque valeur de l'index donne accès à une proportion importante des lignes de la table qui par ailleurs sont distribuées aléatoirement dans l'espace de celle-ci. Ces lignes occupent en général un grand nombre de pages qui feront l'objet d'un accès aléatoire. Dans ce cas, un accès séquentiel aux lignes de la table pourra s'avérer moins coûteux, de sorte que l'index sera ignoré par l'optimiseur. La section de complément (20.9) à la fin de ce chapitre rappelle le modèle de choix d'un index tiré des résultats du chapitre 4.

Remarque. Cette règle doit cependant être nuancée lorsque le nombre de lignes sélectionnées par une valeur déterminée de C est très petit, bien que le nombre moyen soit élevé. Considérons un exemple limite, celui d'une colonne SEXE de la table EMPLOYE, dans une organisation occupant 99% d'hommes. Si un index est défini sur cette colonne, la condition SEXE = 'H' sera évaluée par un accès séquentiel alors que la condition SEXE = 'F' sera certainement évaluée via l'index. D'ailleurs, connaissant d'avance cette répartition inégale, on aura sans doute créé un *index partiel* limité à la population SEXE = 'F' (section 4.10.4)

Règle 9 Si un index non identifiant est défini sur une table de petite taille (quelques centaines de Ko par exemple), on peut envisager de le supprimer.

Justification. Si cette table est fréquemment utilisée, ses pages seront probablement résidentes dans le tampon. Une lecture séquentielle de ses lignes ne réclamera pas d'accès au disque et sera donc aussi rapide qu'un accès par index.

20.4.4 Maintenance du jeu d'index

Le jeu d'index élaboré par les neuf règles précédentes constitue un premier essai qui devra certainement être ajusté lors du fonctionnement de la base de données, non seulement en raison de l'évolution des besoins, mais aussi à cause de l'imprécision des évaluations qui a conduit à la sélection de ces index.

Règle 10 Lors de l'exploitation, on surveillera le taux d'utilisation des index ainsi que les plans d'exécution élaborés par l'optimiseur du SGBD pour les requêtes les plus critiques.

Justification. La plupart des SGBD offrent, parfois sous forme graphique², une fonction EXPLAIN qui fournit le plan d'exécution d'une requête ainsi que des informations sur son coût d'exécution. On supprimera les index non rentables, c'est-à-dire ceux qui sont peu utilisés et dont la maintenance est coûteuse. En revanche, on évaluera l'intérêt d'ajouter les index susceptibles de remplacer l'accès séquentiel filtré apparaissant dans les plans d'exécution par un accès par index. Enfin, on repérera les index temporaires que le plan d'accès crée puis supprime pour l'exécution de certaines requêtes. Ici encore on évaluera l'opportunité de créer ces index de manière permanente.

20.5 TECHNOLOGIE DES INDEX

L'organisation technique des index influence leur performances. Parmi les différentes techniques, nous en retiendrons quatre : table organisée en séquentiel indexé, table organisée en calculé, index secondaire ordonné et table en vrac, index secondaire ordonné et table ordonnée (*clustering index*).

Règle 11 Les techniques basées sur un dictionnaire de valeurs ordonné (organisé en séquentiel indexé) seront préférées lorsque l'accès par intervalle à de petites séquences d'enregistrements est fréquent. Lorsqu'en outre l'accès par intervalle risque de produire des séquences relativement longues ou lorsqu'un accès séquentiel ordonné est fréquent, alors on recommandera un index primaire en séquentiel indexé ou un index secondaire sur fichier ordonné.

Justification. Voir règle 4.

Règle 12 Les techniques basées sur une organisation séquentielle indexée (table et/ou dictionnaire de valeurs) sont recommandées lorsque (1) le volume de données croît de manière imprévisible, (2) le temps d'accès par index doit être uniforme pour tous les enregistrements et (3) les réorganisations sont fréquentes. Dans les cas contraires, les techniques calculées pourront être sélectionnées.

Justification. Propriétés comparées dans la section 4.9.6.

Règle 13 Les techniques basées sur une organisation calculée (table ou dictionnaire de valeurs) sont appropriées lorsque (1) le temps moyen d'accès par clé doit être très court et (2) la place mémoire occupée sur disque et dans le tampon doit être minimale.

Justification. Propriétés comparées dans la section 4.9.6

Règle 14 Si un index est susceptible d'être supprimé et recréé à tout moment, sans suspension du fonctionnement de la base de données, alors il doit être secondaire.

Justification. Propriétés comparées dans la section 4.9.6

2. Via *Visual Explain* notamment, qui s'interface avec DB2, mais aussi avec PostgreSQL, SQL Server, Sybase et Oracle, entre autres. Une version MySQL existe mais en semi-graphique.

Lorsqu'on chargera les données initiales d'une table, il conviendra de déterminer le moment auquel les index secondaire seront créés : avant ou après le chargement. Cette question a été traitée à la section 4.12.

Certaines requêtes concernant un petit nombre de colonnes peuvent être accélérées par un index secondaire augmenté qui évite l'accès à la table (section 4.10.6).

20.6 SÉLECTION DES ESPACES DE STOCKAGE

Chaque table est assignée à un espace de stockage. Il est en général possible de ne pas monter³ en ligne un espace de stockage. Dans ce cas, les tables qu'il contient ne sont pas accessibles par les programmes d'applications. L'implémenteur de la base de données s'appuiera avant tout sur les conseils des manuels d'administration de SGBD. Néanmoins, nous suggérons la règle suivante, d'application générale.

Règle 15 Dans une grande base de données, s'il est possible d'identifier un ensemble de tables utilisées simultanément mais de manière sporadique, alors on envisagera de les stocker dans un ou plusieurs espaces qui leur sont exclusivement réservés.

Justification. Possibilité de déconnecter temporairement les tables moins souvent utilisées.

20.7 OPTIMISATION AVANCÉE

Seules ont été considérées jusqu'ici les constructions simples tels que les index et les espaces de stockage. Selon leur disponibilité on envisagera éventuellement l'utilisation de structures plus complexes telles que les index augmentés, les index partiels ou les *clusters*, ainsi que tous les mécanismes propres au SGBD adopté qui pourraient améliorer les performances.

En cas de stricte nécessité, lorsque tous les efforts d'optimisation entrepris jusqu'ici s'avèrent insuffisants pour satisfaire les critères d'espace et de temps, on pourra recourir à des *restructurations du schéma* en vue d'améliorer ses performances. De telles pratiques sont à déconseiller car elles dénaturent le schéma logique et dégradent des qualités essentielles telles que sa lisibilité, sa maintenabilité et son évolutivité. Elles compliquent notamment la rédaction des programmes d'application. On pourrait les voir comme le *détricotage* raisonné du processus de normalisation décrit à la section 18.7.

On distingue trois familles de restructuration d'optimisation : les restructurations sans redondance, les redondances structurelles et la dénormalisation.

3. Pour un système d'exploitation, *monter* une unité consiste à la rendre visible et accessible par le système et donc par ses utilisateurs.

20.7.1 Restructurations sans redondance

Ces techniques, qui procèdent à une nouvelle répartition des données, ne sont limitées que par l'imagination du concepteur. Nous ne mentionnerons que les plus connues : partitionnement et fusion de tables. Dans les exemples illustrant ces transformations, nous n'avons pas inclus les contraintes d'intégrité des schémas résultants. Le lecteur complètera sans difficulté ces schémas.

a) Réordonner les colonnes d'une table

Lorsque les valeurs des colonnes d'une ligne sont stockées séquentiellement (section 4.5.3), il peut être utile de réordonner la déclaration de ces colonnes de manière qu'apparaissent en début d'enregistrement :

- les colonnes les plus demandées (y compris identifiants et clés étrangères),
- les colonnes obligatoires, de petite taille, de taille fixe.

Ceci permet, dans le cas de lignes de grande taille ($N_r > N_p$) de réduire le temps d'accès et de décodage pour extraire les valeurs des colonnes demandées.

b) Partitionnement vertical d'une table

Les colonnes d'une table source sont réparties dans deux (ou plusieurs) tables qui se partagent le même identifiant (figure 20.3). Cette répartition s'effectue selon le taux d'utilisation des colonnes. Ici, on suppose que les données de la table CLIENT réduite sont utilisées fréquemment tandis que celles de l'autre table sont utilisées plus rarement.

Les lignes de chacune des tables résultantes sont plus courtes. Chaque table occupera moins d'espace sur le disque, ce qui réduira notamment (1) le temps de lecture séquentielle, (2) la taille de l'index si la table est organisée en séquentiel indexé, (3) le temps de copie de la table (backup). On pourra aussi laisser hors ligne la seconde table lorsqu'elle n'est pas utilisée. Le paramètre **Mrpp** étant augmenté, le comportement des index primaires s'améliore.

Parmi les inconvénients, mentionnons la moindre lisibilité du schéma, un nombre supérieur d'index et la nécessité d'effectuer une jointure lorsque les données des deux tables sont nécessaires.

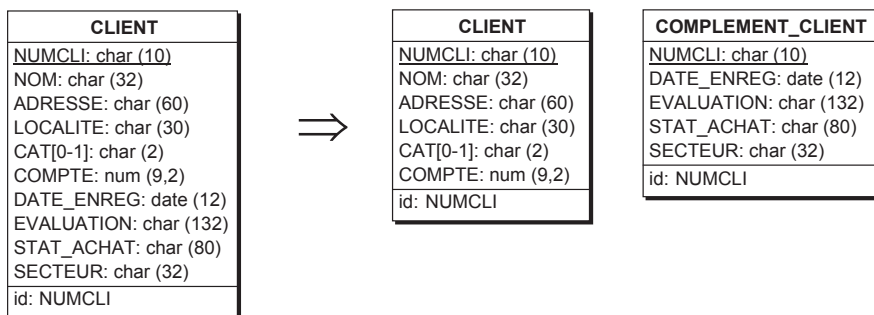


Figure 20.3 - Partitionnement vertical d'une table

c) *Partitionnement horizontal d'une table*

Les lignes sont distribuées dans deux (ou plusieurs) tables de même schéma selon un critère de fréquence d'utilisation⁴ ou de type d'exploitation. Dans l'exemple de la figure 20.4, on a stocké dans la table CLIENT_NATIONAL les lignes décrivant les clients locaux, dont on suppose qu'ils sont le plus fréquemment concernés par le traitement informatique. De ce fait, la table la plus utilisée contiendra moins de lignes, ce qui réduira (1) le temps de lecture séquentielle, (2) la taille de l'index si la table est organisée en séquentiel indexé, (3) la taille des index secondaires, (4) le temps de copie de la table.

En revanche, le schéma devient plus complexe, moins lisible. En particulier, les identifiants de la table sources deviennent globaux. Les traitements sur la totalité des lignes sources nécessiteront de procéder sur l'union des deux tables, ce qui entraîne une plus grande complexité des traitements.

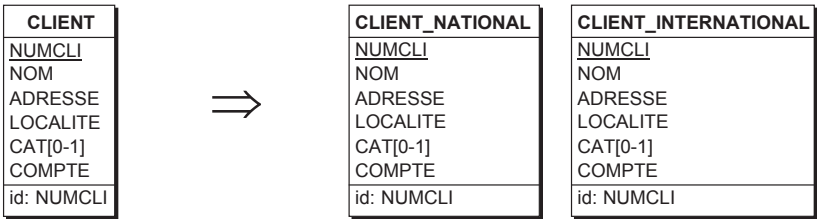


Figure 20.4 - Partitionnement horizontal d'une table

d) *Fusion verticale de tables*

Cette restructuration remplace deux tables unies par une clé étrangère identifiante (type d'associations 1:1) par une table unique qui réunit leurs colonnes (figure 20.5). La fusion est d'une certaine manière l'inverse du partitionnement. Son évaluation se déduit donc des avantages et des inconvénients de cette dernière. Ajoutons encore que le schéma ainsi que les programmes deviennent moins lisibles car la nouvelle table regroupe des données de nature différente.

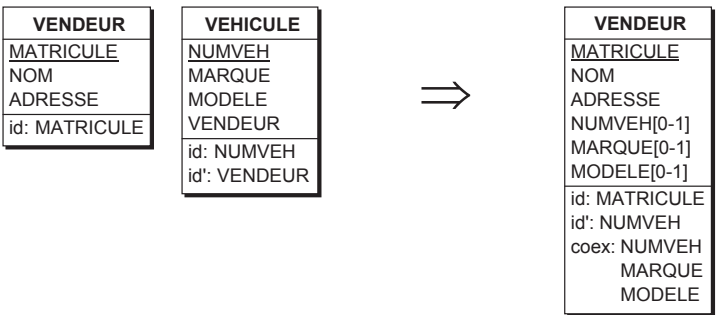


Figure 20.5 - Fusion verticale de deux tables

4. Selon la loi de Pareto 80-20 par exemple : 80% des traitements concernent 20% des lignes.

e) **Fusion horizontale de tables**

Deux tables de schéma similaire sont remplacées par une seule table qui rassemble leurs lignes. Ces tables décrivent des entités de nature différente mais similaire, par exemple des clients et des fournisseurs (figure 20.6). Ici encore, la réduction du nombre de tables et d'index se paie d'une plus grande complexité du schéma et des programmes.

20.7.2 Redondances structurelles

On ajoute au schéma des constructions dont les instances sont calculables à partir de celles de constructions existantes. Considérons notre schéma de la figure 2.6. Si on introduit dans la table COMMANDE les colonnes NOM, ADRESSE, LOCALITE copiées de la table CLIENT (figure 20.7) on évite dans certains traitements la jointure nécessaire à l'acquisition de ces données.

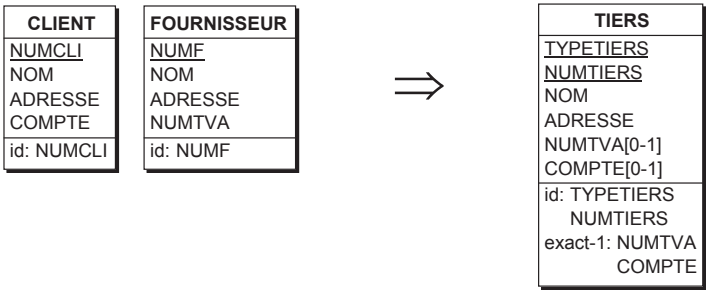


Figure 20.6 - Fusion horizontale de deux tables similaires

La valeur de la nouvelle colonne MONTANT est calculée à partir des valeurs de QCOM et PRIX et la valeur de la nouvelle colonne MONTANT_TOTAL est calculée comme la somme des MONTANT des lignes de DETAIL qui lui sont attachées.

Si le gain en accès est évident dans certains cas, il se paie assez lourdement d'autres côtés : accès séquentiel plus long, tables et index plus volumineux, complexité de gestion de la redondance (via des triggers), schéma plus complexe et moins lisible, évolution plus complexe.

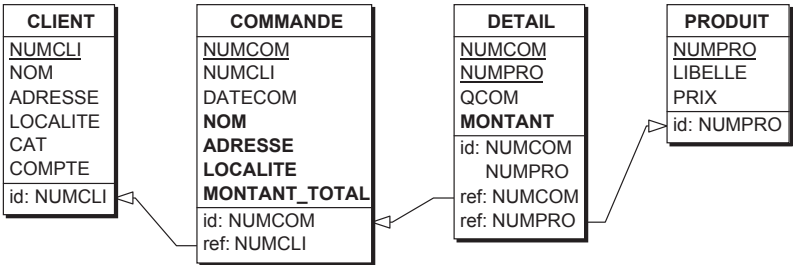


Figure 20.7 - Colonnes dupliquées et calculées

20.7.3 Dénormalisation

On envisagera deux techniques, avec et sans redondance interne.

a) Dénormalisation avec redondance interne

La dénormalisation proposée est l'inverse de la normalisation relationnelle, étudiée au chapitre 3. Elle consiste à remplacer deux tables en association 1:N via une clé étrangère non identifiante par leur jointure. La figure 20.8 montre une nouvelle version de la table DETAIL qui intègre les données des commandes, lesquelles ont perdu leur représentation autonome, bien qu'il soit possible de les reconstituer :

```
select distinct NCOM,NCLI,DATECOM
from DETAIL;
```

Les avantages (on évite une jointure) et les inconvénients (redondance) de cette transformation sont faciles à identifier.

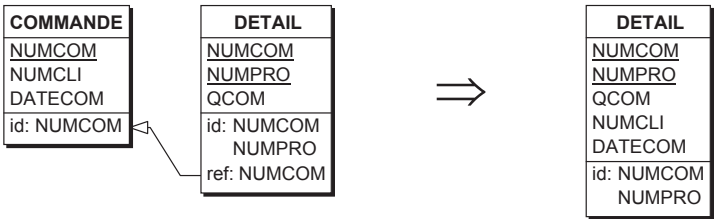


Figure 20.8 - Dénormalisation des tables COMMANDE et DETAIL (1)

b) Dénormalisation sans redondance interne

On peut intégrer la table DETAIL à la table COMMANDE sous la forme d'une colonne complexe. COMMANDE n'est plus en première forme normale. Cette structure, autorisée par SQL3, mais aussi par les nouveaux SGBD orientés documents (chapitre 10), peut offrir à la fois une vue objet des données et de meilleures performances d'accès par l'élimination des jointures COMMANDE * DETAIL. En revanche, le schéma perd de sa symétrie, la jointure PRODUIT * DETAIL devenant beaucoup plus complexe.

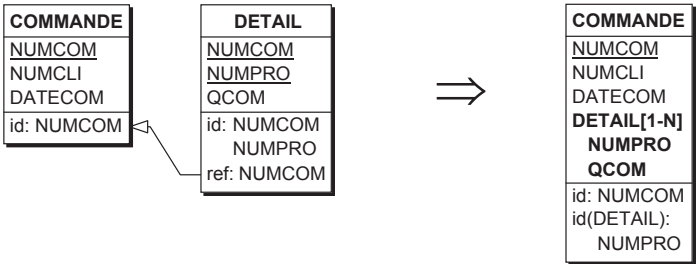


Figure 20.9 - Dénormalisation des tables COMMANDE et DETAIL (2)

20.8 ESTIMATION DES VOLUMES ET DES TEMPS DE TRAITEMENT

Lorsque les structures de données (espaces de stockage, tables, index, clusters) ont été choisies et lorsque les principales requêtes ont été identifiées, ainsi que leur fréquence, il est possible de calculer une estimation du volume de la base de données et des temps d'accès selon les règles et modèles du chapitre 4. Cette estimation permet d'évaluer les caractéristiques de la configuration informatique nécessaire au bon fonctionnement de la base de données et des programmes d'applications.

20.9 COMPLÉMENT : MODÈLE DE CHOIX D'UN INDEX

On propose ci-dessous une synthèse, appliquée à un exemple, du modèle développé à la section A4.7 de l'annexe A4, en guise d'éclairage des règles 3 et 8. Examinons la requête suivante.

```
select NCLI, NOM, ADRESSE
from   CLIENT
where  LOCALITE = 'Poitiers';
```

Lors de son évaluation, le SGBD utilisera-t-il l'index qui a été défini sur LOCALITE ? Admettons les valeurs suivantes :

N_r = 500 000 enregistrements,
 N_p = 25 000 pages,
 N_v = 1000 valeurs distinctes de LOCALITE.

On en déduit :

$N_{rpp} = N_r / N_p = 20$ enregistrements par page en moyenne,
 $N_{rpv} = N_r / N_v = 500$ enregistrements par valeur de LOCALITE en moyenne,

Estimons à présent le coût de l'évaluation de la requête selon les deux stratégies en concurrence.

a) Évaluation par lecture séquentielle

Le temps t_{q1} de lecture séquentielle de la table se calcule comme suit (section 4.6.1), en admettant une lecture anticipée d'une piste :

$$t_{q1} = t_{lsf2} = N_p \times t_{ls1} = 25\,000 \times 0,184 = 4\,600 \text{ ms.}$$

b) Évaluation via l'index

Nous devons faire quelques hypothèses : (1) quel que soit le mode de stockage des lignes, celles-ci sont en désordre par rapport aux valeurs de LOCALITE, (2) la probabilité que plusieurs lignes dont LOCALITE = 'Poitiers' se trouvent dans une même page est très faible. Les lignes sont distribuées aléatoirement dans l'espace de la table, de sorte que l'accès au fichier de base comprendra 500 accès à des pages d'adresses aléatoires. En négligeant la consultation de l'index, le temps d'accès est de :

$$t_{q2} = N_{rpv} \times t_{la1} = 500 \times 12,3 = 6\,150 \text{ ms.}$$

c) Conclusion

Il apparaît donc que l'utilisation de l'index conduit à un coût d'évaluation supérieur, sans même comptabiliser le temps que réclamera sa gestion. La création de cet index est donc déconseillée.

Calculons la valeur pivot de N_v , c'est-à-dire la valeur pour laquelle les deux temps d'exécution sont identiques. On a les relations suivantes :

$$t_{q1} = N_p \times t_{ls1}$$

$$t_{q2} = N_{rpv} \times t_{la1} = N_r / N_v \times t_{la1}$$

$$N_p \times t_{ls1} = N_r / N_v \times t_{la1}$$

On en tire la valeur pivot :

$$N_v = (N_r / N_p) \times (t_{la1} / t_{ls1}) = N_{rpp} \times t_{la1} / t_{ls1}$$

Soit, ici,

$$N_v = 1337, \text{ et } t_{q1} = t_{q2} = 4\,600 \text{ ms.}$$

Une conclusion prudente pourrait être qu'un index sur LOCALITE ne serait utile pour l'évaluation de notre requête que si le nombre de valeurs était supérieur à 1 500.

Cette petite expérience ne signifie pas que cet index doive nécessairement être écarté. D'une part, en conjonction avec d'autres conditions évaluables par index il peut réduire le nombre d'accès aux pages de données, et, d'autre part, il sera intéressant pour des valeurs rares de LOCALITE.

20.10 QUE RETENIR ?

La conception physique est un processus complexe lié à un SGBD spécifique mais dont le principal apport est la détermination des index et de leur technologie, ainsi que des espaces de stockage.

On imposera un index pour chaque identifiant. On envisagera la création d'index pour chaque clé étrangère, pour les colonnes faisant fréquemment l'objet de conditions d'égalité ou d'intervalle. On veillera cependant à éviter les index inutiles, notamment les index qui seront dédaignés par le SGBD (qui leur préférera un accès séquentiel), les index préfixes, les index définis sur une colonne dont le domaine de valeurs est réduit et pour les tables de petite taille.

Les technologies basées sur l'organisation séquentielle indexée (*B-arbres*) favorisent les accès par intervalle, la croissance rapide du volume des données et les temps d'accès uniformes. Les technologies basées sur l'organisation calculée favorisent les temps d'accès très courts et l'économie d'espace.

En l'absence de critères clairs, on donnera la préférence aux index secondaires dont le dictionnaire est organisé en séquentiel indexé.

Les erreurs de choix d'index peuvent en général être corrigées en cours d'exploitation de la base de données, moyennant une surveillance régulière du comportement du SGBD, et en particulier des plans d'exécution des requêtes.

20.11 POUR EN SAVOIR PLUS

Les différents problèmes de la conception physique, en tant que processus intervenant dans la conception d'une base de données, n'ont pas suscité une avalanche de travaux. Parmi les ouvrages généralistes, on recommandera [Shasha, 2002], [Lightstone, 2007] et [Garcia-Molina, 2008]. Le lecteur intéressé consultera ensuite la documentation de son SGBD (*Administration guide* notamment) ainsi que les nombreuses références et les sites consacrés à l'amélioration des performances (par exemple <http://www.oaktable.net> pour Oracle). Ce chapitre évoque brièvement quelques principes de traduction de requêtes SQL en opérations physiques telles que la lecture séquentielle d'une table ou l'usage d'un index. Il s'agit d'une matière complexe relevant de la théorie des plans d'exécution optimisés. Les ouvrages précités abordent cette question en détail.

20.12 EXERCICES

20.1 Compléter le schéma logique de la figure 19.35 en y ajoutant les index induits par les structures logiques.

20.2 Une table LIVRAISON représente les livraisons d'articles dans une société de distribution. Elle dispose d'un identifiant composé des colonnes (DATE, REF_CLI). La table comprend actuellement 4 000 000 lignes d'une longueur fixe de 400 octets. Il y a en moyenne 1.000 livraisons par jour. Les colonnes de l'identifiant ont des longueurs respectives de 12 et 28 caractères. La table est stockée dans un espace qui lui est réservé, implanté sur un disque du modèle de référence, décomposé en pages de 4 Ko. On retient les deux requêtes typiques, d et c étant des constantes quelconques :

```
select * from LIVRAISON where DATE = 'd' and REF_CLI = 'c'
select * from LIVRAISON where DATE = 'd'
```

Proposer un jeu d'index favorisant ces deux requêtes. Pour chaque index, proposer une technique d'implémentation. Calculer l'espace occupé par la table et ses index. Calculer le temps moyen d'exécution de ces deux requêtes.

Exercices résolus

- **Exercice 20.1.** Voir figure 20.10.
- **Exercice 20.2.** Les requêtes suggèrent deux index, l'un, I1, sur (DATE + REF_CLI) et le second, I2, sur DATE. Les paramètres sont **Nr** = 4 000 000, **Lp** = 4 096, **Lr** = 400. Pour l'index I1 et **Lk1** = 12 + 28 = 40. Pour l'index I2, **Nr_{pv}** = 1 000, **Nv** = **Nr**/**Nr_{pv}** = 4 000 et **Lk2** = 12.

I2 est composé d'un préfixe de I1. On peut l'ignorer si I1 est implémenté par un index trié : index primaire en séquentiel indexé ou index secondaire avec dictionnaire en séquentiel indexé. On choisit la seconde technique pour I1 et on écarte I2.

Par précaution, on vérifie que l'utilisation de l'index lors de l'accès selon DATE est utile. On utilise la feuille de calcul Scan ou index.xls, qui indique que l'accès séquentiel coûte 92 s et l'accès par index 12,3 s (hors coût index).

Volume de la table. Avec un taux de remplissage typique de 0,8, on obtient **Nrpp** = 8 et **Np** = 500 000 pages.

Calcul de l'index secondaire. Le dictionnaire de valeurs contient 4 000 000 entrées. Il correspond à un fichier séquentiel indexé dont **Nr** = 4 000 000, **Lr** = 12 + 28 + 4 = 44, **Lk** = 12 + 28 = 40, $\tau = 0,8$ et **Lp** = 4 096. La feuille de calcul ISAM.xls nous fournit les résultats suivants : **Npb** = 53 764, **Npi** = 734, **n** = 3. Les niveaux d'index occupent respectivement 1, 10 et 723 pages.

Taille de la table. Table + index occupent 500 000 + 53 764 + 734 = 554 498 pages.

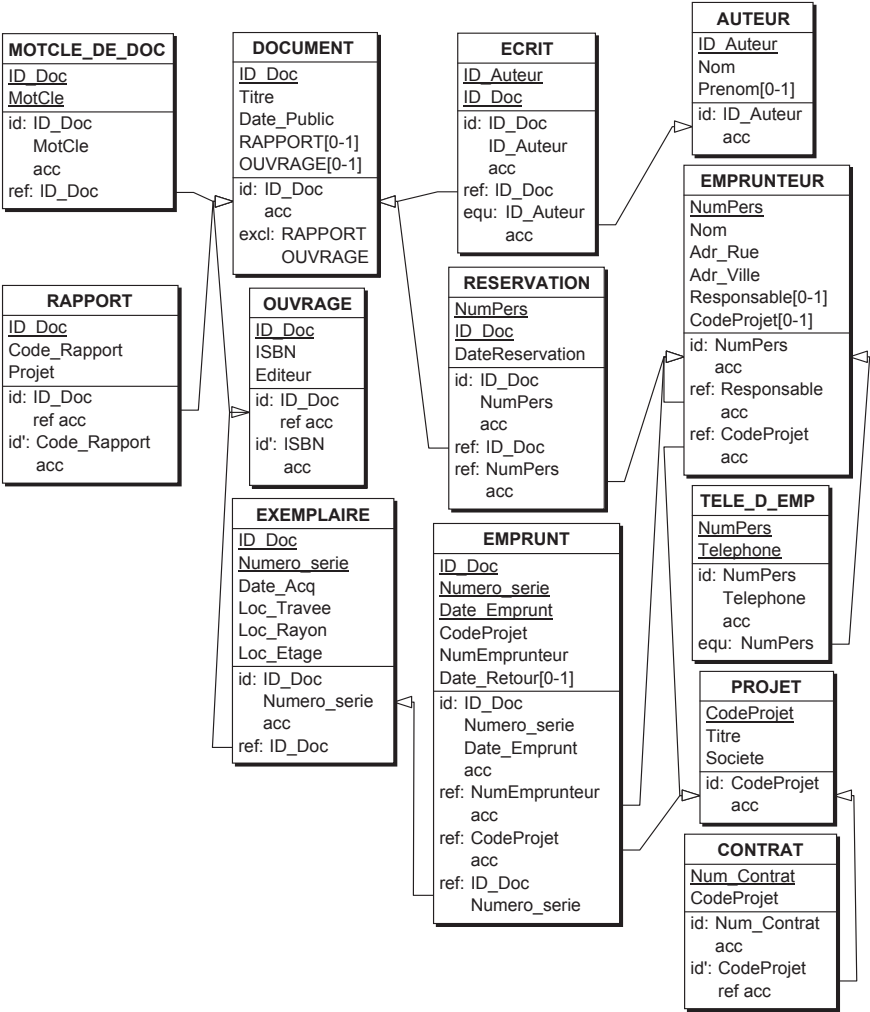


Figure 20.10 - Index du schéma logique de la figure 18.33

Temps d'accès par I1. Avec un tampon de $1 + 10 + 1 = 12$ pages, on compte $2 + 1 = 3$ accès aléatoires, soit 36,9 ms. Avec un tampon de $1 + 10 + 723 + 1 = 735$ pages, il faut 2 accès aléatoires, soit 24,6 ms.

Temps d'accès par I2. On suppose que **la table est en vrac**. Sachant que **Nrpp** = 74,4 pour le dictionnaire (toujours selon ISAM), les **Nc** = 1 000 entrées de même valeur de DATE occupent $1\,000/74,4 = 14$ pages consécutives du dictionnaire, pour un coût (négligeable) de $14 \times 0,184 = 2,6$ ms. Viennent s'ajouter les accès aléatoires à 1 000 pages de la table, soit 12,3 s. Total : 12,3 secondes.

Temps d'accès par I2. On suppose à présent que l'index I1 est **du type clustering**. Les lignes de même valeur de DATE sont consécutives dans les pages de la table. Elles occupent $1\,000 / 8 = 125$ pages consécutives. Le coût d'accès est de $125 \times 0,184 = 23$ ms. Le coût total, index + pages de base, est de $2,6 + 23 = 25,6$ ms. Cette organisation est donc, du moins théoriquement, $12,3 / 0,023 = 535$ fois plus rapide que la précédente pour l'évaluation de la requête I2.

Des compléments et des exercices supplémentaires le plus souvent accompagnés d'une suggestion de solution sont disponibles à l'annexe A20.

Chapitre 21

Production du code d'une base de données

Le schéma résultant de la conception logique puis de la conception physique, ainsi que les vues, sont ensuite traduits dans le code SQL DDL du SGBD. Certaines contraintes doivent être codées de manière spécifique, à l'aide de prédicats (check), de déclencheurs (triggers), de procédures SQL (stored procedures) ou de méthodes. On développe particulièrement la traduction des structures issues des relations *is-a*.

21.1 INTRODUCTION

Tel qu'il a été présenté au chapitre 13, le processus de rédaction des instructions SQL DDL relatives à un schéma de tables est en principe relativement direct, d'autant plus qu'on dispose aujourd'hui de générateurs de code DDL qui produisent automatiquement ces instructions à partir du schéma physique.

Le problème n'est malheureusement pas toujours aussi simple. En effet, les SGBD ne peuvent prendre en charge qu'un nombre réduit de structures et de contraintes d'intégrité. Ainsi, les SGBD relationnels ne gèrent que les identifiants, les colonnes obligatoires et les clés étrangères. Le problème se pose donc de traduire les autres contraintes (dites *additionnelles*), de manière à garantir l'intégrité des données telle qu'elle a été définie dans le schéma conceptuel, et qui, si tout s'est passé correctement, est intégralement exprimée dans le schéma physique. En outre, les structures Entité-association avancées ont introduit dans les schémas logique et physique de nouvelles contraintes que le langage SQL ne peut exprimer de manière explicite. Pensons par exemple à la contrainte *equ*, aux cardinalités spéciales ainsi

qu'aux contraintes issues des relations *is-a*. Il faudra donc recourir pour traduire ces contraintes à des mécanismes tels que les prédicats, les déclencheurs, les procédures SQL et les méthodes de table typée. Puisque nous allons programmer nous-mêmes le code de validation de ces contraintes, nous devons à l'occasion décider de la réaction à adopter en cas de tentative de violation.

La figure 21.1 donne la description externe du processus de codage. Le **code DDL** représente l'expression explicite des concepts du schéma physique que nous avons évoqués au chapitre 20, ainsi que celle des vues. Le **code additionnel** exprime les contraintes additionnelles. C'est surtout sur ce dernier que nous allons concentrer nos efforts. Sauf mention contraire, nous n'utiliserons pas les constructions *relationnelles objet* de SQL3.

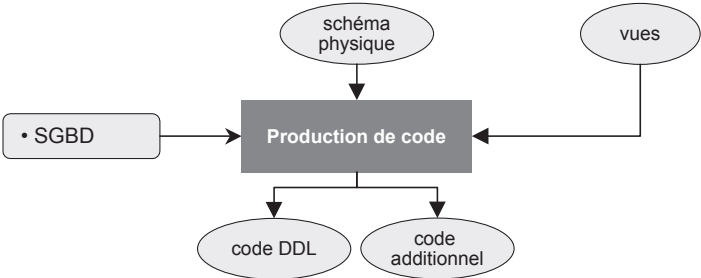


Figure 21.1 - Le processus de codage du schéma de la base de données

21.2 EXPRESSION DÉCLARATIVE DES STRUCTURES

Les constructions de base du schéma seront déclarées explicitement sous la forme de scripts notés **code DDL**. Les seules exceptions admissibles sont celles des comportements non standard lors des mises à jour, notamment en ce qui concerne les clés étrangères. Ce point sera discuté à la section 21.4.

objet	code et commentaires
schéma	<i>note</i> : pas de standardisation, spécifique à chaque SGBD create database <nom schéma>; create schema <nom schéma>;
table	create table <nom table> (<colonnes> <contraintes>) in ...;
colonne facultative	<nom colonne> <type colonne>
colonne obligatoire	<nom colonne> <type colonne> not null
identifiant primaire	primary key (<composants>)
identifiant secondaire	unique (<composants>)
clé étrangère	foreign key (<composants>) references <nom table>
index	<i>note</i> : diverses extensions selon le SGBD, notamment pour les index primaires ; ne pas déclarer si création automatique pour les identifiants create index <nom index> on <nom table> (<composants>)

espace de stockage	<code>create tablespace; create dbspace; etc.;</code> <i>note : pas de standardisation, spécifique à chaque SGBD</i>

SQL admet diverses variantes *stylistiques*, notamment en ce qui concerne la déclaration des contraintes. Celles-ci peuvent être déclarées comme *contraintes de colonne*, comme *contraintes de table* ou comme *objets autonomes*. La dernière option est sans doute la plus souple, au détriment de la concision. Par exemple, la déclaration d'un identifiant primaire se présenterait comme suit¹ :

- contrainte de colonne : `NCLI char(10) not null`
`[constraint CLI_PK] primary key,`
- contrainte de table : `[constraint CLI_PK] primary key (NCLI),`
- objet autonome : `alter table CLIENT add constraint CLI_PK`
`primary key (NCLI);`

21.3 LES CONCEPTS SQL D'EXPRESSION DE CONTRAINTES

Le langage SQL DDL dispose de plusieurs mécanismes permettant d'exprimer les contraintes d'intégrité d'un schéma logique. Nous les avons étudiés au chapitre 9. On en rappelle les principes dans la perspective du codage d'un schéma.

• **Les déclarations natives**

Les identifiants primaires et secondaires, les colonnes obligatoires et les contraintes référentielles sont explicitement exprimables en DDL, selon les règles 21.2. Des comportements standard en cas de tentative de violation de contraintes référentielles s'expriment via les clauses `on delete` et `on update` (chapitre 6).

• **Les prédicats de colonne et de table (*check*)**

Un prédicat permet d'exprimer une propriété simple de la valeur d'une colonne d'une ligne (prédicat de colonne) ou une relation entre les valeurs de colonnes d'une ligne (prédicat de table). Des variantes plus complexes faisant référence à d'autres lignes de la table ou même à d'autres tables sont possibles mais ne sont pas disponibles dans tous les SGBD. On recommandera de les éviter pour des raison de portabilité² (section 9.7).

• **Les vues filtrantes (*view with check option*)**

Ce mécanisme est similaire aux prédicats. La clause `with check option` interdit aux utilisateurs d'une vue d'appliquer une modification dont le résultat ne respecterait pas la clause `where` de la définition de la vue (section 9.3). Pour garantir le contrôle de l'intégrité, on force les utilisateurs à réaliser toutes les modifications via la vue. On suggère donc de retirer l'autorisation de modifier la table de base (`revoke`) puis d'accorder l'autorisation de le faire via la vue (`grant`). Il est à noter

1. Une clause [entre crochets] est optionnelle.
2. La **portabilité** d'un composant est son aptitude à être installé sans effort sur une autre plateforme matérielle/logicielle que celle pour laquelle il a été développé. Par exemple, un script DDL généré pour un SGBD est portable s'il est exécutable sans modification sur d'autres SGBD.

que la vue filtrante est plus puissante que le prédicat de table dans sa version limitée (imposée par Oracle et DB2 par exemple). En particulier, la clause `where` peut citer d'autres tables. En Oracle, la vue suivante est valide et **modifiable** :

```
create view FACTURE_CORRECTE as
  select * from FACTURE
  where MODE in (select MODE_PAIEMENT from MP)
  with check option;
```

• Les déclencheurs (*trigger*)

Le déclencheur est le mécanisme idéal pour gérer les contraintes dynamiques, son corps ayant accès aux états de la ligne courante avant (*old*) et après (*new*) la modification qui l'a déclenché (section 9.9). On l'utilisera également pour la prise en charge des contraintes statiques complexes. Son usage est plus délicat que celui des prédicats. En particulier, une contrainte se traduit souvent par plusieurs déclencheurs. On suggère la procédure suivante pour traduire une contrainte **C**, qu'on suppose exprimée sous une forme raisonnablement précise (section 16.12.1, adaptée aux structures relationnelles).

- *Analyse de C*. On repère dans l'expression de **C** l'ensemble **O** des objets du schéma qui y sont cités directement ou implicitement.
- *Analyse des éléments de O*. Pour chaque objet de **O**, on identifie les opérations (*insert*, *delete*, *update*) susceptibles d'affecter la satisfaction de la contrainte **C**.
- *Définition des réactions*. Pour chacune de ces opérations, on définit une réaction appropriée en cas de violation : annulation, propagation, correction, recalcul, etc.
- *Codage des réactions*. Pour chaque opération (ou événement), on écrit le code qui traduit la réaction, souvent sous la forme d'un déclencheur. On regroupe éventuellement les déclencheurs similaires sur une même table. On donne au déclencheur la forme générale :

```
before <opérations> on T
for each row
begin
  if ¬C then abort(); endif;
end;
```

• Les procédures SQL

Une procédure SQL (section 9.8) sera utilisée pour remplacer l'opérateur de modification natif d'SQL. Au lieu de demander l'exécution d'une requête SQL de modification (*insert into CLIENT*), l'utilisateur (un programme généralement) invoque la procédure qui la simule (*call inserer_CLIENT(CLI_Record, Status)*). Le corps de la procédure assure la gestion de l'ensemble des contraintes d'intégrité portant sur la table cible. Cette technique est plus facile à appliquer que celle des déclencheurs. Elle offre aussi l'avantage de rendre les programmes d'application plus portables vis-à-vis du SGBD, le langage SQL étant moins (voire plus du tout) utilisé. En revanche, elle impose au programmeur un mode d'interaction spécifique avec la base de données ; en outre les procédures doivent être réécrites pour chaque SGBD.

- **Autres techniques**

À ces techniques exploitant les fonctions des SGBD s'ajoute une gamme de méthodes et architectures concourant, avec plus ou moins de bonheur, au même but : gérer l'intégrité des données. Mentionnons d'abord trois extensions de la technique des procédures : les modules d'accès, les méthodes et les O/RM.

- *Modules d'accès.* Le programmeur développe, dans son langage d'application (Java par exemple), un ensemble de procédures et fonctions dont le but est de réaliser les principales opérations de modification de la base de données, en gérant au passage les contraintes d'intégrité. Cette technique très ancienne a été largement utilisée pour encapsuler les opérations sur les fichiers mais est parfaitement adaptée aux opérations sur les bases de données. La modification directe des données via les instructions SQL doit être interdite.
- *Méthodes.* Le modèle relationnel objet des SGBD modernes permet de développer des méthodes généralistes ou spécifiques attachées aux types de tables. On peut développer des constructeurs, des modificateurs et des destructeurs qui encapsulent la gestion des contraintes d'intégrité. Les avantages et inconvénients sont analogues à ceux des procédures SQL.
- *Interfaces Object/Relational Mapping ou O/RM.* Un O/RM est un logiciel s'interposant entre le programme d'application et le SGBD, et permettant de construire des classes offrant de la base de données une vue orientée objet. La correspondance entre ces classes et la base de données est assurée (semi-)automatiquement selon des règles de correspondance prédéfinies³. Une tendance croissante chez les programmeurs consiste à analyser les besoins d'une application en termes de classes, puis de générer automatiquement le schéma de la base de données à partir de ces classes. Il peut en résulter un schéma de piètre qualité, difficile à maintenir, peu évolutif et mal adapté aux besoins d'autres applications.

Parmi les autres techniques, mentionnons encore les suivantes : sections de code de validation distribuées dans les programmes d'application (à déconseiller de manière absolue), procédures de validation attachées aux écrans de saisie (à déconseiller), programme de validation des données sources avant chargement, programme de validation des données après chargement (applicables surtout pour les entrepôts de données).

21.4 LES COMPORTEMENTS NON STANDARD

Lors d'une demande de modification, le SGBD adopte un comportement standard, éventuellement particularisé par une directive SQL. Tel est le cas des clés étran-

3. On consultera en particulier les sources suivantes : [Ambler, 2006b], http://en.wikipedia.org/wiki/Object-relational_mapping, http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software (juin 2015)

gères, dont le comportement lors d'une modification des données peut être paramétré par les clauses `on delete` et `on update`. D'une part, certains SGBD n'offrent pas certains comportements et d'autre part des comportements non standard sont désirables dans certains cas. On remplacera ou modifiera alors les règles du SGBD par des déclencheurs ou des procédures. Nous décrirons deux exemples liés à la gestion des clés étrangères.

1. Lors de l'insertion d'une ligne COM de COMMANDE, le SGBD vérifie l'existence d'une ligne CLI de CLIENT telle que `CLI.NCLI = COM.NCLI`. En cas d'échec, l'insertion est rejetée. À ce comportement standard, on voudrait en substituer un autre, plus constructif, selon lequel la ligne de CLIENT manquante est créée avant l'opération demandée, laquelle sera dès lors exécutée avec succès. C'est ce que code le déclencheur suivant⁴ :

```
create trigger TRG_COMMANDE_CLI_MANQUANT
before insert on COMMANDE
for each row
begin
    declare I number
    select count(*) into :I from CLIENT where NCLI = new.NCLI;
    if (I = 0) then
        insert into CLIENT (NCLI,NOM,ADRESSE,LOCALITE,COMPTE)
        values (new.NCLI,'?','?','?',0);
    endif;
end;
```

2. Si on n'admet pas de commande sans détails, alors il conviendrait, soit d'empêcher la suppression de la dernière ligne de DETAIL associée à une ligne de COMMANDE (mode *bloquant*), soit de supprimer la ligne de COMMANDE pour laquelle on vient de supprimer la dernière ligne de DETAIL associée (mode *propagation*). Le déclencheur suivant code ce dernier comportement.

```
create trigger TRG_DETAIL_SUPP_DERNIER
after delete on DETAIL
for each row
begin
    declare I number
    select count(*) into :I
    from DETAIL where NCOM = old.NCOM;
    if (I = 0) then
        delete from COMMANDE where NCOM = old.NCOM;
    endif;
end;
```

4. Les déclencheurs et procédures de ce chapitre sont rédigés dans une syntaxe fictive proche de celle de PL/SQL d'Oracle ou du langage d'InterBase.

21.5 LES CONTRAINTES D'INTÉGRITÉ DE BASE

Nous étudierons d'abord la traduction des contraintes élémentaires présentées à la section 16.11. Nous ignorerons dans un premier temps la traduction des dépendances fonctionnelles, le schéma logique étant normalisé par construction.

a) Contraintes de domaine et contraintes entre valeurs

Une contrainte de domaine ou une contrainte entre valeurs d'une même ligne se codera idéalement par un prédicat :

```
alter table EMPLOYE add constraint CHECK_DOM_EMP_SEXE
check (upper (SEXE) in ('M', 'F')) ;
```

Lorsque le domaine de valeurs de la colonne comporte un grand nombre de valeurs et lorsque cet ensemble est évolutif, alors on envisagera la création d'une table reprenant les valeurs du domaine (éventuellement accompagnées de leur signification) et la définition d'une clé étrangère sur cette colonne (section 19.3.5).

b) Contraintes d'existence

Le tableau de la section 16.11.3 nous fournit les règles d'interprétation des contraintes binaires faciles à coder en SQL. Leur extension à plus de deux colonnes est immédiate, comme le montre le tableau ci-dessous. On note **A** la propriété "A is not null" et $\neg \mathbf{A}$ la propriété "A is null".⁵

contrainte	prédicat
coex: A, B	$(A \wedge B) \vee (\neg A \wedge \neg B)$
coex: A, B, C	$(A \wedge B \wedge C) \vee (\neg A \wedge \neg B \wedge \neg C)$
excl: A, B	$(\neg B) \vee (\neg A)$
excl: A, B, C	$(\neg B \wedge \neg C) \vee (\neg A \wedge \neg C) \vee (\neg A \wedge \neg B)$
at-lst-1: A, B	$A \vee B$
at-lst-1: A, B, C	$A \vee B \vee C$
exact-1: A, B	$(A \wedge \neg B) \vee (\neg A \wedge B)$
exact-1: A, B, C	$(A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge C)$
if: A, B	$B \vee \neg A$
if: A, B, C	$(B \wedge C) \vee \neg A$

L'expression de ces prédicats sous forme SQL s'en déduit sans difficulté. Par exemple, la contrainte de coexistence coex: A, B, C s'écrit comme suit :

5. Portant sur n colonnes, une contrainte coex comporte 2^n conditions élémentaires, excl en comporte $n(n-1)$, at-lst-1 en comporte n, exact-1 en comporte n^2 et if en comporte n. Il est possible de réduire la complexité des contraintes excl et exact-1 comme suit, en ne calculant chaque condition élémentaire qu'une seule fois. On associe à chaque colonne C de la contrainte la valeur 0 si C is null et 1 dans le cas contraire. On calcule alors la somme S de ces valeurs pour la contrainte. excl est vraie si $S \leq 1$ et exact-1 est vraie si $S = 1$. La conversion logique/numérique s'obtient par la fonction (case when C is null then 0 else 1).

```
alter table T add constraint CHECK_T_COEX_ABC
check( (A is not null and B is not null and C is not null)
or (A is null and B is null and C is null) );
```

21.6 LES CONTRAINTES D'INTÉGRITÉ ADDITIONNELLES

Ces contraintes, décrites à la section 16.12, peuvent être plus complexes à coder. Nous analyserons les *contraintes cycliques*, les propriétés du *graphe des instances* et les *contraintes de redondance*.

a) Contraintes cycliques

Au niveau conceptuel, ces contraintes consistent en une comparaison de deux ensembles d'entités, qui se traduit par une comparaison de deux ensembles de valeurs d'identifiants de lignes. Pour deux ensembles I1 et I2 de ces valeurs, cette comparaison est l'une des suivantes : $I1 = I2$, $I1 \subseteq I2$ et $I1 \cap I2 = \emptyset$. Le prédicat et le déclencheur conviennent pour ce type de contrainte, selon la localité des données (intra-ligne ou inter-lignes) intervenant dans la comparaison. Examinons par exemple la traduction de la contrainte suivante (figure 12.27 et section 16.12.3) qui indique que les stations de départ et d'arrivée d'une SECTION sont distinctes :

SECTION.départ.STATION $\neq$ SECTION.arrivée.STATION

Ici, un simple prédicat suffit, compte tenu de la traduction relationnelle des deux types d'associations départ et arrivée en clés étrangères (DEPART) et (ARRIVEE) :

```
alter table SECTION add constraint CHECK_SECTION_STATIONS
check(DEPART <> ARRIVEE);
```

Considérons la figure 14.9, pour laquelle on admet la contrainte qui impose que *le client qui a signé le contrat qui couvre un véhicule est le client à qui appartient ce véhicule* :

VEHICULE.couvre.CONTRAT.signe.CLIENT = VEHICULE.appartient.CLIENT

Cette contrainte, en apparence complexe, va se traiter d'une manière un peu particulière. En effet, on observe dans la table VEHICULE la présence de deux colonnes référençant une ligne de CLIENT, l'une, NUMCLIENT, de manière directe, et l'autre, SIGNATAIRE, transitivement. Il suffit donc d'imposer l'égalité de ces deux valeurs. Inutile cependant de définir un prédicat, il suffit de supprimer la colonne SIGNATAIRE, et de la remplacer par NUMCLIENT dans la clé étrangère vers CONTRAT.

La gestion de certaines contraintes cycliques exigera l'intervention de valeurs issues de plusieurs tables et donc la traduction sous la forme d'un déclencheur. Tel sera le cas de la contrainte additionnelle du schéma 19.35, qui impose qu'*un exemplaire ne peut être emprunté par un emprunteur si celui-ci le réserve* :

```
pour tout EMPRUNT E,
il n'existe pas (RESERVATION R
where E.ID-Doc = R.ID-Doc and E.NumEmprunteur = R.NumPers)
```


On peut traduire cette contrainte par un déclencheur sur la table EMPRUNT :

```
create trigger TRG_EMPRUNT_NON_RESERVE
before insert on EMPRUNT
for each row
declare I number
begin
  select count(*) into :I from RESERVATION R
  where new.ID_Doc = R.ID_Doc
  and new.NumEmprunteur = R.NumPers;
  if (I > 0) then abort(); endif;
end;
```

Cependant, d'autres déclencheurs sont nécessaires. En effet, si nous suivons les éléments méthodologiques décrits en 21.3, il apparaît que d'autres modifications sont susceptibles de perturber cette contrainte et doivent être prises en compte pour le maintien de l'intégrité ; pour chacune, la réaction en cas de violation sera l'annulation de la modification. On examinera en particulier :

- update OUVRAGE set ID-Doc
- update EXEMPLAIRE set ID-Doc
- insert into RESERVATION
- update RESERVATION set ID-Doc
- update RESERVATION set NumPers

b) Contraintes sur le graphe des instances

Nous prendrons comme exemple la contrainte de la section 16.12 qui exprime le fait qu'*une personne ne peut, directement ou indirectement, être son propre responsable*. La contrainte ne peut être violée que par la **modification** de la colonne RESPONSABLE d'une ligne existante. Nous proposerons une protection par déclencheur qui vérifie, avant modification d'une ligne, que celle-ci est absente de l'ensemble des lignes obtenues en suivant les liens RESPONSABLE successifs, partant de new.RESPONSABLE. Le déclencheur ressemblerait à ceci :

```
create trigger TRG_SUPERVISE_ACYCL
before update of RESPONSABLE on PERSONNE
for each row
begin
  declare S char(4)
  S := new.RESPONSABLE;
  while (S is not null)
    if (S = new.NPERS) then abort(); endif;
    select RESPONSABLE into :S from PERSONNE where NPERS = :S;
  endwhile;
end;
```

c) Contraintes de redondance

C'est de toute évidence par un jeu de déclencheurs qu'on gèrera une redondance. Montrons-le sur l'exemple du schéma de la figure 21.2. La traduction relationnelle

sous forme des tables CLIENT et COMMANDE, évidente, n'est pas illustrée. La table CLIENT comporte une colonne TOTAL dont il faut garantir, pour toute ligne CLI de CLIENT, que la valeur est égale à la somme des valeurs de Montant des lignes de COMMANDE dont NUMCLI = CLI.NUMCLI.

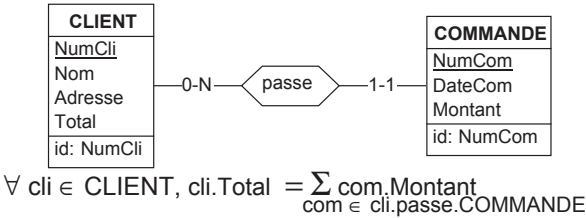


Figure 21.2 - Total est un attribut redondant

Soit CLI une ligne de CLIENT et COM toute ligne de COMMANDE telle que COM.NUMCLI = CLI.NUMCLI. La redondance s'exprime symboliquement par la relation suivante, qui doit être vérifiée à tout instant :

CLI.TOTAL = $\sum_{COM} COM.MONTANT$

Les événements susceptibles d'entraîner une violation de cette relation, et les réactions appropriées, sont les suivants.

Événement	Réaction et codage
insert into CLIENT	mise à 0 de TOTAL ; <i>codage</i> : par trigger ; "default 0" insuffisant (l'insert pourrait spécifier une valeur ≠ 0)
update CLIENT set TOTAL	opération interdite; <i>codage</i> : "revoke update(TOTAL) on CLIENT from public" ou trigger annulant l'opération
update CLIENT set NUMCLI	propager la modification vers la clé étrangère ; <i>codage</i> : clé étrangère avec "on update cascade"
insert into COMMANDE	ajouter à TOTAL la valeur de MONTANT ; <i>codage</i> : trigger TRG_COMMANDE_INSERT
delete from COMMANDE	retirer de TOTAL la valeur de MONTANT ; <i>codage</i> : trigger TRG_COMMANDE_DELETE
update COMMANDE set MONTANT	ajouter à TOTAL la différence des valeurs nouvelle et ancienne de MONTANT ; <i>codage</i> : trigger TRG_COMMANDE_UPDATE_MONTANT
update COMMANDE set NUMCLI	retirer de TOTAL de l'ancien client la valeur de MONTANT ; ajouter à TOTAL du nouveau client la valeur de MONTANT ; <i>codage</i> : trigger TRG_COMMANDE_UPDATE_NUMCLI

À titre d'exemple, le déclencheur TRG_COMMANDE_UPDATE_NUMCLI pourrait se coder comme suit :

```
create trigger TRG_COMMANDE_UPDATE_NUMCLI
after update of NUMCLI on COMMANDE
for each row
begin
  update CLIENT set TOTAL = TOTAL - old.MONTANT
  where NUMCLI = old.NUMCLI;
```

```
update CLIENT set TOTAL = TOTAL + new.MONTANT
where NUMCLI = new.NUMCLI;
end;
```

21.7 LES CONTRAINTES DÉRIVÉES DU MODÈLE ENTITÉ-ASSOCIATION

La traduction relationnelle des constructions avancées du modèle Entité-association introduit des contraintes nouvelles. Nous étudierons les techniques de codage de deux d’entre elles.

a) Contraintes référentielles totales (equ)

Nous avons analysé brièvement les propriétés dynamiques de la clé étrangère totale (equ) à la section 19.7.6. Nous avons conclu à l’impossibilité de contrôler ces propriétés par un jeu de déclencheurs attachés aux tables de base. Reprenons la figure 21.3 qui nous avait servi de support de raisonnement.

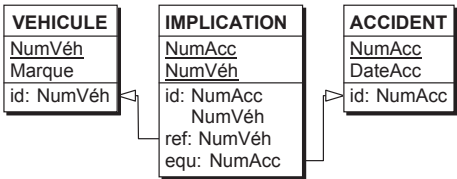


Figure 21.3 - Une clé étrangère totale

Pour décider d’un codage adéquat, nous devons distinguer le comportement de la base de données pour chaque type de modification. Nous n’étudierons pas l’update, dont l’analyse est laissée à l’initiative du lecteur.

Événement	Réaction et codage
insert into ACCIDENT	au moins une ligne d’IMPLICATION doit être insérée simultanément ; <i>codage</i> : transaction ou procédure SQL
insert into IMPLICATION	si la ligne d’ACCIDENT n’existe pas, elle est insérée simultanément ; <i>codage</i> : transaction ou procédure SQL
	si la ligne d’ACCIDENT existe déjà, comportement standard
delete from ACCIDENT	blocage ou propagation ; <i>codage</i> : "on delete no action" ou "on delete cascade"
delete from IMPLICATION	si cette ligne n’est pas la dernière de sa ligne d’ACCIDENT, comportement standard
	si cette ligne est la dernière de sa ligne d’ACCIDENT, soit l’opération est annulée, soit cette ligne d’ACCIDENT est aussi supprimée ; <i>codage</i> : trigger ou procédure SQL

Les solutions basées sur des transactions imposent au programmeur une discipline dont le respect absolu n’est pas garanti. Le développement de procédures SQL, ou

toute variante sous forme de **méthodes** ou de **modules d'accès**, semble plus approprié. Observons d'abord que les tables ACCIDENT et IMPLICATION forment des agrégats logiques qu'on pourrait qualifier d'*objets complexes* : une ligne d'ACCIDENT accompagnée de ses lignes d'IMPLICATION forme un objet autonome. On peut donc proposer un jeu de quatre procédures organisé autour du concept d'**accident** et dont la rédaction ne pose pas de problèmes particuliers :

- ACCIDENT_CREER : insère une ligne d'ACCIDENT et au moins une ligne d'IMPLICATION ;
- ACCIDENT_SUPPRIMER : supprime une ligne d'ACCIDENT et toutes les lignes d'IMPLICATION associées ;
- ACCIDENT_AJOUTER_VEHICULE : ajoute une ligne d'IMPLICATION à une ligne d'ACCIDENT ;
- ACCIDENT_RETIRER_VEHICULE : supprime une ligne d'IMPLICATION associée à une ligne d'ACCIDENT ; si celle-ci ne possède plus de ligne d'IMPLICATION, soit l'opération est annulée, soit la ligne d'ACCIDENT est également supprimée.

Remarquons encore que l'intégration de IMPLICATION dans ACCIDENT sous la forme d'une colonne complexe comme suggéré à la figure 20.9 peut régler ce problème si on adopte les structures de SQL3.

b) Les cardinalités spéciales

Une cardinalité maximale d'un rôle différente de 1 ou N produit lors de la conception logique une contrainte sur le nombre de lignes associées à une ligne déterminée dans une jointure. Reprenons la contrainte du schéma 19.35 selon laquelle *il n'y a pas plus de 10 instances de MOTCLE_DE_DOC pour une même valeur de ID_Doc*. Le déclencheur suivant gère cette contrainte pour les opérations insert et update ID_Doc :

```
create trigger TRG_MOTCLE_DE_DOC_MAX_10
before insert or update of ID_Doc on MOTCLE_DE_DOC
for each row
begin
    declare I number
    select COUNT(*) in :I from MOTCLE_DE_DOC
    where ID_Doc = new.ID_Doc;
    if (I = 10) then abort(); endif;
end;
```

21.8 LES CONTRAINTES D'INTÉGRITÉ DYNAMIQUES

Une contrainte d'intégrité dynamique définit les changements d'état valides. C'est encore par un jeu de déclencheurs que nous l'implémenterons. Admettons qu'on n'autorise la suppression d'un client que s'il n'a plus envoyé de commandes depuis le 1^{er} janvier 2014. Cette règle est prise en charge par le déclencheur suivant.

```
create trigger TRG_CLIENT_OBSOLETE
before delete on CLIENT
for each row
begin
  declare I number
  select count(*) into :I from COMMANDE
  where NCLI = old.NCLI and DATECOM >= '2014-01-01';
  if (I > 0) then abort(); endif;
end;
```

21.9 TRADUCTION DES RELATIONS IS-A

La traduction correcte d’une structure composée d’une table surtype et de ses tables sous-types n’est pas une tâche triviale. Nous examinerons ci-après diverses techniques d’automatisation de la gestion des données organisées selon une hiérarchie *is-a*. Le code SQL complet du schéma de la figure 21.4 et de ses mécanismes de gestion ainsi que des techniques alternatives sont proposés dans l’annexe A21.

21.9.1 Gestion à partir des tables de base

Examinons le schéma de la figure 21.4, reprise de la section 19.6.2/c, qui propose une version de schéma logique basée sur les *clés étrangères non minimales*.⁶ Il dérive d’une structure constituée du surtype DOCUMENT et des deux sous-types RAPPORT et OUVRAGE disjoints non couvrants (**D** et **¬T**)⁷.

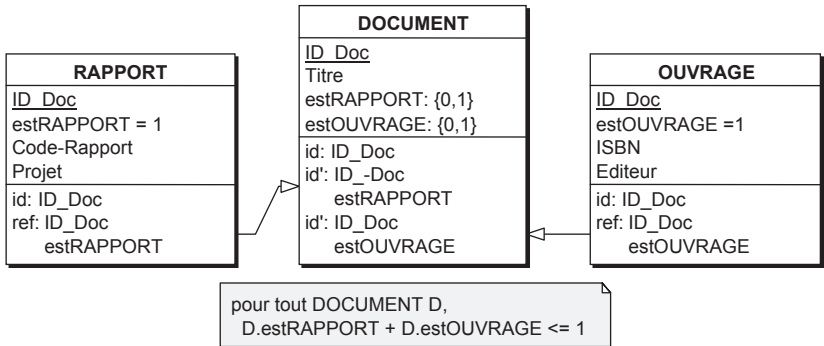


Figure 21.4 - Représentation des relations *is-a* par des *clés étrangères non minimales*. La contrainte additionnelle correspond à une *disjonction*.

6. On rappelle que ce fragment de relation *is-a* aurait pu se traduire par une hiérarchie de tables typées.
7. Rappelons que la totalité simple se traduit par la contrainte $estRAPPORT + estOUVRAGE > 0$ et la partition par la contrainte $estRAPPORT + estOUVRAGE = 1$.

Pour des raisons qui seront expliquées plus loin, nous préfixons le nom de ces tables du symbole `_`. La traduction des tables `_DOCUMENT`, `_RAPPORT` et `_OUVRAGE` (non illustrée ici) est immédiate :

```
create table _DOCUMENT(
    ID_Doc      char(12) not null primary key,
    estRAPPORT  integer  not null default 0,
    estOUVRAGE  integer  not null default 0,
    Titre       varchar(32) not null,
    unique (ID_Doc,estRAPPORT),
    unique (ID_Doc,estOUVRAGE),
    check(estRAPPORT in (0,1)),
    check(estOUVRAGE in (0,1)),
    check(estRAPPORT + estOUVRAGE <= 1) );

create table _RAPPORT(
    ID_Doc      char(12) not null primary key,
    estRAPPORT  integer  not null default 1,
    Code_Rapport integer not null,
    Projet      char(20) not null,
    foreign key (ID_Doc,estRAPPORT)
    references _DOCUMENT(ID_Doc,estRAPPORT)
        on delete cascade on update cascade,
    check(estRAPPORT = 1) );
```

On notera la traduction de la colonne constante `estRapport` dans la table sous-type `_RAPPORT` par un prédicat `check` assez *dictatorial*. Les clauses `default` permettent d'ignorer les indicateurs lors des insertions mais sont facultatives. Ce code exprime complètement le schéma logique. Il nous reste cependant deux problèmes à résoudre.

a) Suppression cohérente d'une ligne sous-type

Si l'insertion d'une ligne de `_RAPPORT` est correctement gérée, il n'en va pas de même de sa suppression. On pourrait en effet supprimer cette ligne en oubliant de mettre à 0 l'indicateur de sous-type dans la ligne `_DOCUMENT` correspondante. On règle cette question par un déclencheur :

```
create trigger TRG_RAPPORT_DELETE_ISA_DOC
after delete on _RAPPORT
for each row
begin
    update _DOCUMENT set estRAPPORT = 0
    where ID_DOC = new.ID_DOC;
end;
```

On observe que, en cas de contrainte de totalité, ce déclencheur empêchera la suppression de la dernière ligne sous-type d'une ligne de `_DOCUMENT`.

b) Gestion de la contrainte de totalité

Une contrainte de totalité (**D** ou **P**) pose encore une difficulté particulière car elle impose que l'insertion d'une ligne de `_DOCUMENT` s'accompagne simultanément de celle d'une ligne de `_RAPPORT` ou de `_OUVRAGE`, alors qu'on ne peut insérer une ligne de `_RAPPORT` que quand la ligne correspondante de `_DOCUMENT` existe déjà. Parmi les diverses possibilités de création simultanée de plusieurs lignes (transaction, procédure SQL, module d'accès externe, méthode, O/RM), nous allons étudier celle des **vues modifiables**, qui offre une solution particulièrement simple et élégante pour le programmeur d'application.

21.9.2 Gestion via des vues modifiables

La technique de représentation des hiérarchies *is-a* par matérialisation offre beaucoup de souplesse mais la répartition des données en plusieurs tables rend le schéma SQL peu naturel pour l'utilisateur (le programmeur d'application par exemple).

En définissant des **vues SQL** sur le schéma de base, on peut offrir des structures de données plus proches de la vue conceptuelle sans faire appel à des intermédiaires logiciels tels que les procédures SQL ou les O/RM, qui masquent toute la puissance du langage SQL.

a) Vues des sous-types

Pour chaque sous-type, nous créons une vue qui applique l'**héritage descendant** (vues `RAPPORT` et `OUVRAGE`, non illustrée.) On crée également une vue regroupant les lignes surtypes sans sous-type, utile en l'absence de contrainte de totalité (`DOCUMENT_SEUL`).

```
create view RAPPORT(ID_Doc,Titre,Code_Rapport,Projet) as
  select D.ID_Doc,D.Titre,R.Code_Rapport,R.Projet
  from   _DOCUMENT D,_RAPPORT R
  where  D.ID_Doc = R.ID_Doc;
```

```
create view DOCUMENT_SEUL(ID_Doc,Titre) as
  select ID_Doc,Titre
  from   _DOCUMENT
  where  estRAPPORT = 0 and estOUVRAGE = 0;
```

b) Vue du surtype

Le programmeur pourrait aussi être intéressé par une vue globale de tous les documents, obtenue par **héritage ascendant**. Cette vue est définie par des jointures externes des tables surtype et sous-types. On inclut les indicateurs de sous-type.

```
create view DOCUMENT(ID_Doc,estRAPPORT,estOUVRAGE,Titre,
                    Code_Rapport,Projet,ISBN,Editeur) as
  select D.ID_Doc,D.estRAPPORT,D.estOUVRAGE,D.Titre,
         R.Code_Rapport,R.Projet,O.ISBN,O.Editeur
  from   _DOCUMENT D
```

```

left outer join _RAPPORT R
    on D.ID_Doc = R.ID_Doc
left outer join _OUVRAGE O
    on D.ID_Doc = O.ID_Doc;

```

c) La question des mises à jour des données

Les vues DOCUMENT, RAPPORT et OUVRAGE peuvent faire l'objet de toutes les formes de la requête **select**. Malheureusement, la plupart des SGBD les considèrent (à tort !) *non modifiables*, interdisant ainsi les requêtes insert, delete et update. Pour les rendre modifiables, nous utiliserons des déclencheurs *instead of*, disponibles dans tous les SGBD, à l'exception (en 2015) de MySQL. Le premier déclencheur gère la **création d'un rapport** en insérant les deux lignes de données :

```

create trigger TRG_RAPPORT_INSERT
instead of insert on RAPPORT
for each row
begin
    declare I number
    select count(*) in :I from DOCUMENT
    where ID_DOC = new.ID_DOC;
    if (I = 1) then abort(); endif;

    insert into _DOCUMENT(ID_Doc,D.Titre,estRAPPORT)
    values(new.ID_Doc,new.Titre,1);
    insert into _RAPPORT(ID_Doc,Code_Rapport,Projet)
    values(new.ID_Doc,new.Code_Rapport,new.Projet);
end;

```

La **suppression d'un rapport** est gérée par un déclencheur sur delete attaché à la vue RAPPORT, supprimant les deux fragments _DOCUMENT et _RAPPORT :

```

create trigger TRG_RAPPORT_DELETE
instead of delete on RAPPORT
for each row
begin
    delete from _DOCUMENT where ID_Doc = old.ID_Doc;
end;

```

Les corps des déclencheurs de suppression de RAPPORT, OUVRAGE et DOCUMENT_SEUL sont identiques.

La **modification d'un rapport** peut se gérer par un seul ou par plusieurs déclencheurs. La première version est présentée ci-après :

```

create trigger TRG_RAPPORT_UPDATE
instead of update on RAPPORT
for each row
begin
    if (new.Code_Rapport <> old.Code_Rapport)
        update _RAPPORT set Code_Rapport = new.Code_Rapport

```



```

        where ID_Doc = old.ID_Doc;
    endif;
    if (new.Projet <> old.Projet)
        update _RAPPORT set Projet = new.Projet
        where ID_Doc = old.ID_Doc;
    endif;
    if (new.Titre <> old.Titre)
        update _DOCUMENT set Titre = new.Titre
        where ID_Doc = old.ID_Doc;
    endif;
    if (new.Doc_Id <> old.Doc_Id)
        update _DOCUMENT set ID_Doc = new.ID_Doc
        where ID_Doc = old.ID_Doc;
    endif;
end;

```

La gestion des opérations sur la table surtype DOCUMENT se fait par trois déclencheurs. Celui qui prend en charge la **création d'un document** tient compte des éventuels sous-types du document. Son code est donné ci-dessous :

```

create trigger TRG_DOCUMENT_INSERT
instead of insert on DOCUMENT
for each row
begin
    declare I number
    insert into _DOCUMENT(ID_Doc,estRAPPORT,estOUVRAGE,Titre)
    values (new.ID_Doc,new.estRAPPORT,new.estOUVRAGE,new.Titre);

    if (new.estRAPPORT = 1)
        insert into _RAPPORT(ID_Doc,Code_Rapport,Projet)
        values (new.ID_Doc,new.Code_Rapport,new.Projet);
    endif;

    if (new.estOUVRAGE = 1)
        insert into _OUVRAGE(ID_Doc,ISBN,Editeur)
        values (new.ID_Doc,new.ISBN,new.Editeur);
    endif;
end;

```

Le déclencheur gérant la **modification d'un document** est un peu plus délicat. Il doit en effet prendre en compte, non seulement la modification des colonnes d'attribut mais aussi des indicateurs de sous-type, qui peuvent spécifier la mutation du document, c'est-à-dire son changement de sous-type.

```

create trigger TRG_DOCUMENT_UPDATE
instead of update on DOCUMENT
for each row
begin
    declare I number
    if (new.Titre <> old.Titre)
        update _DOCUMENT set Titre = new.Titre
        where ID_Doc = old.ID_Doc;
    end if;
end;

```

```

endif;

if (old.estRapport = 1 and new.estRAPPORT = 0)
  delete from _RAPPORT
  where ID_Doc = old.ID_Doc;
  update DOCUMENT set estRAPPORT = 0
  where ID_Doc = old.ID_Doc;
endif;

if (old.estOUVRAGE = 1 and new.estOUVRAGE = 0)
  delete from _OUVRAGE
  where ID_Doc = old.ID_Doc;
  update DOCUMENT set estOUVRAGE = 0
  where ID_Doc = old.ID_Doc;
endif;

if (old.estRapport = 0 and new.estRAPPORT = 1)
  insert into _RAPPORT(ID_Doc,Code_Rapport,Projet)
  values(new.ID_Doc,new.Code_Rapport,new.Projet);
  update DOCUMENT set estRAPPORT = 1
  where ID_Doc = old.ID_Doc;
endif;

if (old.estOUVRAGE = 0 and new.estOUVRAGE = 1)
  insert into _OUVRAGE(ID_Doc,ISBN,Editeur)
  values(new.ID_Doc,new.ISBN,new.Editeur);
  update DOCUMENT set estOUVRAGE = 1
  where ID_Doc = old.ID_Doc;
endif;

if (new.Doc_Id <> old.Doc_Id)
  update _DOCUMENT set ID_Doc = new.ID_Doc
  where ID_Doc = old.ID_Doc;
endif;
end;

```

La suppression d'un document est identique à celle des sous-types.

21.10 TRADUCTION DES VUES

Lors de la conception logique, chaque sous-schéma conceptuel a été converti en un ensemble de tables relationnelles (section 19.9). La traduction en SQL de ces tables se fera naturellement sous la forme de vues (*views*) SQL. Cette traduction sera très simple lorsque la vue logique est un sous-ensemble du schéma logique complet. Cependant, un sous-schéma conceptuel dont l'intégration a été plus complexe va faire l'objet d'une définition plus élaborée. Tel est le cas du *schéma 3* qui a été décrit dans les sections 18.9 et 19.9, et dont la transformation relationnelle inclut une table ACHAT qui apparaît comme résultant de l'agrégation de lignes d'EXEMPLAIRE. On proposera pour ACHAT la définition suivante, qui s'appuie sur une traduction relationnelle standard du schéma conceptuel 18.70 :

```

create view v3_ACHAT (SERVICE,NUMENREG,DATE_ACHAT,QUANTITE) as
  select S.NOM,E.NUMENREG,E.DATE_ACHAT,count (*)
  from    PUBLICATION P,EXEMPLAIRE E,SERVICE S
  where   P.NUMENREG = E.NUMENREG
  and     E.SERVICE = S.NOM
  and     P.OUVRAGE is not null
  group by S.NOM,E.NUMENREG,E.DATE_ACHAT;

```

L'usage d'un O/RM, mentionné plus haut, permet également de construire des vues, mais l'usage de celles-ci est limité aux programmes d'application.

21.11 QUE RETENIR ?

Les contraintes issues des constructions standard du modèle relationnel (identifiants, colonnes obligatoires, clés étrangères) se codent à l'aide des clauses déclaratives du langage SQL DDL. Il existe cependant un grand nombre de contraintes dont la traduction doit faire appel à des prédicats (check), des déclencheurs (triggers) ou des procédures SQL (stored procedures).

Lorsque le comportement standard du SGBD en cas de violation d'une contrainte est inadéquat (intégrité référentielle par exemple), on lui substituera des règles spécifiques qui seront codées dans des déclencheurs ou des procédures SQL.

Les contraintes d'intégrité de base (domaines, inter-valeurs, existence) sont codées sous la forme de prédicats.

Selon leur complexité, les contraintes cycliques seront prises en charge par un prédicat ou un déclencheur. Les contraintes sur le graphe des instances et les redondances seront gérées par des déclencheurs.

Les contraintes *equ* et les cardinalités spéciales (0-10 par exemple) seront traduites par des déclencheurs. Il en est de même des contraintes dynamiques.

Les contraintes nées de la transformation des relations *is-a* peuvent être complexes à traduire en SQL. Leur gestion requiert à la fois des déclencheurs et un contrôle d'accès. En outre, la contrainte de totalité (T) sur les sous-types impose la construction de procédures SQL, à moins que le SGBD n'autorise soit la modification des données via les vues sur jointure, soit les triggers *instead of*.

Les vues se codent sans problèmes. On veillera en particulier à traduire les vues relatives aux schémas sources ayant fait l'objet d'une intégration conceptuelle.

Toute contrainte gérée par des déclencheurs peut aussi être prise en charge par des procédures SQL. Dans ce cas, on se prive toutefois de la puissance d'expression du langage SQL. La construction des déclencheurs en charge d'une contrainte doit faire l'objet d'une analyse très soignée afin de contrôler toutes les mises à jour susceptibles d'entraîner une violation de la contrainte.

21.12 GÉNÉRATION DU CODE À L'AIDE D'UN OUTIL

Le volume et la complexité du code résultant de la traduction de schémas logiques, même de petite taille, rendent indispensable l'utilisation d'outil de génération de

code SQL. Il n'existe, à la connaissance de l'auteur, aucun outil capable de produire du code à la fois correct et complet pour les contraintes étudiées dans ce chapitre.

L'atelier DB-MAIN offre plusieurs générateurs SQL, de styles et de SGBD cibles différents, que le lecteur explorera avec profit. Il comprend également un composant spécialisé dans la génération de code Oracle et DB2 selon différentes techniques de codage des principales contraintes (simple commentaire, traduction native, checks, views with check option, triggers, stored procedures).

Signalons en outre que le développement d'un générateur SQL spécialisé ne pose pas de difficultés particulières et constitue un exercice fort intéressant pour comprendre en profondeur les mécanismes des bases de données. DB-MAIN permet le développement rapide en Java de générateurs de code.

21.13 EXERCICES

21.1 Traduire en SQL chacun des trois schémas qui clôturent la section 3.8.5.

21.2 Le code suivant, réputé avoir été généré par l'atelier Rational Rose, a été présenté il y a quelques années dans un ouvrage sur UML comme la traduction d'une structure formée des classes A, B et C, cette dernière héritant de A et de B. Que penser de cette traduction ?

```
create table A(A_Id number(5) primary key) ;
create table B(B_Id number(5) primary key) ;
create table C(A_Id number(5) references A,
               B_Id number(5) references B,
               primary key (A_Id,B_Id)) ;
```

Exercices corrigés

- **Exercice 21.1.** On ignore les tables R1 et R2, communes aux trois schémas. Les tables sont renommées I (pour inscription) et A (pour attribution).

La peste (3FN). Un déclencheur vérifie qu'en cas d'insertion et de modification, il n'existe pas dans la table I de valeur de PROF qui serait associée à une autre valeur de MAT que celle qui est présente dans la ligne à insérer.

```
create table I(ETUD char(20) not null,
               MAT char(20) not null,
               PROF char(20) not null,
               primary key (ETUD,MAT)) ;

create trigger C_DF_PROF_MAT
before insert or update on I
for each row
begin
    declare N number
    select count(*) into :N from I
    where PROF = new.PROF and MAT <> new.MAT;
```

```

    if (N > 0) then abort(); endif;
end;
```

Le choléra (FNBC). Le travail du déclencheur est ici plus complexe. Il recherche les inscriptions existantes de l'étudiant (jointure de I et A) pour la même matière que celle du professeur de l'inscription courante. Il ne doit pas en exister. On veillera également à contrôler les autres opérations de mise à jour des données.

```

create table A (PROF char(20) not null primary key,
               MAT char(20) not null);
create table I (ETUD char(20) not null,
               PROF char(20) not null,
               primary key (ETUD, PROF),
               foreign key (PROF) references R3);

create trigger C_DF_ETUD_MAT_PROF
before insert or update on I
for each row
begin
    declare N number;
    select count(*) into N: from I, A
    where I.ETUD = new.ETUD and I.PROF = A.PROF
    and MAT in (select MAT from A
               where PROF = new.PROF);
    if (N > 0) then abort(); endif;
end;
```

La peste et le choléra (FNCE). Le codage est très simple et ne nécessite pas de déclencheur.

```

create table A (PROF char(20) not null primary key,
               MAT char(20) not null,
               unique (PROF,MAT));
create table I (ETUD char(20) not null,
               MAT char(20) not null,
               PROF char(20) not null,
               primary key (ETUD,MAT),
               foreign key (PROF,MAT)
               references A (PROF,MAT) );
```

- **Exercice 21.2.** Ce code comporte plusieurs erreurs graves. D'une part, tous les composants d'un identifiant primaire doivent être obligatoires. D'autre part, chacune des clés étrangères de C est en elle-même un identifiant.

Code corrigé :

```

create table A (A_Id number(5) not null primary key);
create table B (B_Id number(5) not null primary key);
create table C (A_Id number(5) not null references A,
               B_Id number(5) not null references B,
               primary key (A_Id),
               unique (B_Id));
```

D'autre part, si on admet que A et B héritent d'une super-classe commune (AB), hypothèse adoptée dans le présent ouvrage et justifiée par le fait que A et B partagent des instances communes, alors A.A_Id est une clé étrangère vers la table de AB, et de même pour B. Dans ce cas, A_Id = B.Id dans C (voir chapitre 19), d'où le code minimal suivant :

```
create table C(A_Id number(5) not null primary key,  
              foreign key (A_Id) references A,  
              foreign key (A_Id) references B);
```

Des compléments à ce chapitre sont disponibles à l'annexe A21.

Chapitre 22

Rétro-ingénierie d'une base de données

L'utilisation d'une base de données existante nécessite une documentation à jour des données, et notamment son schéma conceptuel. Lorsque celui-ci n'est plus disponible, on le reconstruit par rétro-ingénierie. Ce processus exploite diverses sources d'information disponibles, telles que le code DDL, les programmes d'application, les données elles-mêmes et les écrans des programmes, pour reconstruire le schéma logique et le schéma conceptuel de la base de données.

22.1 INTRODUCTION

Telle qu'elle a été présentée dans cet ouvrage tout au long des onze chapitres qui lui ont été consacrés, la construction d'une base de données est articulée autour d'une suite de documents développés de manière systématique : les schémas conceptuel, logique, physique, les vues ainsi que le code SQL.

Ces documents, riches en information, s'avéreront particulièrement précieux lorsqu'il s'agira de modifier la base de données en fonction de l'évolution des besoins des utilisateurs. De même, le développement de nouveaux programmes, l'intégration de bases de données existantes, la migration d'une base de données ancienne vers une technologie plus moderne, la publication de données sur le Web, l'extraction de données à destination d'un entrepôt de données sont autant d'exemples de processus qui réclament une description précise des données, tant de leur format que de leur signification.

Malheureusement, la réalité est souvent tout autre. Pour nombre de bases de données, on ne dispose plus d'une documentation à jour, d'autres n'en ayant d'ailleurs jamais eu ! Beaucoup de bases de données ont fait l'objet de ce qu'on appelle pudiquement une *conception empirique* : les tables et les colonnes ont été progressivement ajoutées au fur et à mesure des besoins, souvent par de multiples intervenants, sans planification ni plan d'ensemble. Le résultat fonctionne, mais plus personne ne dispose aujourd'hui d'une description précise des données et de leur signification. Dans certains cas, la seule information disponible est le code DDL de la base de données ou, ce qui est équivalent¹, le contenu des tables du catalogue.

Il est donc nécessaire de reconstruire cette documentation actuellement périmée ou manquante à partir de toutes les sources d'information susceptibles de nous en apprendre sur la nature et les propriétés des données. Cette reconstruction est appelée **rétro-ingénierie**. On pourrait la définir comme suit.

La rétro-ingénierie d'une base de données est le processus de reconstruction du schéma logique complet et du schéma conceptuel d'une base de données mal (ou non) documentée à partir de diverses sources d'informations disponibles, notamment, mais pas exclusivement, le code DDL du schéma, le code additionnel (y compris celui des prédicats, déclencheurs et procédures SQL), le contenu de la base de données, le code source des programmes d'application, l'interface de ceux-ci avec les utilisateurs, les messages échangés entre applications.

Essayons de mesurer la distance entre une base de données bien construite et une base de données qui, en quelque sorte, souffre des *outrages des ans*.

a) Une situation idéale

La figure 22.1 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.

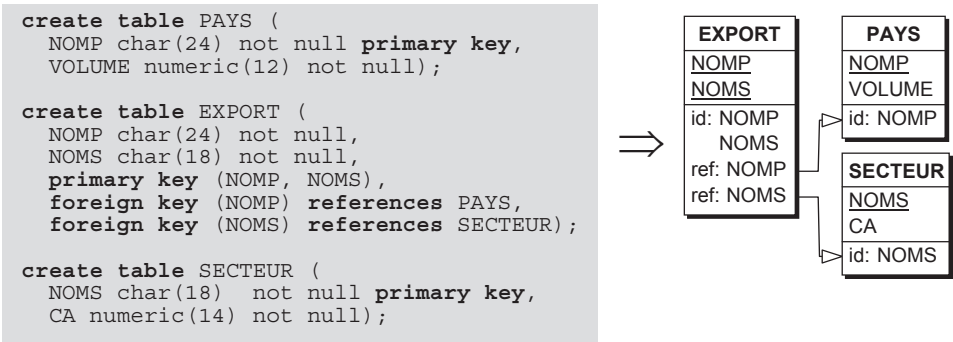


Figure 22.1 - Reconstruction du schéma de tables à partir du code SQL DDL

1. Encore faut-il disposer de la dernière version de ce code et que celle-ci soit complète. Le code DDL peut comprendre plusieurs centaines de milliers de lignes, réparties en plusieurs centaines de fichiers.

Ce schéma dérive normalement d'un schéma conceptuel. La figure 22.2 illustre le processus inverse : chaque élément du schéma de tables est issu d'un objet conceptuel qu'il s'agit de *deviner*. On peut avancer sans risque que :

- 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 *plusieurs-à-un* ou *un-à-un* ;
- à chaque *identifiant* de table correspond un identifiant de type d'entités ; un identifiant qui inclut une clé étrangère est l'image d'un identifiant hybride.

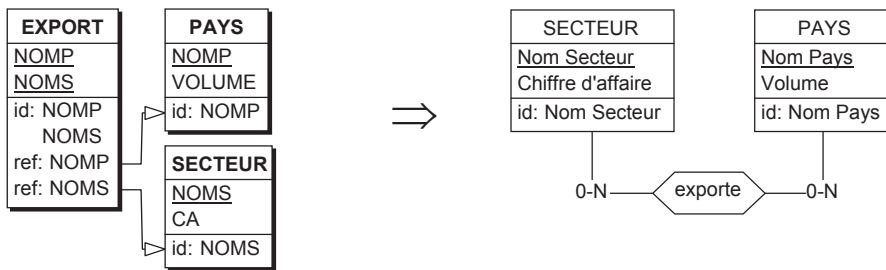


Figure 22.2 - Dérivation du schéma conceptuel à partir du schéma de tables

Le schéma résultant doit être *normalisé* au sens de la section 18.7. En particulier, un *type d'entités association*, jugé insuffisamment expressif, sera converti en type d'associations (section 18.7.2). On veillera aussi à retravailler les noms des objets de manière à les rendre plus représentatifs des concepts qu'ils désignent.

b) Une situation réaliste

Sur le terrain cependant, la rétro-ingénierie est loin de se présenter de manière aussi simpliste. Nous serons souvent confrontés à des schémas dont certaines sections sont beaucoup moins explicites que celui que nous venons d'examiner. L'histoire de la base de données, les faiblesses et lacunes des SGBD, des contraintes techniques que nous avons ignorées jusqu'ici, ainsi que certaines pratiques de développement et de programmation sont venues compliquer le processus. 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 22.3 décrit une base de données de même contenu que celle du schéma 22.1. Cependant, le schéma de tables est plus pauvre et donc 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.

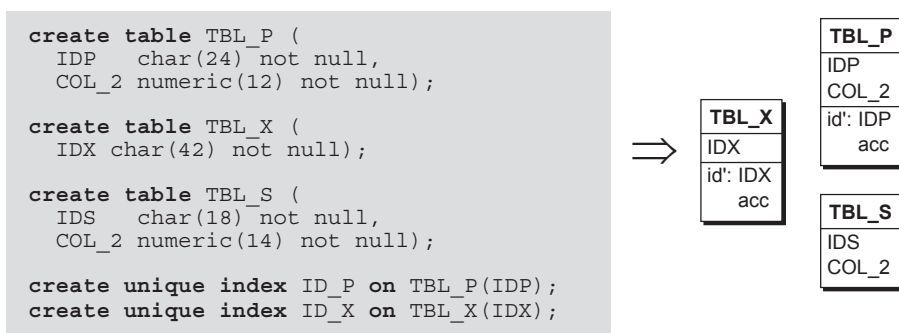


Figure 22.3 - La simple analyse du code DDL ne suffit pas toujours à retrouver le véritable schéma de tables

On peut deviner 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.

c) Au-delà du code DDL

Retrouver le véritable schéma de la base de données, c'est-à-dire celui de la figure 22.2 (gauche), est le premier objectif de la rétro-ingénierie. Le premier schéma issu de l'analyse du code DDL est progressivement enrichi des découvertes qu'on fera grâce à l'analyse des informations complémentaires : examen du code des programmes d'application, analyse des données elles-mêmes, 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 suivante devrait renvoyer un résultat vide :

```
select IDS,count(*) from TBL_S
group by IDS having count(*) > 1;
```

Pour résumer, l'analyse du code DDL nous renseigne sur les **constructions explicites** du schéma de la base de données. Cependant, le véritable schéma comprend également d'autres constructions dite **implicites**, que nous chercherons à identifier par une analyse approfondie d'autres sources d'information telles que le code additionnel, le contenu de la base de données, le code source des programmes d'application, etc.

22.2 MÉTHODE GÉNÉRALE DE RÉTRO-INGÉNIERIE

Considérée globalement, la rétro-ingénierie apparaît comme le processus inverse de la conception d'une base de données. Alors que cette dernière produit du code DDL à partir d'un schéma conceptuel, la rétro-ingénierie tente de tirer un schéma conceptuel du code SQL. Il est donc légitime d'adopter une démarche de rétro-ingénierie dont les processus sont inverses de ceux de la conception. C'est ce que suggère la figure 22.4, qui met en évidence trois processus apparaissant comme les inverses approximatifs de processus standard de la conception d'une base de données.

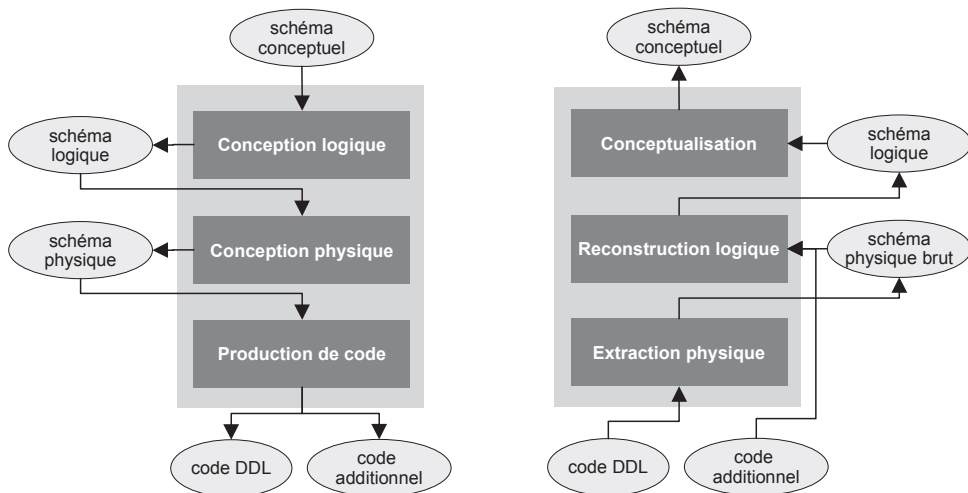


Figure 22.4 - Processus de conception et de rétro-ingénierie d'une base de données

- L'**extraction physique** produit le *schéma physique brut* (non affiné) qui a été traduit dans le code du SGBD. On note que seul le code DDL est utilisé. Ce processus est l'inverse de la production de code.
- La **reconstruction logique** tire du schéma physique un *schéma logique* qui reprend les constructions explicites présentes dans le premier mais aussi toutes les *constructions implicites* retrouvées notamment par l'analyse du code additionnel. Ce processus est l'inverse de la conception physique.

- La **conceptualisation** reconstitue un schéma conceptuel normalisé qui pourrait être celui du schéma logique. La conceptualisation apparaît comme l'inverse de la conception logique.

La figure 22.5 resitue ces processus dans le cadre général d'un projet complet de rétro-ingénierie. Il est particulièrement difficile de prévoir le coût et d'élaborer un calendrier précis pour un tel projet, de sorte qu'on réalise le plus souvent un *projet pilote*. Celui-ci consiste à redocumenter une partie réduite de la base de données. L'évaluation du temps, du coût, des difficultés et des ressources nécessaires à ce mini-projet permet d'estimer avec une certaine précision le coût et le calendrier du projet complet. Outre les processus techniques déjà mentionnés, et que nous détaillerons ci-après, on signalera celui de *Gestion des sources*, qui consiste à identifier toutes les sources d'information utiles à la rétro-ingénierie, à en évaluer la pertinence et la qualité, à sélectionner celles que nous allons utiliser et à les mettre en forme de manière à en faciliter le traitement ultérieur.

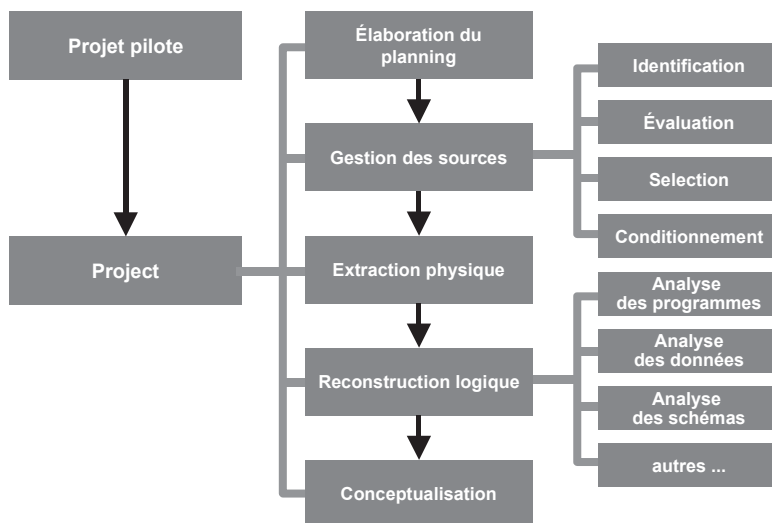


Figure 22.5 - Architecture d'un projet de rétro-ingénierie

22.3 EXTRACTION PHYSIQUE

L'extraction du schéma physique, dit *brut*, s'effectue au moyen d'un analyseur de code dédié au DDL du SGBD. Dans le cas d'une base de données relationnelle², on obtient un schéma constitué des constructions explicitement déclarées en DDL, soit des tables, des colonnes, des domaines de valeurs, des identifiants primaires et secondaires, des clés étrangères, des index, des espaces de stockages et, selon le

2. En toute généralité, les techniques décrites dans ce chapitre ne sont pas spécifiques aux bases de données relationnelles, mais sont applicables à des bases de données CODASYL, IMS, et même à de simples collections de fichiers classiques.

SGBD, des *clusters*. Il est conseillé également d'extraire les *vues*, qui nous apporteront ultérieurement des compléments d'information essentiels. L'extraction physique est un processus relativement simple mais qui ne nous fournit qu'une partie seulement des informations nécessaires pour comprendre le schéma. Le résultat est qualifié de *brut* dans la mesure où les constructions implicites, qui ont été traduites notamment dans le code additionnel, restent inconnues à ce stade.

22.4 RECONSTRUCTION DU SCHÉMA LOGIQUE

Il s'agit à présent de retrouver les constructions implicites à partir des informations fournies par les différentes sources à notre disposition, de les introduire dans le schéma physique puis de tirer de ce dernier un schéma logique complet. Nous avons vu au chapitre 22 comment les contraintes additionnelles du schéma logique peuvent se traduire en code additionnel. L'analyse de ce dernier, s'il existe, nous permettra donc de compléter notre schéma physique brut. Il existe cependant de nombreuses autres techniques permettant de vérifier la qualité des données introduites dans la base de données. Nous examinerons d'abord les principales sources d'information et leurs techniques d'analyse. Nous décrirons ensuite leur utilisation dans le but d'identifier les principales constructions implicites.

22.4.1 Les sources et leurs techniques d'analyse

Nous retiendrons cinq sources d'information majeures qui nous permettront d'identifier ou de préciser des constructions manquantes dans le schéma physique brut. Il est important de noter que les seules informations certaines sont le contenu des tables du catalogue et, s'il est à jour et complet, le code SQL DDL qui nous a permis d'extraire le schéma physique³. Toutes les autres sources peuvent apporter des indices positifs ou négatifs sur la présence d'une construction implicite mais aucune ne peut à elle seule garantir cette présence. Au final, c'est à l'analyste de décider sur la base des indices récoltés de l'existence ou l'absence d'une construction.

a) Le schéma physique

Le schéma physique lui-même peut suggérer l'existence de constructions implicites. Nous en citerons trois exemples.

- Certaines constructions physiques sont associées à des constructions logiques. Quelques exemples : un *index* défini sur une colonne non identifiante peut suggérer que celle-ci est une *clé étrangère*, un *cluster* est généralement construit pour favoriser une jointure du type *identifiant/clé étrangère*, un *index déclaré unique* définit un *identifiant*, un *identifiant technique* est généralement de type entier et apparaît souvent en tête de la liste des colonnes.

3. Le lecteur qui trouverait cette méfiance excessive doit savoir que les composants additionnels tels que les prédicats *check* ou un déclencheur peuvent être désactivés avant une mise à jour. Les contraintes qu'ils codent ne sont donc pas garanties de manière absolue.

- La formation des noms des colonnes peut suggérer le rôle de celles-ci dans le schéma. Quelques exemples : présence de fragments tels que "ID", "REF", "CODE", "OID", "CPT", "NBR" suggérant un rôle d'identifiant, de clé étrangère, de compteur. Le nom lui-même peut évoquer en clair ce rôle : DETAIL.PRODUIT, COMMANDE.CLI, CONTRAT.SIGNATAIRE, VEHICULE. PROPRIO, etc.
- Enfin, la similitude de type et de nom entre deux colonnes, dont l'une constitue un identifiant, peut suggérer que la seconde est une clé étrangère.

b) Le code SQL additionnel

Nous avons montré à suffisance dans le chapitre 20 à quel point les prédicats, les déclencheurs, les procédures SQL et les méthodes de tables typées pouvaient s'avérer riches en information concernant les contraintes additionnelles. Leur analyse sera évidemment hautement recommandée.

c) Le code source des programmes d'application

Dis-moi comment on te traite, je te dirai qui tu es ! L'examen des sections des programmes d'application qui extraient, insèrent, suppriment et modifient des lignes de tables est susceptible d'apporter des indications très précises sur les structures et les propriétés des données. Par exemple, si chaque insertion d'une ligne de COMMANDE est précédée de la recherche de la ligne de CLIENT dont NCLI = COMMANDE.NCLI, alors on peut supposer que cette recherche a pour but de vérifier l'intégrité référentielle de la clé étrangère implicite COMMANDE.NCLI. D'autre part, la requête "select ADRESSE into :ADR ..." qui assigne les valeurs de la colonne ADRESSE déclarée char(50) à la variable ADR, composée des champs (NUMERO, RUE, LOCALITE) nous permet d'affiner la structure de cette colonne.

Certaines techniques d'analyse du code source des programmes sont empruntées à une branche particulière de l'ingénierie du logiciel dénommée *compréhension de programme* (*program understanding*). D'autres techniques sont spécifiques à la rétro-ingénierie des données. Nous décrirons brièvement six d'entre elles.

- **Analyse des flux de données** (*dataflow analysis*). Le graphe des flux de données d'un programme (*dataflow diagram*) modélise la circulation des données entre, notamment, les colonnes des tables et les variables du programme. Les instructions de lecture transfèrent des valeurs de colonnes dans des variables (`select NCLI into :NC from CLIENT ...`), les instructions de traitement déplacent des valeurs entre variables (`CLI = NC;`) et les instructions d'écriture dans la base de données transfèrent des valeurs de variables dans les lignes de tables (`insert into COMMANDE values(..., :CLI, ...)`). On tire d'un tel graphe des informations importantes sur les types des colonnes et sur le rôle de celles-ci. En effet, un arc entre deux objets (constante, variable ou colonne) indique qu'à un certain moment de l'exécution, ceux-ci ont la même valeur. Ils ont donc très certainement la même structure et une sémantique commune.
- **Analyse des dépendances** (*dependency analysis*). Le graphe de dépendance entre objets d'un programme est une extension du graphe des flux. Il indique les relations de dépendance entre valeurs de variables et/ou de colonnes. Une instruction

telle que "if A = 1 then B := C" indique que la valeur de B dépend de celle de C, mais aussi de celle de A. L'usage de ce type de graphe est similaire à celui des graphes de flux.

- **Program slicing.** Cette technique très puissante s'appuie sur un jeu de graphes de dépendance du programme. Étant donné un programme **P**, un point **I** bien précis dans **P** (une instruction) et un objet **O** de **P** (une variable par exemple), le *program slicing* de **P** par rapport à (**I**, **O**) extrait un fragment **F** de **P** constitué de toutes les instructions de **P**, et elles seulement, qui contribuent directement ou indirectement à l'état de **O** au point **I**. **F** est en principe un programme exécutable dont l'effet est d'attribuer à **O** au point **I**, dans toutes les circonstances possibles, les mêmes états que le ferait l'exécution de **P** dans les mêmes circonstances. Le *program slicing* est utilisé pour réduire l'espace de recherche dans **P**. Considérons par exemple un point **I** constitué de l'instruction "insert into COMMANDE values (:M, :C, :D)" et la variable C comme objet **O**. Le fragment (**I**, **O**) contient le sous-ensemble de **P** dans lequel se trouvent toutes les instructions qui préparent la valeur de C avant son écriture dans la base de données. C'est donc parmi celles-ci qu'on trouvera les éventuelles instructions de validation de la valeur de C.
- **Recherche de patterns de programmation** (*programming clichés*). Un programmeur discipliné et systématique a tendance à utiliser des structures de programme (ou patterns) identiques pour résoudre des problèmes similaires. L'identification d'un pattern facilite le repérage de ses instances dans un programme. Par exemple, dès qu'on a identifié la manière dont un programme vérifie l'intégrité référentielle relative à une clé étrangère implicite, on peut rechercher dans le code du programme toutes les autres instances de ce pattern, lesquelles représentent probablement des indices d'autres clés étrangères.
- **Analyse statique du code SQL.** La structure d'une requête SQL observée dans le code d'un programme, tant d'extraction que de modification de données, renseigne souvent sur la structure et les propriétés des colonnes et des tables utilisées. La plupart des jointures sont basées sur l'égalité entre un identifiant d'une table et une clé étrangère de l'autre. Cette observation nous permet de repérer des clés étrangères implicites potentielles. De même, une requête qui extrait des fragments des valeurs d'une colonne suggère que celle-ci n'est pas atomique mais qu'elle est formée de composants significatifs.
L'analyse des vues SQL rentre dans cette catégorie de techniques. En effet, une vue est définie comme une requête SFW statique.
- **Analyse dynamique du code SQL.** L'analyse dynamique est très récente en rétro-ingénierie de bases de données mais particulièrement prometteuse. Elle consiste à analyser non plus le code du programme mais son comportement en cours d'exécution. Elle constitue pratiquement la seule méthode d'analyser le code SQL dynamique d'un programme (voir section 9.11). En effet, en SQL dynamique, les instructions n'existent pas dans le code mais sont *fabriquées* au moment de l'exécution. C'est donc à ce moment, via des techniques d'analyse dynamiques⁴, qu'il est possible de les observer. Il existe plusieurs techniques

permettant une analyse dynamique de requêtes SQL [Cleve, 2011]. Le SGBD lui-même maintient généralement un journal contenant toutes les requêtes SQL qu'il a exécutées. Un déclencheur peut journaliser certains éléments des requêtes de modification. L'ajout au programme d'instructions de traçage permet d'obtenir des informations sur les requêtes envoyées au serveur et sur les résultats obtenus. Cependant, la technique la plus prometteuse est celle des *aspects*. Actuellement disponible pour Java (p.ex. AspectJ)⁵ et C++ (AspectC++), cette technologie permet d'insérer dans un programme, sans modifier son code, des *déclencheurs* particuliers. En AspectJ, un aspect est spécifié notamment par les événements déclencheurs (les *pointcuts*) tels que l'invocation d'une méthode ou l'accès à une variable, et par ses réactions à ces événements (les *advice*s) sous forme d'une section de code. Elle permet de capturer les instructions SQL réellement envoyées au serveur, ainsi que toutes les invocations aux méthodes de JDBC. Elle renseigne également sur les résultats de l'exécution des requêtes.

d) L'interface programme/utilisateur

Nous avons montré à la section 18.4.2 qu'un écran, un formulaire électronique ou une page web pouvait être perçus comme des vues structurées des données de la base de données. Nous avons aussi montré qu'il était possible d'en tirer un schéma logique que nous avons ensuite transformé en un fragment de schéma conceptuel. En outre, les noms en clair apparaissant à l'écran (le libellé Date de paiement est plus significatif que le nom DATP de la colonne qui lui est associée), les recommandations d'utilisation (bouton Aide) et les messages de réaction à l'introduction de données invalides ("Numéro de client invalide - ce client n'existe pas") peuvent être très informatives. Coupler cette analyse à celle des programmes qui utilisent ces formulaires nous permet en outre de connecter les champs de ces derniers aux colonnes des tables du schéma physique et ainsi d'affiner leur définition et de mieux comprendre leur signification et les contraintes auxquelles elles sont soumises.

e) Les données

Les données sont réputées satisfaire toutes les contraintes d'intégrité qui leur ont été imposées, y compris celles que nous n'avons pas encore identifiées. L'analyse des données permet par exemple de retrouver le domaine de valeurs d'une colonne, des propriétés d'unicité, des contraintes d'inclusion (y compris référentielles) ou des colonnes constituées de la concaténation de valeurs significatives. Cette analyse s'envisagera selon deux scénarios.

- L'analyse est utilisée pour confirmer ou infirmer une hypothèse formulée sur la base des informations fournies par d'autres techniques. Il s'agit alors de vérifier si les données satisfont cette hypothèse. Si l'analyse du code source d'un

4. Attention à la confusion possible : d'une part SQL statique et SQL dynamique, et d'autre part analyse statique et analyse dynamique. On peut effectuer une analyse statique ou une analyse dynamique tant de code SQL statique que de code SQL dynamique.

5. <http://www.eclipse.org/aspectj/> et <http://en.wikipedia.org/wiki/AspectJ>

programme suggère que la colonne B2 de la table B peut être une clé étrangère vers la table A, d'identifiant A1, alors on testera, soit sur un échantillon réduit, soit sur la totalité de la table B, le degré de satisfaction de la propriété $B.B2 \subseteq A.A1$.

- L'analyse est utilisée pour rechercher un certain type de construction implicite. Dans ce cas, on procède en deux temps. On construit d'abord un ensemble d'hypothèses plausibles puis on teste chacune d'elles. La recherche de dépendances fonctionnelles de la table T comportant **N** colonnes se déroulerait comme suit : on construit une liste de tous les sous-ensembles de taille $\leq n$ de colonnes de T. On teste ensuite, par analyse des données, s'il existe une dépendance fonctionnelle entre chaque ensemble de la liste et chacune des autres colonnes de T. Cette approche peut être affinée mais elle reste évidemment très lourde.

On donnera assez naturellement la préférence au premier scénario.

f) Autres sources

La liste précédente n'est pas exhaustive. Mentionnons par exemple les fragments de documentation (même obsolètes) qui seraient disponibles, les dictionnaires de données, les spécifications contenues dans les ateliers logiciels, la connaissance des utilisateurs, l'exécution des programmes afin d'en analyser les réactions, etc.

22.4.2 Recherche des constructions implicites

Toute information sur le schéma logique, jugée nécessaire par les utilisateurs des données et par les programmeurs, mais non déclarée en DDL et donc absente du schéma physique brut, constitue une construction implicite. Il existe plusieurs classes de constructions implicites, dont nous mentionnerons les plus importantes : les structures fines des colonnes et des tables, les colonnes obligatoires et facultatives, les structures à définitions multiples, les identifiants de table, les identifiants de colonnes complexes, les clés étrangères, les domaines de valeurs et leurs contraintes, les contraintes d'existence, les bornes exactes des contraintes de cardinalité, les dépendances fonctionnelles, les structures redondantes et les noms significatifs. Nous décrirons brièvement quatre d'entre elles.

a) Structure fine des colonnes et des tables

La première vérification à effectuer est l'atomicité de chaque colonne. Comme illustré dans l'introduction (figure 22.3), une colonne C en apparence atomique est en réalité formée de la concaténation des valeurs d'une colonne composée ou multivaluée. La figure 13.15 nous montrait un autre exemple d'une telle pratique, considérée comme à éviter.

Les techniques les plus appropriées sont l'*analyse des vues SQL*, l'*analyse des flux* (les valeurs de C sont rangées dans une variable composée ou dans un tableau), l'*analyse des interfaces* (chaque valeur de C est éclatée dans plusieurs champs du formulaire ou reconstituée à partir de ceux-ci) et l'*analyse statique ou dynamique des requêtes SQL* (où les clauses select et where contiennent des fonctions d'extraction de sous-chaînes). L'*analyse des données* peut servir de validation.

b) Structures à définitions multiples

Une structure à définitions multiples est constituée d'ensembles de colonnes dont la présence, la structure et la signification sont variables d'une ligne à l'autre. Par exemple, suite aux analyses effectuées aux points (a), (c) et (d) de cette section, une colonne C se voit attribuer plusieurs structures, fonctions et domaines de valeurs contradictoires. Ce phénomène est en général à interpréter à partir des principes de la représentation des relations *is-a* par héritage ascendant (section 19.6.4). Il signifie le plus souvent que cette colonne est l'implémentation unique de plusieurs colonnes différentes soumises à une contrainte d'exclusion (*excl*). À ce stade, il est prématuré d'appliquer une telle interprétation, mais il est important de documenter de manière précise ces structures multiples. C'est en général l'analyse du code des programmes qui mettra en lumière les différentes manières de manipuler une structure multiple.

c) Clés étrangères

Les clés étrangères constituent des constructions extrêmement importantes qui ne sont pas toujours déclarées, même dans les bases de données les plus récentes. La plupart des sources et des techniques d'analyse pourront fournir des indices contribuant à leur détection.

- *Analyse du schéma physique.* Le nom d'une colonne utilisée comme clé étrangère contient souvent des indices de son rôle de référence : nom ou abréviation de la table cible ou de son identifiant primaire, rôle dans le domaine d'application de l'entité cible (*propriétaire, signataire, bien loué*). En outre, clé étrangère et identifiant cible ont des *domaines* de valeurs identiques ou au moins compatibles. Enfin, la plupart des clés étrangères sont supportées par un index et la plupart des *clusters* matérialisent une jointure impliquant une clé étrangère.
- *Analyse du code SQL additionnel.* Les *déclencheurs* sont souvent utilisés pour implémenter la gestion de clés étrangères implicites. Ils sont généralement de petite taille et facile à analyser par inspection visuelle ou automatiquement.
- *Analyse du code source des programmes.* L'*analyse des flux* est une technique simple mais très puissante dans ce cas. Elle permet d'identifier des flux entre deux colonnes de deux tables. Si une des colonnes est un identifiant, l'autre est probablement une clé étrangère (ou une redondance). La recherche des sections de validation des données (par recherche de *patterns* et par *program slicing*) est également recommandée. La technique d'analyse des instructions SQL, dans les programmes et dans les vues, fournit une information très fiable. En particulier, la plupart des jointures sont basées sur une clé étrangère. On sera attentif à un pattern très fréquent mais plus difficile à analyser : la *jointure procédurale*. Celle-ci procède en deux requêtes SQL. La première extrait la valeur de la clé étrangère et la seconde accède à la ligne cible. L'introduction de ce chapitre (section c) en donne un exemple.
- *Examen de l'interface utilisateur.* Un écran est souvent constitué de valeurs extraites de plusieurs tables. La structure de cet écran, mais surtout le code SQL sous-jacent, vont identifier le couplage entre ces tables. Par exemple, la valeur

d'un même champ est utilisée pour accéder aux données principales de l'écran (une ligne de CLIENT) et aux données secondaires (les lignes de COMMANDE).

- *Analyse des données.* Une clé étrangère est soumise à une contrainte d'inclusion que nous pourrions facilement vérifier par une analyse de données :

```
select NCOM, NCLI from COMMANDE
where NCLI not in (select NCLI from CLIENT)
```

Attention cependant, une inclusion partielle (la requête précédente fournit 2 % des lignes de COMMANDE par exemple) ne signifie pas nécessairement l'absence de clé étrangère, mais peut être le fait de données erronées dont l'insertion a échappé aux mécanismes de validation.

d) Domaines de valeurs et contraintes

Une colonne C de type très générique, char(6) par exemple, peut en réalité contenir un ensemble de valeurs fortement contraint : nombres entiers, codes postaux, taille limitée ('vrai', 'faux'), intervalle de valeurs (de '1990' à '2020'), syntaxe stricte (3 lettres + 3 chiffres). L'identification du domaine le plus précis de C pourra démarrer par une *analyse des données* (certains parleraient de *data profiling*) mais profitera également d'une recherche des *patterns* dans les programmes dédiés à la modification des valeurs de C (quelle validation de C avant écriture dans la base de données) ainsi que d'un examen des écrans de l'interface.

La transformation du schéma physique enrichi en schéma logique consiste simplement à supprimer les constructions physiques désormais inutiles, telles que les index, des espaces de stockages et les clusters. Notons que le schéma logique résultant n'est pas à proprement parler strictement conforme au SGBD. En effet, il contient des constructions telles que des colonnes composées, des colonnes multivaluées ainsi que des contraintes additionnelles inconnues dans le modèle du SGBD.

22.5 CONCEPTUALISATION DU SCHÉMA LOGIQUE

Il s'agit à présent d'interpréter le schéma logique enrichi sous la forme d'un schéma conceptuel normalisé, ce que nous nommerons *conceptualisation*. Si l'interprétation du schéma logique s'est avérée très simple dans le cas de la figure 22.2 de l'introduction, il n'en va malheureusement pas toujours de même dans la réalité. Le chapitre 19 et l'annexe A26 nous auront appris deux choses importantes mais qui vont nous compliquer la vie : (1) il existe souvent plusieurs manières de traduire une construction conceptuelle et (2) il est possible que plusieurs constructions conceptuelles se traduisent par les mêmes constructions logiques. En d'autres termes, les règles de traduction *conceptuel/logique* correspondent à des relations *plusieurs-à-plusieurs*. La conceptualisation risque donc d'apparaître souvent comme un processus complexe et non déterministe. Un exemple pour illustrer ce point : comment interpréter une *clé étrangère identifiante* ? Il peut s'agir de la traduction d'un type d'associations *un-à-un* au plein sens du terme (un contrat *couvre* un véhicule, une personne *est mariée* à une autre), ou de l'association d'un simple fragment

complémentaire à un type d'entités majeur (CLIENT et HISTORIQUE de la figure 18.57 par exemple) ou encore l'expression d'une relation *is-a* (figure 19.20).

Nous rencontrerons d'autres sources importantes de difficultés. La première réside dans les objectifs du schéma logique dans l'esprit de son concepteur. Ce dernier y a souvent introduit de nombreuses constructions visant à **optimiser** les performances de la base de données en temps d'accès et en espace occupé (section 20.7). Ces constructions vont évidemment compliquer la conceptualisation. En effet, nous devons clairement identifier ces constructions comme telle et les éliminer tout en préservant celles qui ont une réelle signification. Nous retrouverons dans cette catégorie la plupart des problèmes de normalisation que nous avons étudiés à la section 18.7, et en particulier les différentes formes de redondance de contraintes, d'instances et internes. Une deuxième source de difficultés à laquelle nous serons souvent confrontés est l'usage de pratiques empiriques, voire maladroites ou même invalides lors de la construction du schéma logique, que nous appellerons collectivement et pudiquement **pratiques non standard**. Enfin, une base de données ressemble parfois à ces édifices historiques dans lesquels on repère des parties de styles et de périodes très différents⁶, des sections à l'abandon ou dédoublées suite à un différent politique. Une grande base de données d'âge vénérable⁷ (et son schéma logique) est souvent constituée de parties de styles hétérogènes, de données inutilisées depuis plusieurs années, de données accidentellement ou volontairement dédoublées.

La prise en compte de ces difficultés nous suggère d'organiser la conceptualisation selon deux processus aux noms barbares, la *détraduction* et la *désoptimisation*, qui produisent le schéma conceptuel brut, lequel est ensuite normalisé⁸.

De même qu'il existe des plans de transformations pour la conception logique, on construira des plans de transformations pour la conceptualisation.

a) Détraduction d'un schéma logique

Ce processus est essentiellement l'inverse de la conception logique que nous avons décrite dans les chapitres précédents. Celle-ci ne se préoccupait pas, ou très peu, de critères d'optimisation, lesquels étaient pris en charge par la conception physique.

Pour un modèle logique M, on se donne un répertoire de transformations qui nous garantit la traduction sans perte d'information de n'importe quel schéma conceptuel en un schéma selon M. En inversant chacune de ces transformations (en les lisant de droite à gauche), on construit une panoplie d'outils de détraduction susceptible de retrouver un schéma conceptuel équivalent au schéma de départ. Bien sûr, l'analyse développée précédemment nous a rappelé que la conception logique utilisait des règles *plusieurs-à-plusieurs*, de sorte qu'un même schéma logique peut produire

6. Églises St-Eustache et St-Sulpice à Paris, cathédrale de Tournai.

7. De nombreuses bases de données sont âgées de plus de 30 ans. Plusieurs applications administratives encore en activité exploitent des bases de données IDS/1 ou IMS qui ont créées à la fin des années soixante.

8. En pratique, il sera souvent impossible d'appliquer ces trois processus indépendamment. Il faut dès lors les considérer plutôt comme trois classes de problèmes à résoudre au moyen de raisonnements et techniques spécifiques.

plusieurs schémas conceptuels différents selon les règles utilisées et l'ordre dans lequel on les applique. Cette variabilité des schémas sera abordée dans le processus de normalisation.

Nous suggérons d'utiliser un très petit nombre de règles de détraduction et de laisser la résolution des problèmes d'ambiguïté d'interprétation au processus de normalisation. Tel que nous l'avons défini, celui-ci dispose en effet d'un arsenal très puissant pour identifier et résoudre les multiples problèmes de qualité des schémas conceptuels. Les règles suivantes seraient d'application générale pour les schémas relationnels :

- une clé étrangère est transformée en un type d'associations fonctionnel⁹ (inverse des règles de la figure 19.14) ;
- une suite d'attributs hétérogènes mais de noms similaires est transformée en un attribut composé (inverse des règles de la figure 19.6/a) ;
- une suite d'attributs homogènes de noms similaires est transformée en un attribut multivalué (inverse des règles de la figure 19.13/c).

Les inverses des autres transformations du chapitre 19 seront assurées par le processus de normalisation.

b) Désoptimisation d'un schéma logique

Il n'existe pas d'étude générale des pratiques d'optimisation de schémas logiques mais la plupart des techniques peuvent se classer en trois catégories (section 20.7).

- *Restructuration de schémas*. On y trouvera principalement les techniques de division de tables. La *division verticale* d'une table remplace celle-ci par deux ou plusieurs tables reprenant une partie seulement des colonnes d'origine. La *division horizontale* d'une table remplace celle-ci par deux ou plusieurs tables de même schéma mais contenant chacune un sous-ensemble des lignes d'origine. *Désoptimisation* : jointure ou union des tables fragmentées.
- *Redondances structurelles*. Les instances d'une construction redondante sont calculables à partir de celles d'autres constructions de la même base de données. *Désoptimisation* : suppression des constructions redondantes.
- *Redondances internes*. Il s'agit principalement de structures non normalisées au sens du chapitre 3. *Désoptimisation* : décomposition des objets (types d'entités et d'associations) non normalisés.

On observera que l'exposé très large que nous avons fait du processus de normalisation d'un schéma conceptuel (section 18.7) inclut les raisonnements de désoptimisation suggérés précédemment. Il ne serait cependant pas opportun d'écarter le processus de désoptimisation. En effet, celui-ci attaque un problème spécifique qu'il est recommandé de traiter comme tel. Pour s'en convaincre, on se référera à des ouvrages tels que [Shasha, 2002] et [Lightstone, 2007] qui, bien que consacrés à la

9. Il s'agit d'une règle particulièrement simpliste. Elle sera complétée dans la référence [Hainaut, 2009c] et développée de manière plus exhaustive dans la référence [Hainaut, 2009d], qui lui est entièrement consacrée.

conception physique d'une base de données relationnelle, suggèrent des transformations du schéma logique en vue d'une amélioration des performances. De tels raisonnements *très physiques*, sont éloignés des préoccupations du *normalisateur conceptuel* !

c) Normalisation d'un schéma conceptuel brut

Nous donnerons à ce processus une grande importance. D'une part, il permet de retrouver de manière systématique les constructions avancées du modèle Entité-association. D'autre part, comme il fait partie intégrante de la conception d'une base de données, les techniques qu'il utilise sont supposées connues et maîtrisées, de sorte que la rétro-ingénierie apparaît comme une extension naturelle de l'ingénierie classique. Nous ne reprendrons de la section 18.7 que les transformations les plus représentatives. On n'oubliera pas cependant que tous les *patterns* que nous y avons décrits sont susceptibles d'apparaître dans un schéma produit par détraduction/désoptimisation.

- Transformation des types d'entités attributs (18.7.1/a).
- Transformation des types d'entités associations (18.7.2/a).
- Transformation des attributs complexes (18.7.2/b).
- Recouvrement des relations *is-a* implicites (18.7.2/f).

d) Un plan de transformation simple pour la conceptualisation

La figure 22.6 décrit un plan de transformation applicable à des schémas logiques relationnels bien formés, c'est-à-dire qui ont été obtenus par un plan de transformation systématique et non optimisant tel que celui qui a été proposé au chapitre 19. Il comprend les règles de *détraduction* et de normalisation les plus courantes :

- *Substitution des noms*. Lors de la recherche des constructions implicites, des synonymes plus significatifs ont été détectés pour les noms de colonnes et de tables. On propose de les assigner à ces objets.
- *Transformation des clés étrangères en types d'associations*. Sauf dans des schémas complexes dotés de clés étrangères multivaluées, on produit ainsi des types d'associations fonctionnels.
- *Reconstruction des attributs composés*. Une suite d'attributs hétérogènes mais de noms similaires est transformée en un attribut composé.
- *Reconstruction des attributs multivalués*. On utilisera deux transformations classiques : les séries d'attribut de même type et de noms similaires ainsi que les types d'entités attributs.
- *Reconstruction des types d'associations complexes*. On appliquera la transformation de types d'entités associations en types d'associations.
- *Reconstruction des relations is-a*. On s'inspirera des règles de traduction de la section 19.6. Il est illusoire d'espérer une uniformité dans les règles d'expression utilisées par le développeur du schéma. Il est probable que nous rencontrerons des

traces de chacune des trois techniques décrites dans cette section. On sera en particulier attentif aux constructions suivantes :

- ensemble de types d'associations 1:1 issus d'un même type d'entités (transformation par matérialisation) ;
- types d'entités munis d'attributs et rôles similaires (transformation par héritage descendant) ;
- groupes d'attributs et de types d'associations soumis à des contraintes d'existence complexes (transformation par héritage ascendant).
- *Suppression des identifiants techniques inutiles*. Les identifiants dont le rôle est uniquement de permettre la définition de clés étrangères sont supprimés.
- *Traitement des noms*. Des noms significatifs sont assignés aux nouveaux objets. Les noms sont normalisés.

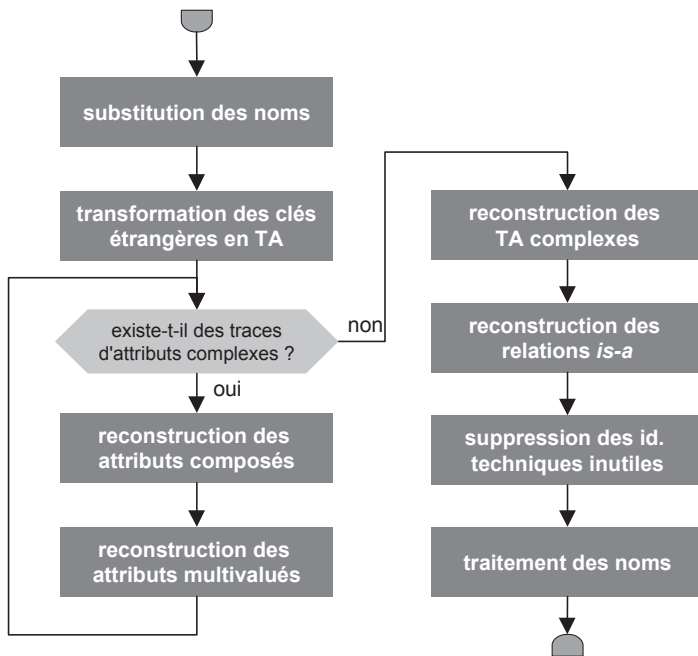


Figure 22.6 - Un plan de transformation pour la conceptualisation de schémas relationnels *bien formés*

22.6 QUE RETENIR ?

La rétro-ingénierie d'une base de données consiste à reconstruire son schéma logique et un schéma conceptuel plausible. Elle comprend trois grandes phases : l'extraction du schéma physique exprimé dans le code SQL, l'enrichissement de ce dernier des constructions implicites retrouvées par l'analyse de diverses sources d'information : schéma physique, contenu de la base de données, code source des programmes d'application, écrans de l'interface utilisateur, etc. le résultat est le

schéma logique complet. De ce dernier, on extrait un schéma conceptuel normalisé. Pour ce faire, on applique d'abord des transformations de détraduction (inverses de celles de la conception logique) et de désoptimisation, suivies d'un processus de normalisation.

22.7 POUR EN SAVOIR PLUS

Ce chapitre est une synthèse de la référence [Hainaut, 2009c]. Celle-ci développe toutes les techniques mentionnées dans ce chapitre et contient en particulier une bibliographie étendue. Le chapitre 23 contient une étude de cas principalement basée sur l'analyse du code source de programmes. D'autres sont présentées à l'annexe A23. Peu d'ouvrages traitent spécifiquement de la rétro-ingénierie des bases de données ([Aiken, 1995], épuisé mais peut-être encore disponible sur Amazon). Les principales sources plus techniques sont les actes des conférences WCRE et CSMR ainsi que les revues *Journal of Software Systems*, *Journal of Software Maintenance and Evolution*, *Data & Knowledge Engineering* et *Information Systems*.

22.8 RÉTRO-INGÉNIERIE À L'AIDE D'UN OUTIL

La plupart des ateliers logiciels offrent des fonctions élémentaires de rétro-ingénierie de bases de données relationnelles. Ces fonctions se réduisent en général à un extracteur de schéma physique brut, à la recherche de clés étrangères implicites sur la base de l'égalité des noms *identifiant primaire/clé étrangère* et à un processus de conceptualisation limité à la transformation *clé étrangère/type d'associations fonctionnel*, et, dans de rares cas, à la transformation (parfois abusive) des clés étrangères identifiantes en relations *is-a*. L'atelier DB-MAIN offre en revanche une gamme étendue d'outils d'aide à la rétro-ingénierie de bases de données, y compris :

- les extracteurs SQL pour les principaux SGBD, des extracteurs COBOL, CODASYL, IMS ;
- un assistant de détection de clés étrangères exploitant plusieurs heuristiques ;
- des assistants d'analyse de code source (analyse des dépendances, analyse des flux, *program slicing*, recherche de patterns paramétriques, analyse statique de requêtes SQL, analyse dynamique de requêtes SQL) ;
- des assistants d'analyse et de transformation de schémas ;
- un générateur de programmes d'analyse de données.

L'annexe A22 propose un approfondissement de certaines techniques mentionnées dans ce chapitre. Les scripts implémentant ces techniques (*analyse de données et recherche des identifiants potentiels*) sont disponibles dans la distribution de SQLfast (SQLfast/Scripts/BD-2018/Chapitre-22). L'annexe A23 propose plusieurs études de cas de rétro-ingénierie.

Chapitre 23

Étude de cas

Ce chapitre développe une étude de cas représentative dans le domaine des bases de données. Cette étude est accessible au lecteur débutant. D'autres études de cas sont disponibles sur le site de l'ouvrage.

23.1 LES ANIMAUX DU ZOO

Nous proposerons au lecteur une visite au zoo. Un énoncé en langage courant sera décomposé, analysé puis traduit en un schéma conceptuel. Celui-ci sera ensuite converti en structures de tables. À partir d'un domaine d'application très accessible, on parcourt ainsi l'essentiel des processus de conception et de production décrits aux chapitres 13 et 14.

23.1.1 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 (sels minéraux, vitamines, protéines, etc.), exprimés en grammes 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 grammes par gramme d'ingrédient. Chaque ingrédient a un coût unitaire. Chaque animal requiert des soins qui sont évalués en euros par jour. Ces soins peuvent varier d'un jour à l'autre.

23.1.2 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. Nous traduirons le résultat de cette analyse sous la forme d'un schéma Entité-association étendu (il comprend des types d'associations avec attributs). Pour le lecteur débutant, le schéma conceptuel sera exprimé dans le modèle Entité-association de base (types d'associations sans attributs). Il est représenté à la figure 23.2.

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 régime**. Ce type d'associations peut être *un-à-plusieurs* ou *un-à-un*. On choisit la seconde proposition, 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. On attache au type d'associations **composé de** l'attribut **Quantité**.
5. *Le régime que reçoit un animal dépend du jour.* Un animal peut donc avoir plusieurs régimes. Le type d'associations **suit régime** 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 hybride 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 **besoins**.
8. *Calcium, protéines sont des exemples de nutriments.* Ces exemples suggèrent un attribut **Nom nutriment** identifiant de **NUTRIMENT**.

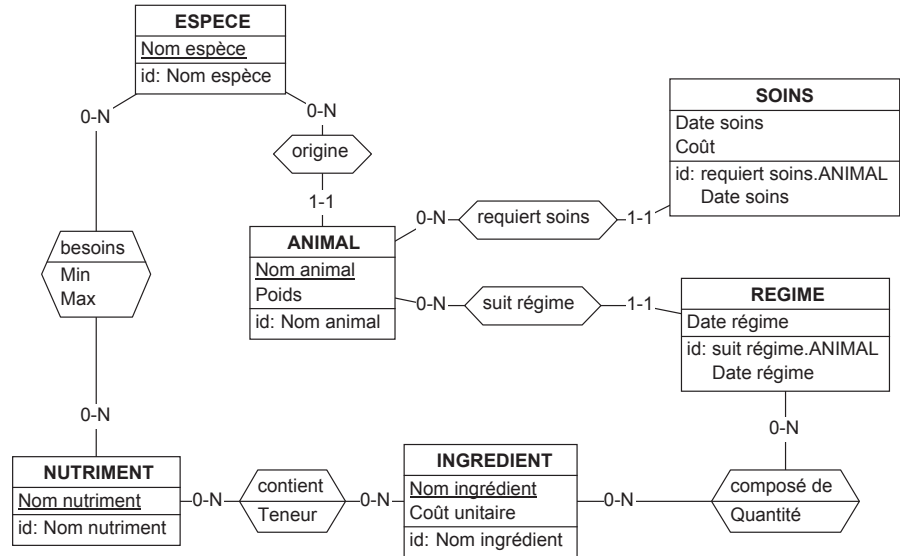
9. *Les besoins d'un animal en un ingrédient sont caractérisés par une quantité minimale ...*. Les besoins d'un animal sont ceux de son espèce. Il n'est donc pas nécessaire d'établir un type d'associations spécifique entre **ANIMAL** et **NUTRIMENT**. Le concept de besoin est caractérisé par une quantité minimale, qui se traduit naturellement par l'attribut **Min** du type d'associations **besoin**.
10. *... et une quantité maximale.* ... nouvel attribut **Max** de **besoin**. La valeur de ce dernier n'est pas inférieure à celle de **Min** de son association, ce qu'on traduit par la contrainte $\text{Max} \geq \text{Min}$.
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.
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, assigné au type d'associations **contient**.
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 requiert soins 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 (euros par jour), une entité **SOINS** représente l'ensemble des prestations destinées à un animal, un jour déterminé. On traduit ce fait par l'identifiant hybride 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 23.1 synthétise les résultats de cette analyse.

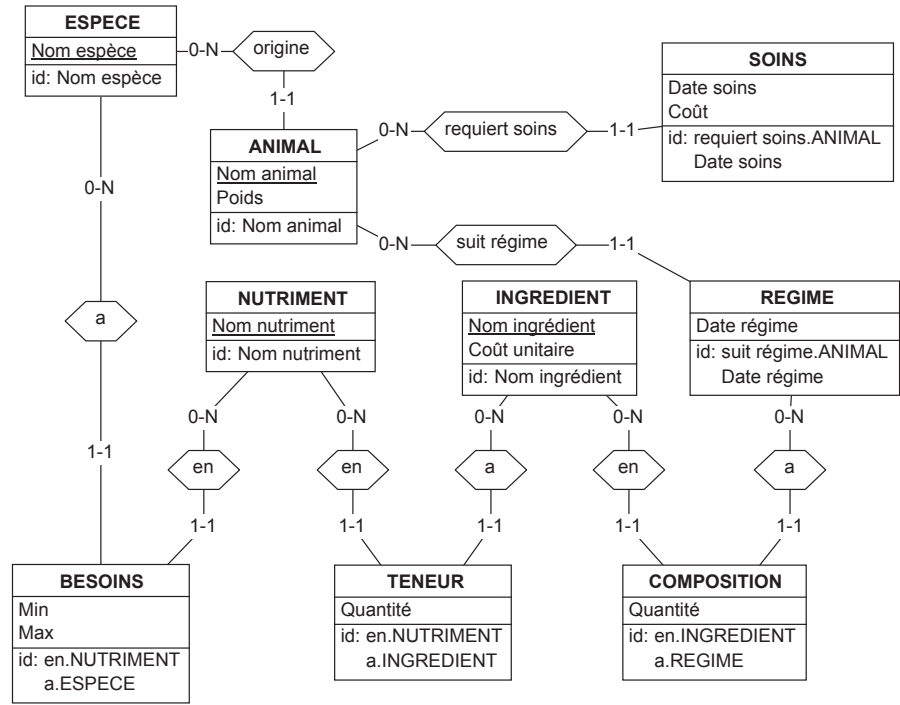
23.1.3 Production du schéma de tables et du code SQL

Les trois types d'associations complexes (besoins, contient, composé de) sont d'abord transformés en types d'entités (schéma 23.2). La traduction de ce schéma sous forme de structures de tables ne pose pas de problèmes particuliers (schéma 23.3). 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 "_". Des index ont été associés à chaque identifiant et chaque clé étrangère. Cependant, les index qui reprennent les premières colonnes d'un autre index sont supprimés.



Contrainte : Pour toute association besoins b, b.Max >= b.Min

Figure 23.1 - Les animaux du zoo : schéma conceptuel (modèle EA étendu)



Contrainte : Pour toute entité BESOINS b, b.Max >= b.Min

Figure 23.2 - Les animaux du zoo : schéma conceptuel (modèle EA de base)

Le code SQL-DDL se déduit immédiatement du schéma des tables. Le script **ZOO-creer-BD.sql** contient ce code de création ainsi qu'un jeu de données relatif à 12 espèces, 25 animaux (y compris *Capitaine*, un *crocodile* de 308 kg), 13 ingrédients et 9 nutriments. Ce script charge également des données sur les teneurs et les besoins, ainsi qu'un exemple de régime (au profit de notre crocodile) et sa composition, illustrés à la figure 23.4. Les données sont exprimées dans les unités suivantes : poids des animaux en kg, besoins en g de nutriment par g d'animal, teneur en g de nutriment par g d'ingrédient et composition en g d'ingrédient. Le coût d'un ingrédient est exprimé par kg.

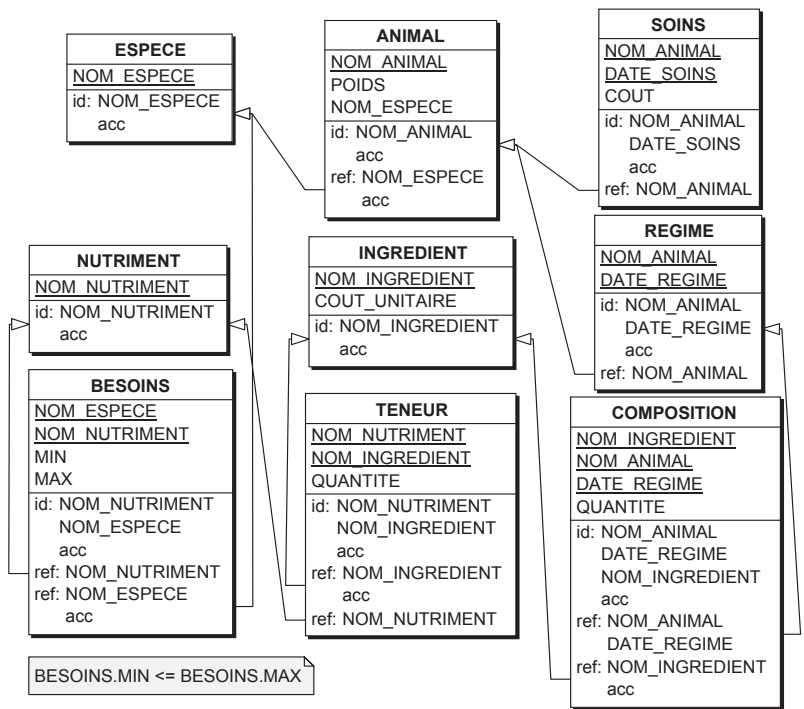


Figure 23.3 - Les animaux du zoo : schéma de tables

23.1.4 Exploitation

À titre d'exemple, on rédigera les requêtes SQL répondant à deux questions typiques auxquelles la base de données que nous venons de construire peut répondre.

- **Quel a été le coût en nourriture de chaque animal le 22 février 2018 ? ¹**

```
select C.NOM_ANIMAL,
       sum(C.QUANTITE*I.COUT_UNITAIRE/1000) as COUT
```

1. On note l'absence des tables ANIMAL et REGIME dans la requête. Elle sont en effet inutiles, leurs identifiants étant présents dans la table COMPOSITION.

```
from COMPOSITION C, INGREDIENT I
where C.DATE_REGIME = '2018-02-22'
and I.NOM_INGREDIENT = C.NOM_INGREDIENT
group by C.NOM_ANIMAL
```

	NOM_INGREDIENT	NOM_ANIMAL	DATE_REGIME	QUANTITE
1	biofin-2	Capitaine	2018-02-22	12
2	granules-P06	Capitaine	2018-02-22	30
3	multivit-ADE2	Capitaine	2018-02-22	2
4	poisson	Capitaine	2018-02-22	4000
5	viande rouge	Capitaine	2018-02-22	10000

Figure 23.4 - Le régime du 22 février du crocodile Capitaine

Soit, pour notre crocodile :

NOM_ANIMAL	COUT
Capitaine	49.1056

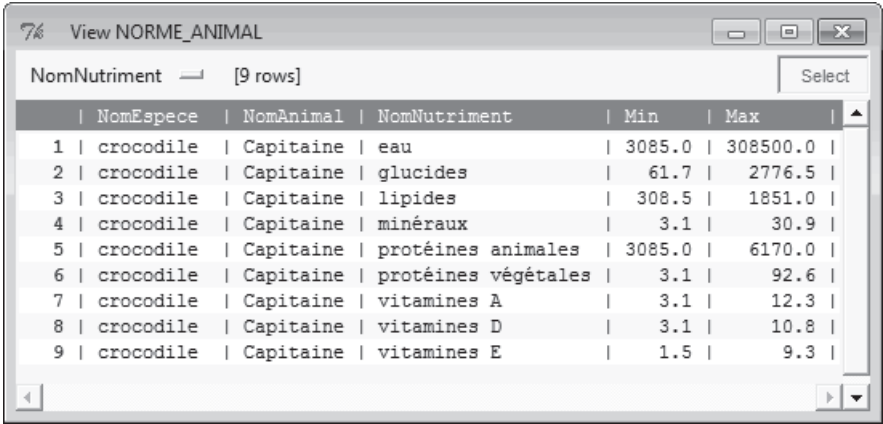
• *Le régime d'un animal est-il conforme à ses besoins nutritionnels ?*

Il s'agit d'une question complexe que nous allons résoudre en deux étapes. La réponse résulte de la comparaison de deux ensembles de données, dont l'extraction constitue la première étape :

1. *données de référence* : quelles sont les **quantités minimale et maximale** de chaque nutriment que tout régime doit fournir à cet animal ?
2. *bilan nutritionnel du régime* : quelles sont les quantités de chaque nutriment que ce régime fournit **effectivement** à cet animal ?

On traduira l'extraction des données de référence par la vue NORME_ANIMAL, dont le contenu est visualisé à la figure 23.5.

```
create view NORME_ANIMAL (NomEspece, NomAnimal, NomNutriment,
                          Min_gr, Max_gr) as
select A.NOM_ESPECE, A.NOM_ANIMAL, B.NOM_NUTRIMENT,
       B.MIN*A.POIDS, B.MAX*A.POIDS
from ANIMAL A, BESOINS B
where A.NOM_ESPECE = B.NOM_ESPECE;
```

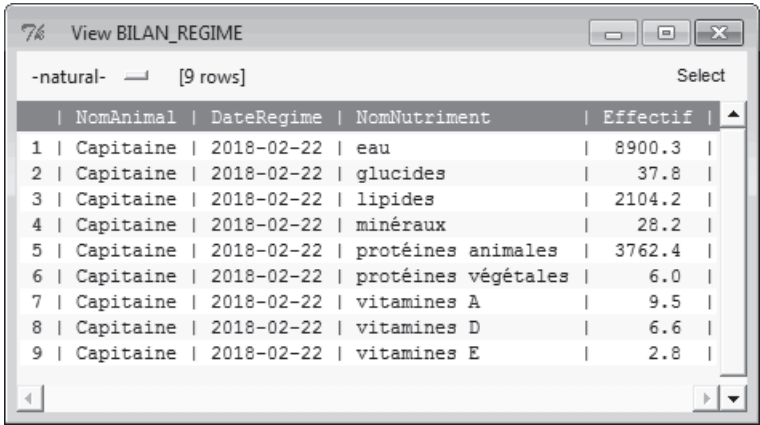


7% View NORME_ANIMAL [9 rows] Select

	NomEspece	NomAnimal	NomNutriment	Min	Max
1	crocodile	Capitaine	eau	3085.0	308500.0
2	crocodile	Capitaine	glucides	61.7	2776.5
3	crocodile	Capitaine	lipides	308.5	1851.0
4	crocodile	Capitaine	minéraux	3.1	30.9
5	crocodile	Capitaine	protéines animales	3085.0	6170.0
6	crocodile	Capitaine	protéines végétales	3.1	92.6
7	crocodile	Capitaine	vitamines A	3.1	12.3
8	crocodile	Capitaine	vitamines D	3.1	10.8
9	crocodile	Capitaine	vitamines E	1.5	9.3

Figure 23.5 - Les besoins standard en nutriments du crocodile *Capitaine*

Le bilan nutritionnel effectif des régimes requiert les données de COMPOSITION de ceux-ci, puis celles des INGREDIENTs composants et enfin de la TENEUR en nutriments de ceux-ci. Le résultat est traduit par la vue BILAN_REGIME. La figure 23.5 en donc un exemple de contenu.



7% View BILAN_REGIME [9 rows] Select

	NomAnimal	DateRegime	NomNutriment	Effectif
1	Capitaine	2018-02-22	eau	8900.3
2	Capitaine	2018-02-22	glucides	37.8
3	Capitaine	2018-02-22	lipides	2104.2
4	Capitaine	2018-02-22	minéraux	28.2
5	Capitaine	2018-02-22	protéines animales	3762.4
6	Capitaine	2018-02-22	protéines végétales	6.0
7	Capitaine	2018-02-22	vitamines A	9.5
8	Capitaine	2018-02-22	vitamines D	6.6
9	Capitaine	2018-02-22	vitamines E	2.8

Figure 23.6 - Bilan nutritionnel du régime

```
create view BILAN_REGIME (NomAnimal,DateRegime,NomNutriment,
                          Effectif) as
select C.NOM_ANIMAL,C.DATE_REGIME,T.NOM_NUTRIMENT,
       sum(C.QUANTITE*T.QUANTITE)
from   COMPOSITION C, INGREDIENT I, TENEUR T
where  C.NOM_INGREDIENT = I.NOM_INGREDIENT
and    I.NOM_INGREDIENT = T.NOM_INGREDIENT
group by C.NOM_ANIMAL,C.DATE_REGIME,T.NOM_NUTRIMENT;
```

L'évaluation d'un régime s'obtient dès lors par une simple jointure de ces deux vues :

```
select NA.NomNutriment, NA.Min, BR.Effectif, NA.Max,
       case when BR.Effectif >= NA.Min and BR.Effectif <= NA.Max
            then 'OK' else 'Erreur'
       end as Evaluation
from   NORME_ANIMAL NA, BILAN_REGIME BR
where  NA.NomAnimal = BR.NomAnimal
and    NA.NomNutriment = BR.NomNutriment
and    NA.NomAnimal = 'Capitaine' and BR.DateRegime = '2018-02-22'
```

Elle produit le tableau ci-dessous :

NomNutriment	Min	Effectif	Max	Evaluation
eau	3085.0	8900.3	308500.0	OK
glucides	61.7	37.8	2776.5	Erreur
lipides	308.5	2104.2	1851.0	Erreur
minéraux	3.1	28.2	30.9	OK
protéines animales	3085.0	3762.4	6170.0	OK
protéines végétales	3.1	6.0	92.6	OK
vitamines A	3.1	9.5	12.4	OK
vitamines D	3.1	6.6	10.8	OK
vitamines E	1.5	2.8	9.3	OK

Les scripts relatifs à cette étude de cas sont disponibles dans le répertoire **SQLfast/Scripts/BD-2018/Chapitre-23** de la distribution SQLfast.

23.2 AUTRES ÉTUDES DE CAS

D'autres études de cas sont disponibles à l'annexe A23.

Bibliographie

- ABITEBOUL, S., HULL, R., VIANU, V., *Fondements des bases de données*, Vuibert, 2000.
- ABRIAL, J.-R., Data Semantics, in *Data Base Management*, North Holland Publish. 1974, pp. 1-59.
- AIKEN, P., *Data reverse engineering: Slaying the legacy dragon*, McGraw-Hill, 1995.
- AKOKA, J. et COMYN-WATTIAU, I., *Conception des bases de données relationnelles*, Vuibert, 2001.
- AMBLER, S., *A UML Profile for Data Modeling*, août 2006, <http://www.agiledata.org/essays/umlDataModelingProfile.html>.
- AMBLER, S., *Mapping Objects to Relational Databases: O/R Mapping In Detail*, 2006b, <http://www.agiledata.org/essays/mappingObjects.html> (janvier 2009).
- AMBLER, S., SADALAGE, P., *Refactoring Databases*, Addison Wesley, 2007.
- ANTONPOULOS, A., *Mastering Bitcoin: Programming the Open Blockchain*, O'Reilly, 2017.
- ANSI, *ANSI Database Language SQL with Integrity Enhancement*, ANSI X3.135.1, 1989. Disponible, ainsi que les rapports suivants, à l'AFNOR.
- ATZENI, P. and DE ANTONELLIS, V., *Relational Database Theory*. Benjamin-Cummings Publishing Co., Inc., 1993.
- BACHMAN, C., Data Structure Diagrams, in *Data Base*, 1(2), 1969, ACM SIG on Business Data Processing.
- BAPTISTE, J.-L., *Merise - Guide pratique*, ENI, 2012
- BARBIER, F., HENDERSON-SELLERS, B., PARC-LACAYRELLE, A., BRUEL, J.-M. Aggregation and Composition in UML: A Rectified Implementation Based on Whole-Part Theory. *IEEE Transactions on Software Engineering*, 29(5), 2003.
- BATINI, C., CERI, S. et NAVATHE, S., B., *Conceptual Database Design - An Entity-Relationship Approach*, Benjamin/Cummings, 1992.
- BEN NATAN, R., *Implementing Database Security and Auditing*, Elsevier Digital Press, 2005.
- BERTINO, E., Data Security, *Data & Knowledge Engineering*, 25 (1998), pp. 199-216, Elsevier.

- BLAHA, M. et PERMERLANI, W., *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, 1998.
- 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.
- BOUFARES, F., BENNACEUR, H., Consistency problems in ER-schemas for database systems, *Information Sciences*, Elsevier, 163, pp. 263–274, 2004.
- BOUZEGHOUB, M., GARDARIN, G. et VALDURIEZ, P., *Les objets*, Eyrolles, 1997.
- BOUZEGHOUB, M., JOUVE, M., PUCHERAL, P., *Le modèle relationnel*, Hermès - Lavoisier, 1998.
- BOWERS, D., Detection of Redundant Arcs in Entity Relationship Conceptual Models, in *Proc. ER 2002 workshops*, LNCS 2784, pp. 275-287, Springer, 2003.
- BRACHMAN, R. What IS-A is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Computer*. 16(10). pp. 30-36. 1983.
- BRUCHEZ, R., *Les bases de données NoSQL*, Eyrolles, 2013.
- CAPEC, *Common Attack Pattern Enumeration and Classification*, 2008, <http://capec.mitre.org/data/dictionary.html>.
- CELKO, J., *SQL avancé : programmation et techniques avancées*, Vuibert, 2000.
- CERI, S. ET FRATERNALI, 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., 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., The Entity-Relationship Model - Towards a Unified View of Data, in *ACM TODS*, Vol. 1, No 1, pp. 9-36, 1976.
- CHEN, P., English Sentence Structure and Entity Relationship Diagrams, *Information Science* 1(1), pp. 127-149, Elsevier, 1983.
- CHODOROW, K., *MongoDB The definitive guide*, O'Reilly, 2013
- CHOOBINEH, J., MANNINO, M., TSENG, V., A Form-Based Approach for Database Analysis and Design. *Comm. ACM*, 35(2), pp. 108-120, 1992.
- CHUNG, L., NIXON, B., YU, E., MYLOPOULOS, J., *Non-functional Requirements in Software Engineering*, Kluwer, 1999.
- CLARK, J., *SQL Injection Attacks and Defense*, Syngress Publish., Elsevier, 2009.
- CLEVE, A., MEURISSE, J-R., HAINAUT, J-L., Database Semantics Recovery through Analysis of Dynamic SQL Statements, in *Journal of Data Semantics*, Vol. XV, pp. 130-157, LNCS 6720, Springer, 2011.
- CODD, E., F., A Relational Model of Data for Large Shared Data Banks, in *Comm. ACM*, Vol. 13, No 6, June 1970.
- CODD, E., F., Extending the Database Relational Model to Capture more Meaning, in *ACM TODS*, Vol. 4, No 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, 2009.
- CUÉNOD, S., *Faussement vrai & vraiment faux, La logique au quotidien*, Presses Polytechniques et Universitaires Romandes, Lausanne, 1999.
- CWE, *Common weakness enumeration - CWE Dictionary*, 2008, <http://cwe.mitre.org/data/dictionary.html>.
- DAHCHOUR, M., PIROTTÉ, A., ZIMANYI, E., Materialization and its Metaclass Implementation, *IEEE Transactions on Knowledge and Data Engineering*, 14(5), 2002.
- DANIALIS, H., A method for validating a conceptual model by natural language discourse generation, in *Proc. CAiSE* 1992.
- 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., DARWEN H. - *The SQL Standard*, Addison-Wesley, 1997.
- DATE, C., J., *Introduction aux bases de données*, Vuibert Informatique, 2004.
- DATE, C., J., *No! to SQL and No! to NoSQL*, <https://iggyfernandez.wordpress.com/2013/07/28/no-to-sql-and-no-to-nosql> (juin 2015)
- DA SILVA, A, LAENDER, A., CASANOVA, M., On the relational representation of complex specialization structures, *Information Systems* 25(6-7), 2000, pp. 399-415.
- DASPET, E. et de GEYER, C., *PHP 5 avancé*, Eyrolles, 2004.
- 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.
- DELMOND, M.H., PETIT Y. et GAUTIER J.M., *Management des Systèmes d'Information*, Dunod, 2007.
- DELOBEL, C., CASEY, R., G., Decomposition of a database and the theory of boolean switching functions, *IBM Journal of Research and Development*, Vol. 17, No 5, pp. 374-386.
- DELOBEL, C., ADIBA, M., *Bases de données et systèmes relationnels*, Dunod Informatique, 1982.
- DENG Y., What is the Future of Disk Drives, Death or Rebirth?, *ACM Computing Surveys*, 43(3), article 23, 2011.
- DENNING D., DENNING P., SCHWARTZ, M., The Tracker: A Threat to Statistical Database Security, *ACM Transactions on Database Systems*, 4(1), 1979, pp. 76-96.
- DETENNE, V., HAINAUT, J-L., CASE Tool Support for Temporal Database Design, in *Proc. of 20th Int. Conf. on Conceptual modeling (ER 2001)*, Springer Verlag LNCS 2224, 2001.
- DULLEA, J., SONG, I.-Y., A Taxonomy of Recursive Relationships and their Structural Validity in ER Modeling, in *Proc. of the 18th International Conference on Conceptual Modeling*, LNCS 1728, Paris, France, November 1999, pp. 384-398.

- D'SOUZA D. et WILLS A., *Objects, Components and Frameworks with UML*, Addison-Wesley, 1999.
- ELMASRI R. et NAVATHE S., *Fundamentals of Database Systems*, 5th Edition, Addison-Wesley, 2006.
- ELMASRI, R., NAVATHE, S., SERAIN, S., *Conception et architecture de base de données*, Pearson Education, 2004.
- FAHRNER, C., VOSSEN, G., A survey of database design transformations based on the Entity-Relationship model, *Data & Knowledge Engineering* 15 (1995) pp. 213-250.
- 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.
- GARDARIN, G., *Bases de données*, Eyrolles, 2003.
- GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., *Database Systems: The Complete Book*, Prentice Hall, 2008.
- GEIGER, *Inside ODBC*, Microsoft Press, 1995.
- GENERO, M., POELS, G., PIATTINI, M., Defining and Validating Measures for Conceptual Data Model Quality, *Proc. CAISE 2002*, LNCS 2348, pp. 724-727, Springer, 2002.
- GOGOLLA, M., HOHENSTEIN, U., Towards a semantic view of an extended entity-relationship model. *ACM TODS* 16 (1991) 369-416.
- GOLDSTEIN, C., STOREY, V., Materialization, *IEEE Transactions on Knowledge and Data Engineering*, 6(5), 1994.
- GOMEZ, F., SEGAMI, C., DELAUNE, C., A System for the Semi-automatic Generation of ER Models from Natural Language Specifications. *Data and Knowledge Engineering* 29(1), pp. 57-81, 1999.
- GRAY, J et REUTER, A., *Transaction processing : concepts and techniques*, Morgan Kaufmann, 1993.
- HAIGH, "A Veritable Bucket of Facts" Origins of the Data Base Management System, *ACM SIGMOD Records*, 35(2), 2006.
- HAINAUT, J.-L., *Conception assistée des applications informatiques - Conception de la base de données*, Masson, 1986. (également disponible sur <http://www.info.fundp.ac.be/libd>).
- HAINAUT, J.-L., Entity-generating Schema Transformations for Entity-Relationship Models, in *Proc. of the 10th Entity-Relationship Approach*, San Mateo (CA), 1991, North-Holland, 1992.
- 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., HICK, J.-M., ENGLEBERT, V., HENRARD, J., ROLAND, D., Understanding implementations of IS-A Relations, in *Proc. of the conference on the ER Approach*, Cottbus, October 1996, LNCS, Springer-Verlag, 1996.
- HAINAUT, J.-L., Specification preservation in schema transformations - Application to semantics and statistics, *Data & Knowledge Engineering*, 16(1), Elsevier Science Publish., 1996.
- HAINAUT, J.-L. - *Introduction to Database Design*, Laboratoire d'ingénierie des applications de bases de données, Université de Namur, 2002. (disponible sur <http://www.info.fundp.ac.be/libd>).
- HAINAUT, J.-L., *UML ou ERA : quel modèle pour l'analyse de l'information ?*, rapport technique LIBD, Université de Namur, 2002b (disponible sur <http://www.info.fundp.ac.be/libd>).
- HAINAUT, J.-L., *Introduction aux SGBD CODASYL DBTG 71*, publication PReCISE/LIBD, université de Namur, septembre 2003. (disponible sur <http://www.info.fundp.ac.be/libd>).
- HAINAUT, J.-L., *Bases de données et modèles de calcul - Outils et méthodes pour l'utilisateur*, 4^e édition, Dunod, 2005.
- HAINAUT, J.-L., The transformational approach to database engineering, in [Lämmel, 2006], pp. 89-138, 2006.
- HAINAUT (a), J.-L., Network Data Model, in *Encyclopedia of Database Systems*, Özsu, T., Ling, L. (Eds), Springer, 2009 (disponible sur <http://www.info.fundp.ac.be/libd>).
- HAINAUT (b), J.-L., Hierarchical Data Model, in *Encyclopedia of Database Systems*, Özsu, T., Ling, L. (Eds), Springer, 2009b (disponible sur <http://www.info.fundp.ac.be/libd>).
- HAINAUT, J.-L., *Introduction to Database Reverse Engineering*, PReCISE Research Center (LIBD), université de Namur, 2009c. (disponible sur <http://www.info.fundp.ac.be/libd>).
- HAINAUT, J.-L., *Interpretation of foreign keys in database reverse engineering*, PReCISE Research Center (LIBD), université de Namur, 2009d. (disponible sur <http://www.info.fundp.ac.be/libd>).
- HALPIN T., *Conceptual Schema and Relational Database Design*, Prentice-Hall, 1995.
- HALPIN, T., MORGAN, T., *Information Modeling and Relational Databases*, Morgan Kaufman, 2008.
- HAREL, D., RUMPE, B., *Modeling Languages: Syntax, Semantics and all that Stuff (or, What's the Semantics of "Semantics"?)*, Technical paper, Weizmann Institute of Science, 2004. Une synthèse dédiée à UML est parue sous le titre *Meaningful modeling: what's the semantics of "semantics"?* dans *Computer*, 37(10), pp. 64-72, 2004.
- HARTMANN, S., Coping with Inconsistent Constraint Specifications, in *Proc. of the International Conference on the Entity-Relationship Approach (ER 2001)*, LNCS 2224, pp. 241-255, Springer-Verlag, 2001.
- HAY, D., *Data Modeling Patterns - A Metadata Map*, Morgan Kaufmann, 2006.
- HENDERSON-SELLERS, B., BARBIER, F., Black and White Diamonds, *Proc. UML '99*, LNCS 1723, Springer, 1999.
- HEWITT, E., *Cassandra: The Definitive Guide*, O'Reilly, 2010.
- JACOBSON, I., CHRISTERSON, M., JONSSON, P. et ÖVERGAARD, G., *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
- JAJODIA, S., NG, P., Translation of Entity-Relationship Diagrams into Relational Structures, *Journal of System and Software*, Vol. 4, 1984.

JCM, *Journal of Conceptual Modeling*, 2008, <http://www.inconcept.com/index.html>.

KNUTH, D., *The art of computer programming, Volume 3: Sorting and Searching*, Second Edition, Addison-Wesley, 1998.

KOSHAFIAN, S., CHAN, A., WONG, A. et WONG, H., *Client/Server SQL Applications*, Morgan Kaufmann, 1992.

LARMAN, C., *UML 2 et les design patterns*. LNCS 4143, Pearson, 2005.

LÄMMEL, R., SARAIVA, J., VISSER, J. (eds), *Generative and Transformational Techniques in Software Engineering*. LNCS 4143, Springer-Verlag, 2006.

LEE, J., ELMASRI, R., WON, J., An integrated temporal data model incorporating time series concept, *Data & Knowledge Engineering*, vol. 24, pp. 257-276, Elsevier, 1998.

LIGHTSTONE, S., TEOREY, T., NADEAU, T., *Physical Database Design: The Database Professional's Guide to Exploiting Indexes, Views, Storage, and More*, Morgan Kaufmann, 2007.

LOOMIS, M., *The Database Book*, MacMillan, 1988.

LUKOVIC, I., MOGIN, P., PAVICEVIC, J., RISTIC, S., An approach to developing complex database schemas using form types, *Software Practice and Experience*, 37, pp. 1624-1656, 2007.

MAIER, D., *The Theory of Relational Databases*, Computer Science Press 1983. (disponible au format PDF dans *ACM - SIGMOD Anthology CD-ROM*, Vol 4, Issue 1).

MARCHAND, M., *Outils mathématiques pour l'informaticien*, De Boeck Université, 2005.

MARCOS, E., VELA, B., CAVERO, J.M., Extending UML for object relational database design, in *Proc. of UML 2001*, LNCS 2185, pp. 225-239, Springer, 2001.

MCBRIEN, P., POULOVASSILIS, A. Data integration by bi-directional schema transformation rules, *Proc 19th International Conference on Data Engineering (ICDE'03)*, IEEE Computer Society Press. (2003).

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.

MOK, W., Designing nesting structures of user-defined types in object-relational databases, *Information and Software Technology*, n° 49, pp. 1017-1029, Elsevier, 2007.

MOODY, M., Graphical Entity Relationship Models: Towards a More User Understandable Representation of Data, *Proc. ER'96*, (1996).

MOODY, M., Metrics For Evaluating the Quality of Entity Relationship Models, *Proc. ER'98*, (1998).

MORK, P., BERNSTEIN, P., MELNIK, S., Teaching a Schema Translator to Produce O/R Views, *Proc. of ER 2007 Int. Conf.*, LNCS 4801, pp. 102-119, Springer, 2007.

NANCY, D. et ESPINASSE, B., *Ingénierie des systèmes d'information Merise, Deuxième génération*, Sybex, 1996.

NOY, N., *Semantic integration: A survey of ontology-based approaches*. ACM SIGMOD Record, 33(4), pp. 65-70, 2004.

OMG, *Request for Proposal - Information Management Metamodel*, OMG Document: ab/2005-12-02, 2005, <http://www.omg.org/docs/ab/05-12-02.pdf>.

OMG, *UML 2.0 OCL Specification*, May 1, 2006, <http://www.omg.org>.

OWENS, K., *Programming Oracle - Triggers and Stored Procedures*. Prentice Hall, 2004.

PARENT, C., SPACCAPIETRA, S. Issues and approaches of database integration. *Communications of the ACM*, 41(5), pp. 166 - 178, May 1998.

PARENT, C., SPACCAPIETRA, S., ZIMÁNYI, E., *Conceptual Modeling for Traditional and Spatio-Temporal Applications: The MADS Approach*, LNCS n° 3534, Springer, 2006.

PAVLO, A., ASLETT M. *What's Really New with NewSQL?*, SIGMOD Record, June 2016 (Vol. 45, No. 2) [<http://db.cs.cmu.edu/papers/2016/pavlo-newsq-sigmodrec2016.pdf>]

PATTON, N.(Ed.), *Active Rules in Database Systems*, Springer-Verlag, Monographs in Computer Science, 1999.

PIROTTE, A., ZIMANYI, E., MASSART, D., Yakusheva, T., Materialization: a powerful and ubiquitous abstraction pattern, *Proc. 20th VLDB Conf.*, 1994.

PROPER, H., Data Schema Design as a Schema Evolution Process, *Data & Knowledge Engineering*, 22(2), pp. 159-189, 1997.

RAHM, E., BERNSTEIN, P. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4), pp. 334-350, Springer, 2001.

REESE, G., *JDBC et JAVA, guide du programmeur*, O'Reilly France, 1998.

ROCHFELD, A., NEGROS, P., Relationship of relationships and other inter-relationship links in E-R model, *Data & Knowledge Eng.*, 9, 1993, pp. 205-221.

ROSENTHAL, A., REINER, D., Tools and Transformations - Rigorous and Otherwise - for Practical Database Design, *ACM TODS*, 19(2). (1994).

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F. et LORENSEN, F., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

RUMBAUGH, J., JACOBSON, I., BOOCH, G., *The Unified Modeling Language*, Reference Manual, 2nd Ed., Addison Wesley, 2005.

SADALAGE, J., FOWLER, M., *NoSQL Distilled*, Addison-Wesley, 2013.

SCHEK, H., SHOLL, M., The relational model with relation-valued attributes, *Information Systems*, 11(2):137-147, 1986.

SEVERANCE, D., G., LOHMAN, G., M. Differential. Files: Their Application to the Maintenance of Large Databases, *ACM TODS*, 1:3, September 1976

SHASHA, D., BONNET, P., *Database Tuning*, Morgan Kaufmann, 2002.

SHVAIKO, P., EUZENAT, J., A Survey of Schema-Based Matching Approaches, *Journal on Data Semantics IV*, LNCS 3730, pp. 146-171, Springer, 2005.

SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., *Database System Concepts*, McGraw-Hill, 2011.

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.

SMITH, J., M., SMITH, D., C., P., Database Abstractions: Aggregation and Generalization, *ACM TODS*, Vol. 2, No. 2, 1977.

- SNODGRASS, R., T., *Developing Time-Oriented Database Applications in SQL*, Morgan-Kaufmann, 2000.
- SOUTOU, C., *UML 2 pour les bases de données*, Eyrolles, 2007.
- SPACCAPIETRA, S. et PARENT, C., View Integration : A Step Forward in Solving Structural Conflicts, *IEEE Trans. on Knowledge and Data Engineering*, October 1992.
- STOREY, V., PURAO, S., Understanding Relationships: Classifying Verb Phrase Semantics, in *Proc. ER 2004*, LNCS 3288, pp. 336–347, Springer, 2004.
- TARDIEU, H., ROCHFELD, A., COLLETTI, R., *La méthode Merise : Principes et outils*, Editions d'Organisation, 2000
- TEOREY, T., YANG, D., FRY, J., A logical design methodology for relational databases using the extended entity-relationship model, *ACM Comput. Surv.*, 18(2), 1986.
- THALHEIM, B., Fundamentals of cardinality constraints, in *Proc. of the 11th International Conference on the Entity-Relationship Approach*, LNCS 645, Karlsruhe, Germany, 1992, pp. 7-23.
- THEODOULIDIS, C.I., LOUCOPOULOS, P., WANGLER, B., A Conceptual Modelling Formalism for Temporal Database Applications, *Information Systems*, 16(4), 1991.
- ULLMAN, J., *Database and Knowledge-base systems (Volume 1)*, Computer Science Press, 1988.
- UML, *OMG Unified Modeling Language Specification*, Version 1.3 alpha R2, OMG Inc., 1999.
- VAISMAN, A., ZIMÁNYI, E., *Data Warehouse Systems: Design and Implementation*, Springer, 2014
- VIDAL P. et PLANEIX P., *Systèmes d'information organisationnels*, Pearson Education, 2005.
- VIGNA, S., Reachability Problems in Entity-Relationship Schema Instances, in *Proc. ER 2004 Conference*, LNCS 3288, Springer, pp. 96-109, 2004.
- VILZ, J., BROGNEAUX, A-F., RAMDOYAL, R., ENGLEBERT, V., HAINAUT, J-L., Data conceptualization for web-based data-centered application design, in *Proc. CAiSE 2006*, Springer, pp. 205-219, 2006.
- WAGNER, C., W., Implementing Abstraction Hierarchies, in *Proc. of the 7th Int. Conf. on ERA*, 1988, North-Holland, 1989.
- ZANIOLO, C., A new normal form for the design of relational database schemata, *ACM Transactions on Database Systems*, 7(3), pp. 489-499, 1982.

Index

A

- acyclique 479, 493, 557
- add
 - check 306
 - column 216, 280
 - constraint 217, 306
 - foreign key 217, 281
 - primary key 217, 281
 - unique 217
- administrateur 313
- ADO.NET 508
- AGL 57, 378, 399, 446, 458, 508, 532, 595, 644, 689, 710
- agrégation
 - cluster 65, 163
 - clustering index 162
 - MapReduce 358
 - UML 522
- algèbre
 - jointure 74
 - projection 74
 - relationnelle 73
 - sélection 75
- alias
 - de colonne 227
 - de table 237
- alter table 216
- analyse
 - conceptuelle 403, 533
 - de documents 547
 - de textes 406, 536
- and 226
- aspect (programmation orientée-) 702
- association (UML) 516
 - agrégation, composition 522
 - binaire 516
 - classe 522
 - n-aires 517
 - qualifiée 520
- attribut 381
 - complexe 466, 610
 - composé 465

- de classe 519
 - de relation 72
 - de type d'associations 470
 - de type d'entités 434
 - facultatif 381
 - multivalué 98, 417, 465
 - obligatoire 381
 - parent de 468
- attribut racine 86, 88
- attribut relationnel
 - externe 79
 - interne 79
 - isolé 79
 - racine 79
 - source 79
- autoincrément 210

B

- Bachman 206
- bande magnétique 117
- base de données
 - active 313, 372
 - distribuée 196, 349
 - fédérée 199
 - multibase 199
 - NoSQL 345
 - optimisation 661
 - orientée objet 47
 - quantique 174
 - relationnelle 38, 50, 72
 - relationnelle objet 48
 - sans schéma 345
 - temporelle 313
 - XML 48
- between 225
- BigData 42, 346
- BitCoin 169, 366
- BLOB 210
- blockchain 361

C

- cache 111

- Call of Duty 351
- cardinalité
 - d'attribut 381, 468
 - de rôle 386, 460
- catalogue
 - de base de données 185, 313, 501
 - disque 124
- chaîne de blocs 361
- check 305, 677
- checkpoint 195
- Chen P. 46, 507
- chiffrement 364
- classe d'objets
 - générique 497
 - UML 516
- classe fonctionnelle 383
 - plusieurs-à-plusieurs 384, 438
 - un-à-plusieurs 383, 434
 - un-à-un 384, 437
- clé étrangère 53, 82, 212
 - composite 53, 56, 235, 240, 260
 - cyclique 60, 253, 278
 - delete et update modes 61, 213
 - facultative 54
 - non minimale 96, 624, 627
 - partiellement facultative 214
- clé USB 115
- client-serveur 184
- CLOB 210
- close 320
- cluster 163, 349
- clustering index 162
- Codd 72
- code auto-correcteur 114
- cohérence finale 349, 350
- cohérence forte 349
- collation 230
- collision 169, 171
- colonne 50
 - ajouter 216
 - autoincrémentée 171
 - facultative 52
 - modifier 216
 - obligatoire 52, 58
 - supprimer 216
- commit en deux phases 200
- Common Table Expression 294
- composition
 - UML 522
- conception logique
 - relationnelle 433, 605
 - relationnelle objet 640
- conception physique 653
- condition
 - d'association 233, 258
 - de jointure 247
 - de non-association 235, 258
 - sur n-uplet 226
- construction
 - implicite 703
 - incohérente 558
 - non initialisable 557
 - non satisfiable 554
- contrainte 673
 - additionnelle 490
 - ajouter 216
 - cyclique 492, 678
 - d'existence 54, 487
 - d'inclusion 82
 - d'intégrité 58, 486
 - d'unicité 53, 60, 392
 - de domaine 487
 - de redondance 494, 679
 - dépendance fonctionnelle 63, 489
 - dynamique 487, 682
 - langage d'expression 490
 - OCL 524
 - référentielle 54, 60, 82, 306, 435
 - statique 486
 - supprimer 216
 - sur graphe d'instances 493, 679
 - UML 523
- contrat intelligent 371
- contrôle d'accès 183, 287
 - inférence statistique 202
 - modèles 201
- correction d'un schéma 552
- CQL 352, 360
- create
 - dbspace 218
 - domain 210
 - index 218
 - procedure 307
 - role 289

- schema 208
- table 209, 301
- trigger 308
- type (UDT) 300
- unique index 218
- view 290
- critère de groupement 266
- cross join 251
- CSV 161
- CTE 294
- CTE recursive 294
- curseur 320
- cyclique
 - clé étrangère 253
 - contrainte 492, 678
 - structure de données 253, 256, 278
 - type d'associations 386

D

- Data Definition Language (DDL) 208
- Data Manipulation Language (DML) 219
- DB-MAIN 57
 - analyse de schéma 597
 - conception logique 644
 - génération de code 690
 - intégration de schémas 597
 - méthodes 458
 - rétro-ingénierie 710
 - schéma conceptuel 399, 508
 - schéma logique 644
 - schéma UML 532
 - transformation de schéma 595, 644
- declare cursor 320
- déclencheur 308, 674
 - instead of 294, 686
 - déclencheur récursif 312
- décomposition
 - d'un type d'associations 505, 562, 573
 - d'un type d'entités 426, 572
 - d'une contrainte d'inclusion 83
 - d'une dépendance fonctionnelle 80
 - d'une relation 89
 - d'une table 63, 339
- default 211, 339, 680
- delete 276, 321
- delete et update mode
 - cascade 213

- no action 213
- restrict 213
- set null 213
- dénormalisation 665
- dépendance fonctionnelle 63, 77
 - anormale 92, 427
 - circuit 82
 - conceptuelle 489
 - de base 81
 - dérivée 81
 - externe 79, 92
 - fermeture 81
 - graphe ADF 79
 - interne 79
 - minimale 80
 - propriétés 79
 - triviale 79
- dépendance multivaluée 100
- déterminant 63
- déterminé 63
- dictionnaire
 - de données 313
 - de valeurs 155
- discriminateur 480, 516
- disque électronique 115
- disque hybride 117
- disque magnétique
 - adressage LBA 109
 - cylindre 109
 - définition 107
 - lecture anticipée 111
 - modèle de référence 113
 - piste 108
 - secteur 107
 - tampon 111
- distinct 223, 300
- distribution des traitements 200
- document 355
- domaine
 - d'application 55, 375, 380
 - de valeurs 50, 209, 615
 - relationnel 72
- données
 - décisionnelles 42
 - groupées 265
 - instables 189
- drop

- column 216, 280
- constraint 217, 307
- domain 216
- index 218
- table 216, 281
- type 300
- view 291

E

- éclatement
 - page 137
 - table 662
- enregistrement 34
- ensemble 72
- ensemble vide 233, 239, 338
- entité 50, 380
- espace de stockage 65, 167, 442, 661
 - créer 218
 - organisation 120
- Ethereum 370
- évaluation d'un schéma 575
- exec SQL 319

F

- fenêtre d'incohérence 350
- fetch 320
- fichier 34
 - accès par clé 146, 155
 - accès par index 135
 - calcul d'adresse 147
 - calculé 146
 - débordement 149
 - lecture par index 140, 161
 - lecture par intervalle 141, 158
 - lecture séquentielle 125, 161
 - séquentiel indexé 134
 - tri 128
- fichier différentiel 169, 195
- filtre de Bloom 169
- fonction agrégative 231, 266, 299, 332
- fonction injective 169, 170
- fonctions SQL 227
- foreign key 212
- Formal Concept Analysis 569
- forme normale
 - 1FN 72, 92, 359
 - 3FN 102

- Boyce-Codd 102
- FNCE 102
- N1FN 97
- partitionnée 100
- fragmentation
 - base de données 349, 358
 - disque 123
 - enregistrement 123
 - fichier 123, 127
 - horizontale 198, 663
 - verticale 198, 662
- from 221, 272
- fusion
 - fichier 128
 - page 137

G

- génération de code 444, 671
- grant 288
- grant option 288
- graphe ADF 79
- group by 265
- group_concat 266, 299
- groupe de lignes 265
- groupement multiniveaux 270

H

- hachage cryptographique 170, 363
- Hadoop 201
- having 267
- héritage 474
 - ascendant 475, 627, 685
 - collatéral 476
 - descendant 475, 626, 685
 - matérialisation 622
 - multiple 477, 629
 - répartitions multiples 479
 - UML 516
- heuristiques d'erreurs 578
- hiérarchie de mémoires 118
- homonyme 422

I

- IDA 507, 598
- identifiant
 - calcul 85
 - composite 52, 631

- composition 390
- d'attribut complexe 486
- de jointure 264
- de relation 76
- de table 52, 60, 439
- de type d'associations 483
- de type d'entités 388, 481
- facultatif 54, 218
- global 626
- hybride 389, 523
- implicite 392, 483, 484
- minimal 53, 391
- non minimal 625
- primaire 52, 76, 212
- secondaire 52, 212
- technique 171
- universel (UUID) 171, 363
- `if exists` 216
- `in` 225, 333
- index 65, 442, 656
 - augmenté 218
 - bitmap 159
 - calculé 146
 - clustering 163
 - créer 217
 - expression 218
 - fragmentation 168
 - liste de pointeurs 155
 - partiel 159, 218, 356
 - primaire 133
 - secondaire 134, 154
 - secondaire augmenté 159, 168
 - supprimer 218
 - sur expression 159
- inférence structurelle 577
- information incomplète 331
- Information Schema 316
- injection de code SQL 329
- `inner join` 252
- `insert` 321
 - `select` 275
 - `values` 274
- instantané 275, 291
- intégration de schémas 544, 579
- intégrité d'entité 53
- interface SQL
 - dynamique classique 323

- JDBC 326
- SQLJ 321
 - statique classique 318
- `is null` 224, 333
- `is-a` 472, 516, 554

J

- Javascript 356
- `join` 246
- jointure 246
 - auto-jointure 255
 - de deux tables 246
 - de plus de deux tables 247
 - externe 248, 259, 685
 - généralisée 260
 - interne 248
 - naturelle 248
 - procédurale 704
 - jointures
- JSON 100, 355, 356, 360

L

- LBA 109
- lecture
 - anticipée 111
 - par index 151
 - séquentielle 151
- ligne 50
- ligne célibataire 248
- `like` 225
- `limit` 273
- `list_agg` 300
- logique ternaire 226, 332

M

- machine de bases de données 118
- MapReduce 198, 201, 358
- masque SQL 225
- `match` 214
- mémoire flash 115, 117, 118
- mémoire ADN 172
- mémoire atomique 173
- Merise 507, 527, 599
- métadonnées 360, 500
- métaschéma 500
- méthode de développement
 - en cascade 456

- itérative 456
- méthodes et opérations
 - UML 519
- minage 367
- mirroring 113
- Model Driven Engineering (MDE) 647
- modèle
 - classes UML 514
 - clé-valeur 346
 - de graphes 348
 - orienté colonnes 347, 351
 - orienté documents 347, 355
 - orienté objet 349
 - RDF 299, 348
 - relationnel 49, 72, 607
- modèle de distribution 198, 349, 355, 358
- modèle Entité-association 379, 459
- Modèle Entité-relation 507
- Modèle Individuel 507
- modèle ORM 507, 599
- modify (colonne) 217
- MOF 530
- moissonneuses-batteuse 527
- monotonie 128
- multi-ensemble 76, 299, 466, 524, 525, 641
- multiplicité (UML) 516

N

- natural join 251
- nest 98
- NewSQL 358
- nom d'objet 494, 632
- nomenclature de produits 43, 256
- nonce 367
- normalisation
 - construction non minimale 560
 - d'un schéma conceptuel 425, 558, 708
 - d'un type d'associations 573
 - d'un type d'entités 572
 - d'une relation 90
 - d'une table 62
 - expressivité insuffisante 564
 - théorie 71
- NoSQL 48
- not 226
- not in 225, 333
- not null 211, 333

- noyau irréductible 87, 95
- null 51, 100, 233, 239, 331
- nulls first 217

O

- O/RM (Object/Relational mapping) 42, 675, 689
- OCL 490
- offset 273
- OLAP 204, 361
- OLTP 204, 360
- ontologie 348, 406
- open 320
- opérateur ensembliste
 - except 250
 - intersect 249
 - union 248
 - union all 249
- opérateur logique
 - conjonction (ou) 226
 - disjonction (et) 226
 - équivalence 226
 - implication 226
 - négation 226
 - ou exclusif 226
 - quantificateur 241
- or 226
- order by 230, 336
- outer join 248

P

- paraphrase 576
- partitionnement d'un réseau 199, 351
- plan de transformation 636, 708
- positioned update 321
- prédicat
 - assertion 305
 - de colonne 306
 - de table 306, 677
- prédicteur à hachage 169, 195
- présentation graphique 55, 575
- preuve d'enjeu 371
- preuve de travail 368
- primary key 212
- privilège 288
- procédure SQL 307
- processus de conception 450

produit relationnel 250, 251
programme d'application 316
proposition
 binaire 406
 contradictoire 423
 élémentaire 406
 existentielle 536
 générale 408
 irréductible 408
 particulière 408
 réductible 411
 relationnelle binaire 537
 relationnelle n-aire 540
 unaire 406
propriété multivaluée 417, 465
protocole à quorum 350
prototypage 577
Python 68, 356

Q

quantificateur
 all 240, 334
 any 240, 334
 existentiel 241
 exists 239, 334
 not exists 239, 334
 universel 241
QUEL 47

R

RAID 113
redondance 664
 interne 71, 426, 572
 structurelle 421
 structurelle d'instances 570
Reed-Solomon 114
référence en avant 217
relation 72
relation is-a 301, 472, 516, 620, 683
réplication des données 198, 349, 358
requête
 exécution 185
 récursive 256, 294
rétro-ingénierie 693
 conceptualisation 705
 extraction physique 698
 reconstruction logique 699

revoke 289
rôle 382
 cardinalité 470
 contrôle d'accès 289
 de cardinalité 1-N 633
 facultatif 385
 multitypes 469, 618
 obligatoire 385

S

scalability 344
schéma
 conceptuel 379, 403, 452, 712
 de tables 55, 433, 713
 logique 452, 606
 physique 452, 653
script SQL 67
secure hashing 170, 363
select 220, 271
 * 222
 colonnes 221
 distinct 223
 données dérivées 227
 for upate 277
sémantique
 du modèle de classes UML 529
 du modèle Entité-association 503
 externe 427, 461
 interne 503
SEQUEL 47
SFW 220
SGBD 34, 40, 181, 654
 architecture 184
 backup 193
 Cassandra 169, 351
 CODASYL 43, 557
 CouchBase 356
 DB2 47, 69, 204, 205, 355
 DBOM 43
 Firebird 68, 205
 IMS 44, 204
 Ingres 47
 in-memory 204, 360
 InterBase 68, 208
 journal 193
 MariaDB 205
 MongoDB 355

- MS Access 47, 56, 68, 205, 355
 - MySQL 69, 205, 355
 - NoSQL 345
 - Open Source 205
 - Oracle 47, 69, 204, 205, 355
 - PostgreSQL 47, 69, 169, 205, 355, 358
 - protection contre les accidents 193
 - Redis 204, 346
 - reprise à chaud 194
 - reprise à froid 194
 - Socrate 46, 507
 - SQL Server 69, 167, 204, 205, 218, 355
 - SQLite 68, 204, 355
 - Sybase 205
 - System/R 47
 - VoltDB 204, 349, 358
 - XML 348, 359
 - SHA-256 171, 363
 - shard 349
 - signature 170
 - simulateur
 - séquentiel indexé 143
 - tri 132
 - snapshot 221, 275
 - sous-requête 234, 258
 - SQL 38
 - array 298
 - DDL 207, 671
 - DML 219, 245
 - extensions objet 297
 - normes 206
 - requête 207
 - row 298
 - SQL3 640
 - table typée 300
 - TDU 300, 642
 - SQL3 359
 - SQLfast 67, 218, 316, 331, 458, 577
 - SSD 115
 - stabilité des adresses 154
 - stéréotype
 - Entité-association 464
 - UML 515
 - Stonebraker 358
 - stored procedure 307
 - striping 114, 116
 - synonyme 422
 - système d'information 449
- ## T
- table
 - fusion 663
 - partitionnement 662
 - relationnelle 50
 - typée 642
 - table statique 253, 275
 - tag 515
 - tampon 111
 - taux
 - d'éclatement 141
 - d'occupation 138
 - de chargement 143, 150, 168
 - de fusion 142
 - de rééquilibrage 142
 - de rupture 140
 - temporary table 209
 - théorème CAP 351
 - transaction
 - ACID 186, 197, 344, 360
 - BitCoin 366
 - chaîne de blocs 362
 - concurrence 188
 - définition 186
 - distribuée 197
 - données instables 189
 - exécution entrelacée 189
 - exécution séquentielle 189
 - exécution sérialisable 191
 - interblocage 191
 - mise à jour perdue 189
 - MVCC 192
 - niveaux d'isolation 192
 - read committed 192
 - read uncommitted 192
 - repeatable read 192
 - verrouillage 190
 - verrouillage en 2 phases 191
 - WAL 195
 - transformation de schéma 414, 458, 595, 607, 634
 - treillis de Galois 569
 - tri
 - externe 128
 - interne 130

- trigger 308, 676
- type d'associations 382, 434, 468
 - classe fonctionnelle 468
 - complexe 618
 - cyclique 439
 - de composition 497
 - de matérialisation 498
 - fonctionnel 617
 - n-aire 468
 - taille de 470
- type d'entités 380, 434, 472
 - association 425, 564, 571
 - attribut 425, 560

U

- UDF 227
- UML 513
 - DB-UML 527
 - profil 516
- unique 212
- unknown 332
- unnest 98
- update 276, 321
- UUID 171

V

- valeur par défaut 211
- vue
 - conceptuelle 638, 688
 - matérialisée 275
 - modifiable 685
 - relationnelle 290

W

- wear leveling 116
- where
 - condition de totalité 241
 - condition composée 226
 - condition simple 222
- with 294
- with check option 293
- with recursive 294

X

- XML 44, 48, 100, 167, 347, 348, 387, 501, 548, 590
- XOR 523
- XPath 348, 490

Z

- zero-table select 253