

Bases de données

Yamine Aït-Ameur
ENSEEIHT
yamine@enseeiht.fr

Christophe Garion
ISAE-SUPAERO
christophe.garion@isae-supaero.fr

16 janvier 2017

Table des matières

1. Introduction	3
2. Modèle de Chen	5
2.1. Généralités sur l'information et sur sa représentation	5
2.2. Notions de base	5
2.3. Structure des types	6
2.3.1. Type d'entités	6
2.3.2. Type d'attribut	7
2.3.3. Type d'association	7
2.3.4. Représentation par diagramme	7
2.4. Notion d'occurrence	7
2.5. Représentation des concepts	8
2.5.1. Notion de clé et de clé primaire	9
2.5.2. Cardinalité de rôle	9
2.5.3. Cas particulier des classes d'associations binaires	11
2.6. Diagramme entité-association	12
2.7. Dépendances entre attributs	14
2.8. Quelques propriétés du modèle entité-association	14
2.8.1. Bases de données	14
2.8.2. Non redondance de l'information	14
2.8.3. Extension de la notion de clé d'une classe d'associations	15
2.9. Entités faibles	17
2.10. Exemple : gestion d'un parc informatique	18
2.11. Le langage de modélisation de données EXPRESS	21
2.11.1. Les entités	21
2.11.2. Les instances	22
2.11.3. Les contraintes	23
3. Stockage des informations	27
3.1. Généralités sur le stockage des données	27
3.1.1. Mémoires à accès aléatoire	27
3.1.2. Mémoires à accès séquentiel	28
3.2. Hiérarchie de composants de stockage dans un ordinateur	31
3.2.1. Mémoires volatiles : le cache et la mémoire principale	31
3.2.2. Système de stockage secondaire	32
3.2.3. Système de stockage tertiaire	34
3.3. Impact du stockage secondaire	35

Table des matières

3.4.	Représentation des données	36
3.4.1.	Stockage des attributs	36
3.4.2.	Stockage des enregistrements	38
3.4.3.	Accès aux données	38
3.4.4.	Ajout, destruction et mise à jour d'enregistrements	39
3.4.5.	Collection d'enregistrements et index	40
3.5.	Conclusion	41
4.	Apports des bases de données et SGBD	43
4.1.	Bases de données	43
4.2.	Système de Gestion de Bases de Données	44
4.2.1.	Structure à trois niveaux	44
4.2.2.	Langages associés aux bases de données	45
4.3.	Caractéristiques des SGBD	46
4.3.1.	Objets, relations et schémas	46
4.3.2.	Les fonctions de base	46
4.3.3.	Indépendance des données et des programmes	46
4.3.4.	Non redondance	47
4.3.5.	Partageabilité	47
4.3.6.	Efficacité des échanges	47
4.3.7.	Cohérence	47
4.3.8.	Sécurité et fiabilité	47
4.4.	Les composants d'un SGBD	47
4.5.	Conclusion et remarques	48
5.	Modèles classiques de BD	51
5.1.	Le modèle hiérarchique	51
5.1.1.	Contexte et définitions	51
5.1.2.	Techniques de transformation entre le niveau logique et le niveau physique	53
5.1.3.	Caractéristiques du modèle hiérarchique	55
5.1.4.	Définition du modèle hiérarchique	55
5.1.5.	Limitations du modèle hiérarchique	55
5.1.6.	Réalisations techniques	56
5.2.	Le modèle réseau	57
5.2.1.	Les spécifications du CODASYL	57
5.2.2.	Description	58
5.3.	Définitions	58
5.3.1.	Réalisations techniques	59
5.4.	Exemple d'un SGBD de type DBTG	60
5.4.1.	Les notions de base	60
5.4.2.	Le langage de définition des données	61
5.4.3.	Généralités sur le langage de manipulation des données	61
5.4.4.	Exemple de construction de modèle et LDD	61

5.5. Exemples de manipulation de données	63
5.5.1. Manipulation des données	63
5.5.2. Quelques mots clés du langage de manipulation de données	64
6. Le modèle relationnel	69
6.1. Description générale	69
6.2. Un exemple intuitif	69
6.3. Définitions	71
6.3.1. Domaine	71
6.3.2. Relations	71
6.3.3. Attributs et clés	73
6.3.4. Schéma relationnel d'une base de données	73
6.4. Conception du schéma relationnel	75
6.4.1. Principe initial	75
6.4.2. Simplification du schéma	76
6.4.3. Règles de traduction	76
7. Algèbre relationnelle	81
7.1. Principe et définitions	81
7.2. Opérateurs ensemblistes	82
7.2.1. Union	82
7.2.2. Intersection	82
7.2.3. Différence	83
7.3. Projection	83
7.4. Sélection	83
7.5. Produit cartésien	84
7.6. Jointures	85
7.6.1. Jointure naturelle	85
7.6.2. θ -jointure	86
7.7. Opérateur de renommage	87
7.8. Opérations de mise à jour	87
7.9. Représentation d'une requête	88
7.9.1. Représentation sous forme d'arbre	88
7.9.2. Notation linéaire	88
7.10. Opérateurs étendus	89
7.10.1. Opérateurs d'agrégation	89
7.10.2. Regroupements	90
7.10.3. Jointures externes	90
7.11. Conclusion	91
8. SQL (Structured Query Language)	93
8.1. Historique et présentation	93
8.2. Le langage de manipulation de données	94
8.2.1. Bloc de qualification	94

8.2.2.	Projection	95
8.2.3.	Sélection	96
8.2.4.	Entêtes de colonnes	97
8.2.5.	Tris des colonnes	97
8.2.6.	Opération de produit	98
8.2.7.	Noms d’alias	98
8.2.8.	Jointures	99
8.2.9.	Opérateurs ensemblistes	101
8.2.10.	Opérateurs d’agrégation	102
8.2.11.	Regroupements	102
8.2.12.	Clause HAVING	103
8.2.13.	Sous-requêtes	104
8.2.14.	Insertion	105
8.2.15.	Modification de tuples	106
8.2.16.	Suppression	106
8.3.	Le langage de description de données	107
8.3.1.	Types de données	107
8.3.2.	Création de table	107
8.3.3.	Destruction d’une relation	108
8.3.4.	Modification du schéma d’une relation	108
8.3.5.	Index	109
8.3.6.	Vues externes	109
8.4.	Le langage de contrôle des données	112
8.4.1.	Utilisateurs d’une base de données	112
8.4.2.	Privilèges	112
8.4.3.	Création de privilèges	112
8.4.4.	Retrait de privilèges	113
8.4.5.	Utilisation des vues	115
9.	Contraintes et triggers	119
9.1.	Clés primaires et clés étrangères	119
9.1.1.	Clés primaires	119
9.1.2.	Contrainte UNIQUE	120
9.1.3.	Clés étrangères	121
9.1.4.	Politique de gestion des clés étrangères	122
9.2.	Contraintes sur les attributs et les tuples	123
9.2.1.	Contrainte NOT NULL	124
9.2.2.	Contrainte CHECK sur un attribut	124
9.2.3.	Contrainte CHECK sur un tuple	125
9.2.4.	Modification de contraintes	126
9.3.	Assertions et <i>triggers</i>	127
9.3.1.	Assertions	127
9.3.2.	<i>Triggers</i>	128

10. SQL et autres langages	131
10.1. Introduction	131
10.2. SQL statique intégré avec C	132
10.2.1. Établissement d'une connexion	133
10.2.2. Production d'un exécutable	133
10.2.3. Partager des variables avec le langage hôte	135
10.2.4. Requêtes sur la base de données	135
10.3. JDBC et les architectures client/serveur	138
10.4. Fonctionnement	139
10.4.1. Enregistrement du driver JDBC	141
10.4.2. Établissement d'une connexion	141
10.4.3. Créer une requête SQL	142
10.4.4. Exécution de la requête	143
10.4.5. Traitement de la requête	143
10.5. Un exemple complet	144
10.6. Conclusion	146
11. Dépendances fonctionnelles	149
11.1. Objectifs	149
11.2. Dépendances fonctionnelles	150
11.2.1. Définition	150
11.2.2. Notion de conséquence	151
11.3. Règles et propriétés des dépendances fonctionnelles	151
11.3.1. Règle de séparation/combinaison	151
11.3.2. Dépendances fonctionnelles triviales	151
11.3.3. Dérivation syntaxique et règles d'inférence d'Armstrong	152
11.3.4. Calcul de la fermeture d'un ensemble d'attributs	154
11.3.5. Couverture irredondante ou minimale	156
11.4. Normalisation de relations	157
11.4.1. Décomposition de relations	157
11.4.2. Première et seconde forme normale	162
11.4.3. Troisième forme normale	162
11.4.4. Troisième forme normale de Boyd-Codd	163
11.4.5. Mise en œuvre	165
11.5. Dépendances multivaluées et normalisation associée	165
11.5.1. Définition	165
11.5.2. Propriétés des dépendances multivaluées	166
11.5.3. Normalisation des dépendances multivaluées	166
11.6. Conclusion	167
12. Gestion des transactions	169
12.1. Notion de transaction	169
12.1.1. Définition	169
12.1.2. Propriétés des transactions : les propriétés ACID	170

12.1.3. Opérations de base du gestionnaire de transactions	171
12.2. Reprise après panne	171
12.2.1. Journal	173
12.2.2. Un exemple simple de reprise : l' <i>undo-logging</i>	173
12.2.3. Reprise avec <i>undo-logging</i>	174
12.2.4. Notion de point de contrôle	175
12.2.5. Journalisation par <i>undo/redo</i>	176
12.2.6. Reprise après panne des supports	178
12.3. Gestion de la concurrence	179
12.3.1. Quelques problèmes de gestion de la concurrence	179
12.3.2. Séquentialité	180
12.3.3. Verrouillage	183
12.3.4. Blocage	187
12.3.5. Verrouillage intentionnel	188
12.4. Bases de données distribuées	190
12.5. Les fonctionnalités fournies par SQL	191
12.5.1. COMMIT et ROLLBACK	191
12.5.2. Transactions en lecture seule	192
12.5.3. Niveau d'isolation	192
12.6. Conclusion	193
13. Bases de données objets	195
13.1. Historique des bases de données orientées objets	196
13.2. Pourquoi des bases de données objets ?	197
13.2.1. Nature des applications	197
13.2.2. Gestion des données fortement structurées	197
13.2.3. Intégration dans un langage de programmation	197
13.2.4. Conversion de types ou <i>impedance mismatch</i>	198
13.2.5. Les interfaces de manipulation	198
13.3. Bases de données objets	198
13.3.1. Base de données objets : définition	199
13.3.2. Système de gestion de base de données objets	199
13.3.3. Schéma conceptuel d'une base de données à objets	200
13.3.4. Bases de données objets : deux approches	200
13.4. Caractéristiques principales des bases de données objets	201
13.4.1. Gestion de la persistance	201
13.4.2. Support des collections	203
13.4.3. Évolution des schémas	203
13.5. L'approche relationnel-objet	204
13.5.1. Évolution du relationnel à l'objet	205
13.5.2. Définition des objets	205
13.5.3. Les collections	208
13.5.4. Manipulation des objets	209

13.6. L'approche orientée objet	210
13.6.1. Le modèle à objets de l'ODMG	210
13.6.2. Définition d'objets avec ODL	213
13.6.3. Interface avec un langage : cas de C++	215
13.6.4. Manipulation des objets avec OQL	216
13.7. Exemples de systèmes de gestion de bases de données orientées objets	218
13.7.1. OPAL de GEMSTONE	219
13.7.2. Objectstore	219
13.7.3. Orion	219
13.7.4. O2	220
13.8. Conclusion	221
A. Utilisation du SGBD PostgreSQL	223
A.1. Respect du standard SQL	223
A.2. Petits détails de syntaxe et autres	224
A.3. Écriture de fonction et de <i>trigger</i>	224
A.3.1. Fonctions dans PostgreSQL	224
A.3.2. Syntaxe de l'écriture d'un <i>trigger</i>	226
A.4. pgAdmin	227
A.4.1. Connexion à une base de données	228
A.4.2. Écriture d'une requête SQL	229
A.5. Utilisation interactive via le terminal	230
A.5.1. Commandes réservées aux utilisateurs privilégiés	231
A.5.2. Le moniteur interactif	232
A.6. Utilisation avec Java	234

1. Introduction

Une entreprise ou une organisation doit conserver la trace d'un volume élevé d'informations. Ces dernières peuvent être, par exemple, les noms, les salaires des employés, des adresses de fournisseurs ou toute autre information utile et donc à consulter. Traditionnellement, différentes parties de ces informations sont conservées par différents départements et/ou différents individus, chacun ayant à sa charge les informations qu'il utilise le plus souvent. Lorsque l'on veut obtenir une information, il faut alors déterminer :

- l'endroit où elle est rangée ou stockée,
- s'adresser au département ou à la personne concernée,
- et enfin la rechercher parmi toutes les informations locales au département ou à la personne.

Souvent, plusieurs informations prises individuellement ne fournissent pas toujours les résultats souhaités. Nous devons utiliser ces différentes informations pour obtenir une information composée « plus riche en information ». Cela implique la navigation d'une information d'un département à un autre ou d'une personne à une autre.

Pour toutes ces raisons, une tendance s'est développée pour combiner toutes les informations d'une organisation dans une base de données intégrée. Le stockage des données y sera entièrement centralisé. Dans l'idéal, il n'existera qu'un exemplaire de chaque élément de données. Les mises à jour ne sont exécutées qu'une seule fois.

Une base de données sera donc un ensemble organisé et intégré de données. Elle correspond à une représentation fidèle des données et de leur structure avec le minimum possible de contraintes imposées par le matériel. Enfin, elle doit pouvoir être utilisée pour une application pratique sans duplication de données.

Le SGBD (Système de Gestion de Bases de Données) est le logiciel qui supporte une telle organisation. C'est en fait un ensemble de logiciels fournissant l'environnement pour décrire, mémoriser, manipuler et traiter des ensembles de données tout en assurant pour celle-ci la sécurité, la confidentialité et l'intégrité sachant qu'un grand nombre d'utilisateurs ayant des besoins variés interagit avec ces ensembles de données.

L'objectif de ce cours est d'introduire les différents concepts liés à l'étude des bases de données. Nous partons du modèle de Chen, qui permet de formaliser les concepts de base utiles en représentation de données, puis nous suivons l'évolution historique des bases de données depuis les anciens modèles tel le modèle hiérarchique (encore utilisé dans certains cas) jusqu'aux modèles les plus récents tels les bases de données orientées objets et les bases de données utilisant la logique comme formalisme de base (appelées bases de données déductives). Nous insistons particulièrement sur le modèle relationnel qui tend à devenir le modèle le plus utilisé et nous présentons le formalisme des dépendances fonctionnelles permettant de générer des modèles relationnels « propres ». Un langage support au modèle relationnel, SQL, est présenté en détail. Enfin, le lecteur trouvera

1. Introduction

également un chapitre sur les problèmes de transactions et de reprises après pannes.

2. Un modèle de représentation des informations : le modèle de Chen

2.1. Généralités sur l'information et sur sa représentation

En matière de représentation et de manipulation d'informations, il faut distinguer :

- le monde réel qui nous entoure,
- le monde perçu qui est une vision donnée ou une perception du monde réel,
- le représenté qui est une transcription de ce que nous percevons selon une certaine représentation.

L'information peut être définie comme un fait du monde perçu. Elle peut être représentée, modifiée et rediffusée.

Afin de pouvoir la représenter, l'information nécessite la définition :

- d'un *langage de représentation*. Il permettra d'exprimer toute information par des mots de ce langage. Ce langage peut être textuel (comme par exemple un langage de programmation type XML), graphique (comme UML) etc.
- d'une *sémantique* qui permet de définir le sens des mots du langage qui représentent de l'information. Cette sémantique doit permettre de lever les ambiguïtés et les non-sens sur les informations.

En général, la sémantique est représentée à l'aide d'un *modèle*. Ce modèle doit permettre de rattacher, à chaque information qui y est représentée, un sens. Nous retrouvons ici le problème classique de la méthodologie de conception d'applications informatiques.

Le modèle de Chen [9], encore appelé modèle entité-relation ou modèle entité-liaison, est un modèle de représentation des informations assez proche des conceptions classiques du monde réel, ce qui constitue le but recherché par la modélisation. Il a donné lieu à l'implantation de systèmes de gestion de bases de données commercialisés, et est également à l'origine de plusieurs prototypes de recherche. On continue à poursuivre les études de ce modèle. Notons par ailleurs que ce modèle peut être adapté à d'autres problèmes que celui de la mise en œuvre d'une base de données.

2.2. Notions de base

La représentation des informations dans le modèle de Chen permet de représenter une partie de la connaissance structurelle (entités et associations) et de la connaissance descriptive (attributs simples). Elle utilise trois concepts de base que nous allons définir intuitivement dans ce qui suit :

- l'*entité* qui correspond à ce qui dans le monde réel possède une existence effective

2. Modèle de Chen

et autonome. Dans plusieurs cas, elle correspond à un nom dans le texte décrivant le problème à traiter (cahier des charges).

- l'*attribut* qui est une propriété qui n'a pas d'existence propre. Il est souvent lié à une entité pour laquelle il représente une certaine propriété.
- l'*association* (lien, liaison, relation) qui permet d'associer plusieurs entités entre elles. Dans plusieurs cas, elle correspond à un verbe dans le texte décrivant le problème à traiter. Il ne faut pas la confondre avec le concept de relation que nous introduirons lors de l'étude du modèle relationnel (cf. chapitre 6).

Enfin, on définira également un certain nombre de *domaines de valeurs* pour chaque attribut. Le domaine de valeurs d'un attribut est l'ensemble des valeurs que peut prendre un attribut. Cet ensemble peut être décrit en intension ou en extension. Il peut aussi être défini par un type. Ce type peut être contraint par une *contrainte d'intégrité*.

Exemple 2.2.1 : Une personne pourra être représentée par une entité. Cette entité possédera plusieurs attributs : nom, prénom, numéro de sécurité sociale par exemple. Deux entités de type personne peuvent être reliées entre elles par une association représentant la parenté. Enfin, l'attribut « numéro de sécurité sociale » a pour domaine de valeur les entiers contraints par les règles de formation du numéro INSEE.

On peut émettre quelques remarques sur les définitions précédentes :

- un attribut peut concerner une entité : couleur d'un objet, âge d'une personne. Il peut aussi concerner une liaison : nombre de pièces fournies par un fournisseur donné par exemple.
- un même concept du monde réel peut dans certains cas être considéré comme attribut, et dans d'autres cas comme entité. Par exemple, la couleur, en général considérée comme l'attribut d'un objet (n'a pas d'existence propre en l'absence de l'objet), peut parfois être considérée comme entité (dans une base de données traitant de colorimétrie) avec des relations entre couleurs, des mélanges de couleurs etc. La couleur est alors considérée comme ayant une existence propre.

2.3. Structure des types

Nous allons maintenant définir les types (ou classes) d'entités, d'attributs et d'associations.

2.3.1. Type d'entités

Un *type d'entités* permet de caractériser un type particulier d'élément du monde réel pouvant avoir une existence propre. Les entités d'un même type ont la même structure. Ce sont des occurrences d'un même type.

Voici quelques exemples de types d'entités : personne, voiture, ville, table, ...

Dans la suite du cours, nous confondons volontairement entité avec type d'entités, mais il faut bien se rendre compte que la distinction qui s'opère entre entité et type d'entités est la même que celle que l'on fait entre objet et classe par exemple.

2.3.2. Type d'attribut

Un type d'attribut peut être considéré comme une fonction faisant correspondre une valeur à un élément d'un ensemble que constitue le type d'entité.

Voici quelques exemples de types d'attribut : couleur, âge, longueur, largeur, ...

Par exemple, si l'on considère une classe d'entités « personne », le type d'attribut « âge » sera une fonction associant chaque entité de type « personne » à un entier positif représentant son âge.

Attention, il ne faut pas confondre l'entité avec les attributs qui permettent de l'identifier.

2.3.3. Type d'association

Un type d'association est une description d'une association entre entités ayant une signification particulière.

Voici quelques exemples de types d'association : conduit, possède, ...

Exemple 2.3.1 : Le nom d'une personne est un type d'attribut de l'entité personne, même s'il est utilisé pour désigner de manière unique une occurrence de personne. En général, un sous ensemble des types d'attribut d'un type d'entités qui permet de distinguer sans ambiguïté des occurrences d'entités de ce type est considéré comme « clé » d'identification ou de dénotation de cette entité. Il peut y avoir plusieurs clés possibles, ou aucune. Dans ce dernier cas, seules des associations dites « faibles » permettront de distinguer deux occurrences différentes d'entités en s'aidant d'occurrences d'autres entités qui lui sont associées par le lien faible. Nous reviendrons sur ces deux notions dans la suite du chapitre.

2.3.4. Représentation par diagramme

On peut représenter graphiquement des classes d'entités, d'attributs et d'association. La représentation graphique consiste à décrire un diagramme dans lequel les entités sont représentées par des rectangles, les associations par des losanges et les attributs par des ellipses reliés aux entités ou aux associations. Des lignes continues relient les entités aux associations.

La figure 2.1 présente un exemple de diagramme entité-association dans lequel les entités sont « Personne », « Voiture » et « Ville » et les associations « Possède » et « Transport ».

On peut également trouver la « syntaxe » présentée sur la figure 2.2 pour les diagrammes (elle sera parfois employée dans la suite du cours, car plus compacte).

2.4. Notion d'occurrence

Les occurrences représentent les valeurs décrites par les types ou les classes. L'ensemble des occurrences décrit une description en extension associée aux types ou aux classes.

2. Modèle de Chen

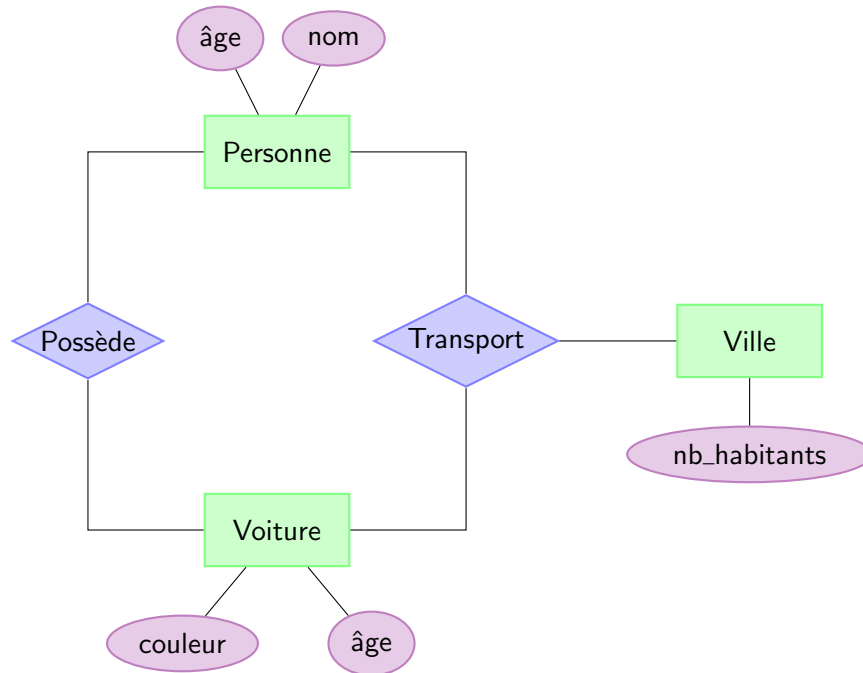


FIGURE 2.1. – Un exemple de diagramme entité-association

- une *occurrence d'entité* est un élément du monde réel ayant une existence propre. En général elle est dénotée par un ensemble d'occurrences (de valeurs) d'attributs.
- une *occurrence d'attribut* (ou une valeur d'attribut) est une valeur, un résultat d'une fonction pour un type particulier d'attribut.
- une *occurrence d'association* est un cas particulier d'information, représentant un lien entre plusieurs occurrences d'entités du monde réel.

Exemple 2.4.1 : Une occurrence de Personne sera dénotée par son nom et son prénom, ou par son numéro de sécurité sociale, mais dans ce cas, l'occurrence de l'entité Personne a une existence propre qui est indépendante de ce que peut être son nom ou son numéro de sécurité sociale.

Une occurrence d'attribut est la valeur « Dupont » de l'attribut nom, de même que la valeur « 30 » est une occurrence de l'attribut âge et la valeur « rouge » une occurrence de l'attribut couleur.

On peut représenter des occurrences d'un diagramme comme sur la figure 2.3.

2.5. Représentation des concepts

Nous avons introduit informellement les concepts généraux nécessaires à la définition d'un schéma. Cette section présente les différents concepts nécessaires à la définition formelle du modèle de Chen.

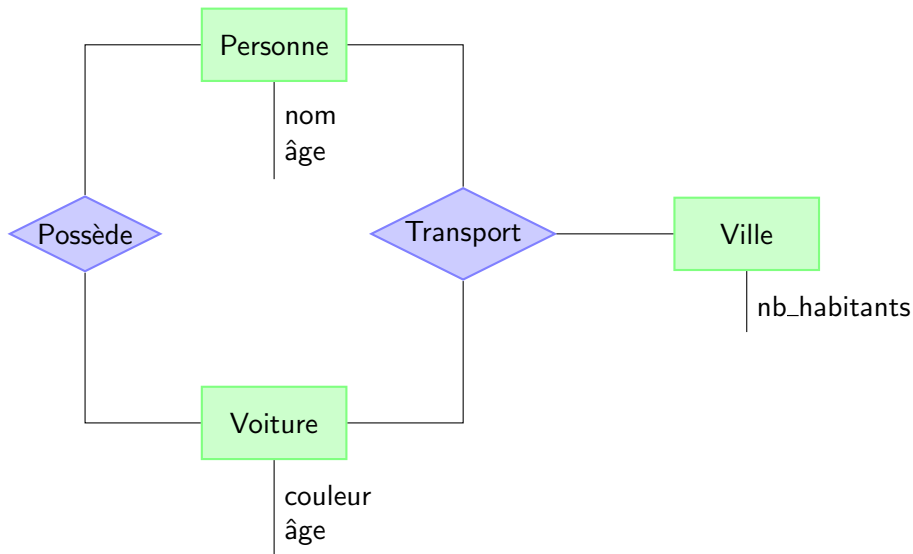


FIGURE 2.2. – Un exemple de diagramme entité-association : nouvelle syntaxe

2.5.1. Notion de clé et de clé primaire

Définition 2.5.1 (clé et clé primaire).

Une clé est un attribut ou un ensemble d'attributs d'une classe d'entités ou d'association dont la connaissance d'une valeur identifie sans ambiguïté une occurrence unique de cette classe. Une clé primaire est une des clés possibles choisie pour servir de moyen d'identification d'une occurrence. □

Quelques remarques peuvent être émises sur cette définition :

- si une classe d'entités ne contient pas un tel attribut, on peut en créer un (par exemple un numéro d'identification, cf. plus loin la section sur les entités faibles) ;
- une association est identifiée par les valeurs des clés primaires des entités qui la composent ;
- les attributs clés d'une classe sont des injections de la classe dans leur domaine de valeurs.

2.5.2. Cardinalité du rôle d'une classe d'entités dans une classe d'associations

Définition 2.5.2 (rôle).

Le rôle d'une classe d'entités E dans une classe d'association A est l'interprétation de la présence d'une entité de classe E dans une association de classe A . □

Par exemple, le rôle joué par « Personne » dans la classe d'association « Possède » avec « Voiture » est « propriétaire ». « Personne » peut jouer d'autres rôles dans d'autres classes d'association.

2. Modèle de Chen

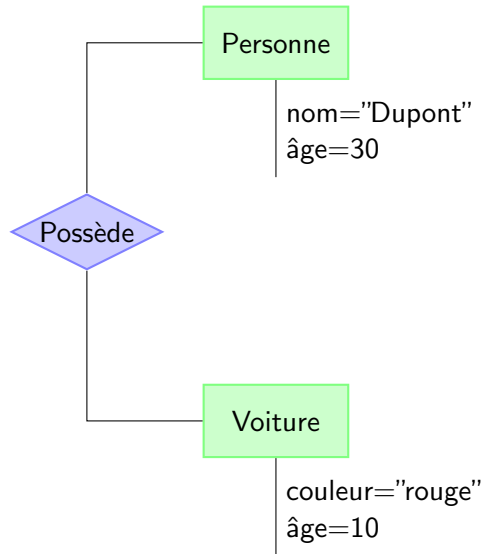


FIGURE 2.3. – Représentation des occurrences d'un diagramme entité-association

Définition 2.5.3 (cardinalité).

La cardinalité est une mesure de l'étendue des interventions possibles d'une entité quelconque d'une classe E dans des associations d'une classe A .

La cardinalité est un intervalle $[min, max]$ (noté également $min..max$) sur \mathbb{N} , où min et max représentent respectivement le nombre minimal et le nombre maximal d'interventions possibles. \square

Il existe plusieurs types de cardinalités : elles peuvent être de type fonctionnel ou hiérarchique.

— *fonctionnel* : ($min = 0$ ou $min = 1$) et $max = 1$.

Dans ce cas, une entité d'une classe d'entité E intervient au plus une fois dans une association A , mais peut ne pas intervenir du tout. Si elle intervient dans une association de classe A , une entité de la classe E intervient une seule fois.

— *hiérarchique* : ($min = 0$ ou $min = 1$) et $max = n$, $n > 1$.

Dans ce cas, une entité de classe E peut intervenir dans plusieurs associations de classe A . Elle peut intervenir plusieurs fois dans la même association. Comme précédemment, la valeur de min nous indique que :

— si $min = 0$, l'entité de classe E peut ne pas intervenir dans une association de classe A ;

— si $min = 1$, l'entité de classe E intervient au moins une fois dans une association de classe A .

Exemple 2.5.1 : Supposons que l'on ait une représentation d'un service de location de voiture comportant une classe d'entités « Voiture » associée à une classe d'association « Location ». Dans ce cas, la cardinalité du rôle « est louée » de « Voiture » dans

« Location » est $[0, n]$: une voiture peut ne pas être louée ou être louée par plusieurs personnes à des moments différents.

Supposons maintenant que l'on ait une classe d'entités « Personne » liée à une classe d'association « Travail » par le rôle « emploie ». On suppose que « Travail » est liée également à « Service » et représente donc le fait qu'une personne travaille dans un service. La cardinalité du rôle « emploie » peut être :

- $[1, 1]$ si un employé n'est attaché qu'à un seul service ;
- $[1, n]$ si un employé est rattaché à plusieurs services. □

2.5.3. Cas particulier des classes d'associations binaires

Ce cas particulier constitue une synthèse des rôles de deux classes d'entités qui interviennent dans une association. Ce cas est courant lors de la modélisation de bases de données. Pour cela, on définit un certain nombre de concept caractérisant les cardinalités des rôles joués par les classes d'entité dans la classe d'association.

Définition 2.5.4.

Soit A une classe d'association entre deux classes d'entités E et F . Cette classe est dite :

- bijective si les rôles joués par E et F sont fonctionnels, i.e. chaque entité de E et de F ne peut intervenir que dans une seule association de A , et si la cardinalité minimale des deux rôles est 1.
On note alors $E^1 \longleftrightarrow^1 F$.
- pseudo-bijective si les cardinalités des rôles joués par E et F sont de type $[0, 1]$.
On note alors $E^0 \longleftrightarrow^1 F$ ou $E^1 \longleftrightarrow^0 F$.
- hiérarchique de E vers F si le rôle joué par E est hiérarchique et le rôle joué par F est fonctionnel.
On note alors $E^1 \longleftrightarrow^n F$.
- fonctionnelle de E vers F si le rôle joué par F est fonctionnel et celui de E hiérarchique.
On note alors $E^n \longleftrightarrow^1 F$.
- maillée si les rôles joués par E et F sont hiérarchiques.
On note alors $E^n \longleftrightarrow^m F$. □

On peut émettre quelques remarques sur ces définitions :

- dans le cas d'une classe d'association bijective, une entité de E est associée dans A à une entité de F et une entité de F est associée dans A à une entité de E ;
- dans le cadre d'une classe d'association hiérarchique de E vers F ou fonctionnelle de F vers E , une entité de E peut être associée dans A à plusieurs entités de F ;
- dans le cadre d'une classe d'association maillée, une entité de E peut être associée dans A à une entité de F .

Dans le monde anglo-saxon, on trouvera les appellations suivantes :

- une classe d'association hiérarchique de E vers F est appelée *many-one* de E vers F ;
- une classe d'association bijective de E vers F est appelée *one-one* de E vers F ;
- une classe d'association maillée de E vers F est appelée *many-many* de E vers F .

2. Modèle de Chen



FIGURE 2.4. – Un schéma entité-association avec rôles et cardinalités

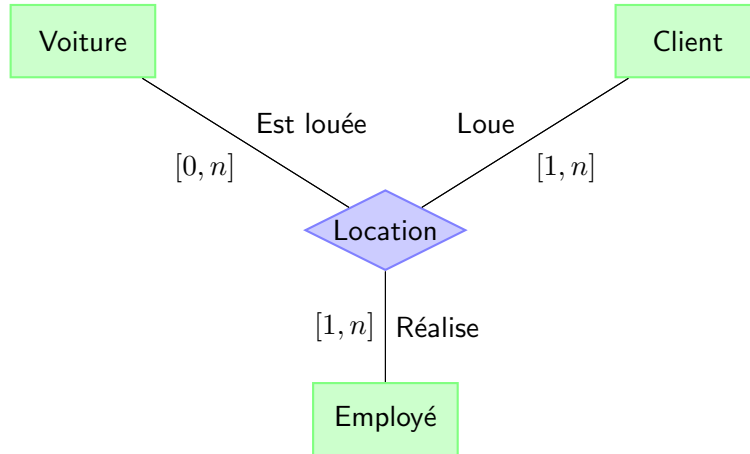


FIGURE 2.5. – Un schéma entité-association avec une association n-aire

2.6. Présentation du schéma conceptuel d'une base de données : le diagramme entité-association

Le diagramme entité-association permet de représenter graphiquement les concepts introduits précédemment. Ce type de représentation aide à la compréhension (donc à la modélisation) des informations dans une base de données. Il permet aussi la représentation des échanges. Cette notion d'échange est importante car elle représente la dynamique des informations qui est une caractéristique difficile à capturer.

Ce diagramme doit donc permettre la représentation graphique des classes d'entités, des classes d'associations, des rôles, des cardinalités et des exemples

Nous étudierons ce type de diagrammes sur des exemples.

La base de données des employés et des services est représentée par le diagramme 2.4.

La base de données des locations de voiture est représentée par le diagramme 2.5.

Les diagrammes entité-association précédents ne sont pas complets. Nous devons y ajouter les clés primaires ainsi que les différents attributs des relations. Par exemple, le diagramme complet représentant la base de données des employés et des services est représenté par la figure 2.6.

Les clés primaires sont représentées sur un diagramme en *soulignant* les attributs concernés.

On pourra remarquer que les cardinalités utilisées dans le modèle de Chen sont utilisées dans le sens « contraire » des cardinalités utilisées dans les associations en UML. Par exemple, la figure 2.7 présente un diagramme entité-association et le diagramme UML

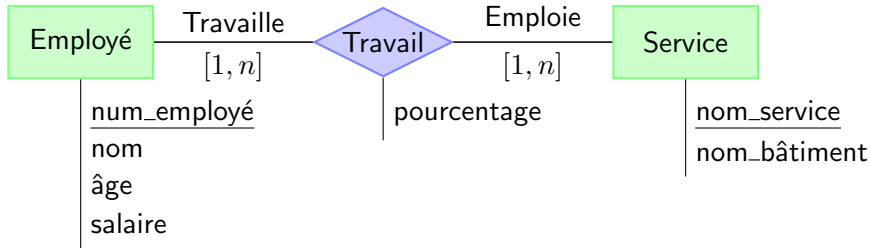


FIGURE 2.6. – Un diagramme entité-association avec clés et attributs

correspondant dans lesquels on suppose qu'une personne ne travaille que deux un seul service. On voit alors que la position des multiplicités est « inversé ».

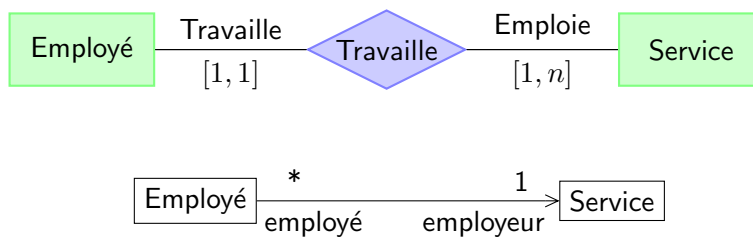


FIGURE 2.7. – Comparaison UML/entité-association pour les cardinalités

Nous pouvons à présent établir la définition d'un schéma conceptuel.

Définition 2.6.1 (schéma conceptuel d'une base de données).

Un schéma conceptuel est composé des éléments suivants :

- un diagramme entité-association ;
- un dictionnaire des attributs ;
- les domaines de valeurs des attributs.

□

Une notation alternative, appelée *crow's foot* (« patte de corbeau », en relation avec la forme d'un des éléments de la notation) et utilisée dans certaines méthodes et outils, propose une autre façon de noter les associations et les multiplicités correspondantes. Les entités sont reliées directement par une ligne avec le nom de l'association. Des symboles permettent de représenter les cardinalités :

- un cercle représente 0
- une barre verticale représente 1
- la « patte de corbeau » représente « plusieurs »

On combine ensuite ces symboles pour donner la multiplicité des associations. Par exemple, dans la figure 2.8 la multiplicité de l'entité Artiste est [1, 1] et celle de Chanson est [0, n].

2. Modèle de Chen



FIGURE 2.8. – Diagramme entité-association utilisant la notation « patte de corbeau »

2.7. Dépendances entre attributs

La dépendance entre deux attributs ou deux groupes d'attributs X et Y rend compte du rapprochement d'attributs au sein d'une même classe d'entités ou d'associations. La dépendance entre deux attributs ou deux groupes d'attributs X et Y est dite :

— *bijective* si chaque valeur de X peut être directement associée à une seule valeur de Y et réciproquement.

On note $X^1 \longleftrightarrow^1 Y$.

— *hiérarchique* si chaque valeur de X peut être directement associée à plusieurs valeurs de Y et si chaque valeur de Y ne peut être associée qu'à une valeur de X .

On note $X^1 \longleftrightarrow^n Y$.

— *fonctionnelle* si la dépendance entre Y et X est hiérarchique.

On note $X^n \longleftrightarrow^1 Y$.

— *maillée* si chaque valeur de X peut être directement associée à plusieurs valeurs de Y et réciproquement.

On note $X^n \longleftrightarrow^m Y$.

On peut remarquer que :

— tous les attributs d'une classe dépendent fonctionnellement de sa clé primaire ;

— il peut ne pas exister de dépendance entre deux attributs : cela ne dépend que de l'univers du discours ;

— la cardinalité du rôle d'une classe d'entités E de clé primaire X dans une classe d'association A est analogue à la nature de dépendance de X vers le groupe d'attributs clé de A .

2.8. Quelques propriétés du modèle entité-association

2.8.1. Bases de données

Le modèle entité-association donne une définition synthétique d'une base de données. Nous obtenons un ensemble d'informations élémentaires connues sur une réalité appelées attributs et regroupables en classes d'entités ou classes d'associations.

2.8.2. Non redondance de l'information

Un attribut ne peut pas être présent dans deux classes d'entités différentes ou dans deux classes d'associations différentes. Un attribut clé d'une classe d'entités est présent dans toute classe d'associations où intervient cette classe d'entités (redondance fictive). Ici, la clé représente la classe d'entités.

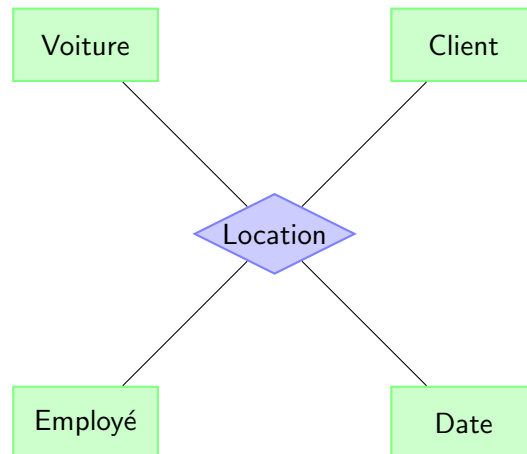


FIGURE 2.9. – Schéma entité-association avec entité « Date »

2.8.3. Extension de la notion de clé d'une classe d'associations

Ce problème est lié à la puissance d'expression du formalisme entité-association et particulièrement à celui de l'association. Nous nous intéressons à la définition de la clé dans une association sur l'exemple de la base de données des locations de voitures. Une location est identifiée par la voiture louée, le client et l'employé qui réalise la location, à condition que le client ne loue pas deux fois la même voiture au même employé. La date de location permet de distinguer ces deux locations. Mais le modèle interdit de faire de la date une partie de la clé de « Location », car seules les classes d'entité peuvent avoir des attributs.

Pour résoudre ce problème, trois solutions sont possibles :

1. créer une classe d'entités « Date » (cf. figure 2.9).

Critique : « Date » n'a d'autre intérêt que de respecter le formalisme entité-association. Ses entités sont toutes les dates du calendrier qui représentent plus naturellement un domaine de valeurs qu'une classe d'entités.

2. transformer « Location » en une classe d'entités (cf. figure 2.10).

Critique : le schéma est plus compliqué. On rajoute une classe d'entités « Location », l'attribut « no_de_location » et une classe d'association regroupant les quatre clés des entités reliées.

3. étendre la clé de la classe d'association « Location » avec la date de location (cf. figure 2.11).

Critique : « Location » n'est plus exactement une classe d'associations. Elle est identifiée par :

- les clés des trois entités « Voiture », « Client » et « Employé » et
- l'attribut propre « date_location »

Malgré l'entorse au formalisme entité-association, nous adopterons cette solution à cause de sa simplicité de compréhension et de représentation.

2. Modèle de Chen

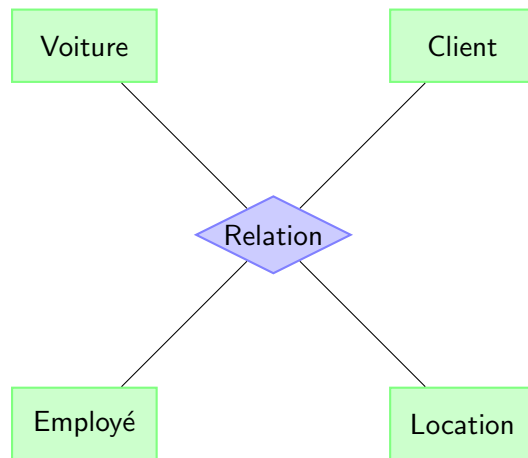


FIGURE 2.10. – Schéma entité-association avec entité « Location »

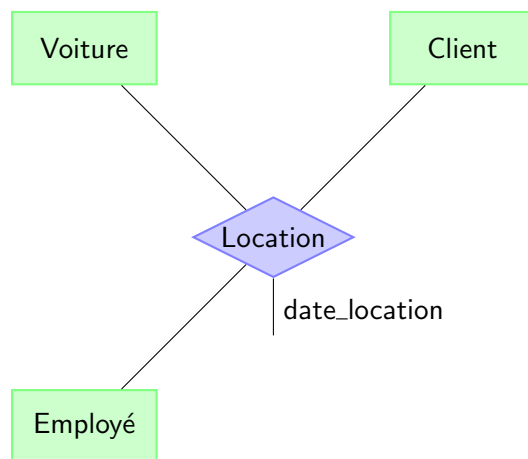


FIGURE 2.11. – Schéma entité-association avec attribut d'association

La possibilité d'étendre la clé d'une seule classe d'associations avec un ou plusieurs attributs est souvent utilisée pour éviter des surcharges d'entités fictives représentant :

- la date;
- l'heure;
- ...
- ou plus généralement, un domaine de valeurs (classe d'entités à un seul attribut).

2.9. Entités faibles

Reprenons le schéma présenté sur la figure 2.6. Celui-ci modélise le fonctionnement d'une entreprise en associant des employés à des services via l'association Travail. Supposons maintenant que l'entreprise souhaite modéliser plus finement l'organisation des services en introduisant l'entité Équipe. Cette entité représente les différentes équipes intervenant dans un service et est caractérisée par un nom. On obtient alors le diagramme représenté sur la figure 2.12.

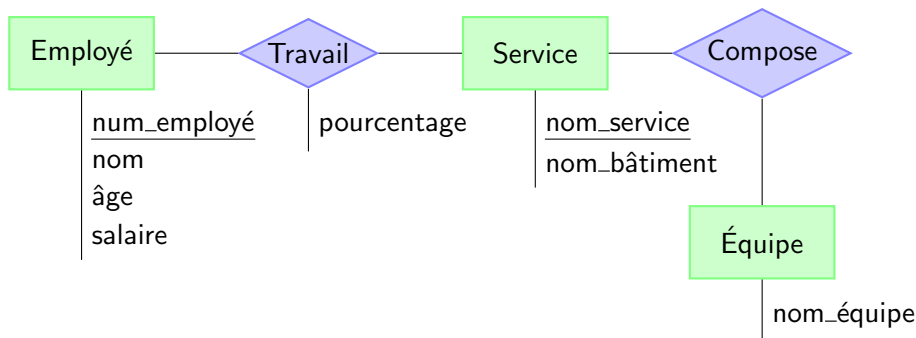


FIGURE 2.12. – Diagramme entité-association avec l'introduction d'Équipe

On se pose maintenant la question de savoir quelle est la clé primaire de l'entité Équipe. Elle ne peut pas être l'attribut nom_équipe de Équipe, car des équipes dans des services différents peuvent avoir le même nom (par exemple, deux services différents ont une équipe « qualité »). Il faut avoir également la connaissance du nom du service de l'équipe pour pouvoir identifier l'équipe. On dit alors que Équipe est une *entité faible*.

Définition 2.9.1 (entité faible).

Une entité E est dite faible si sa clé primaire est composé d'attributs propres à l'entité mais également de d'attributs clés d'autres entités lié à E par des associations fonctionnelles de E vers ces entités. On dit que ces associations sont des associations supports de E . □

Pour représenter cela dans un diagramme entité-association, on encadre l'entité faible et ses associations support par un double trait (cf. figure 2.13).

Remarque : Une solution couramment employée pour pallier le problème des entités faibles est d'introduire un identifiant unique (sous forme d'un entier par exemple) comme

2. Modèle de Chen

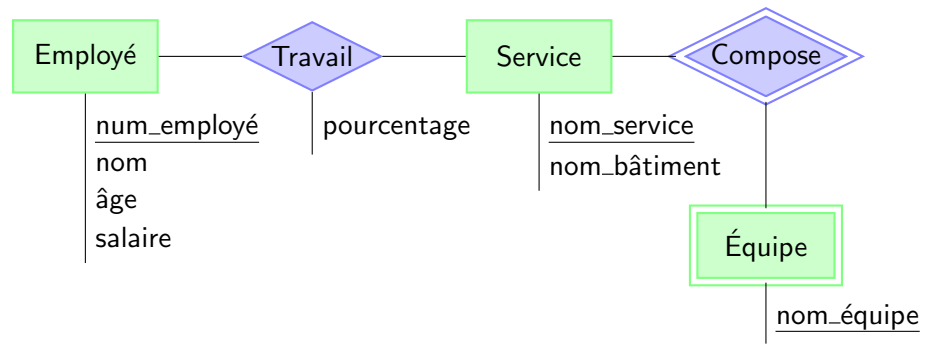


FIGURE 2.13. – Diagramme entité-association avec l'introduction d'Équipe comme entité faible

attribut des entités concernées. On peut par exemple penser au numéro de sécurité sociale dans le cas d'une personne (le nom et le prénom ne permettant pas d'identifier de façon unique une personne). Cette technique permet de simplifier le problème, mais introduit un attribut « artificiel » parfois difficile à manipuler, sa signification n'étant pas très intuitive (il est plus facile de travailler avec un couple Nom/Prénom qu'un numéro de sécurité sociale pour un utilisateur par exemple).

Dans notre cas, l'introduction d'un tel attribut nous amènerait au diagramme présenté sur la figure 2.14.

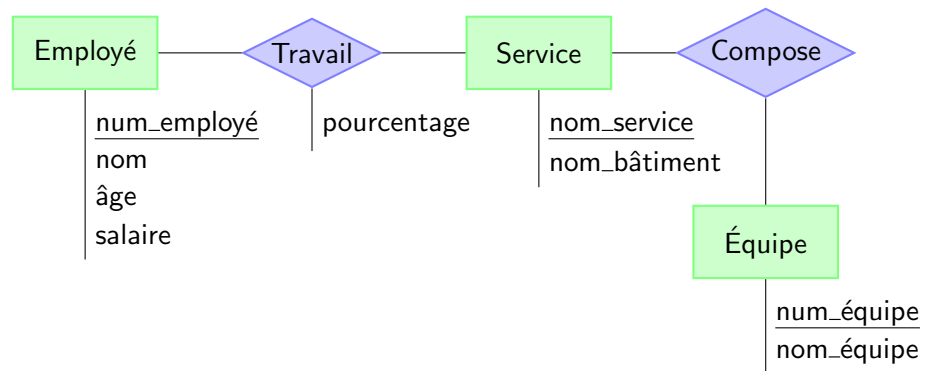


FIGURE 2.14. – Diagramme entité-association avec l'introduction d'un attribut clé

2.10. Exemple : informatisation de la gestion d'un parc informatique

La représentation textuelle suivante présente un modèle entité-association représentant une base de données permettant la gestion d'un (vieux...) parc informatique avec

2.10. Exemple : gestion d'un parc informatique

ordinateurs, types d'ordinateurs, logiciels de base, configuration, relation de compatibilité etc.

classe d'entités Ordinateur

Attributs

référence	domaine	nos de référence	clé
nombre de postes	domaine	entier	
capacité mémoire	domaine	nb de k octets	
nombre disquettes	domaine	entier	
nombre disques	domaine	entier	
prix ordinateur	domaine	somme en francs	

classe d'entités Type d'ordinateur

Attributs

type	domaine	types d'ordinateurs	clé
constructeur	domaine	constructeurs	
pays	domaine	pays	
revendeur	domaine	revendeurs	
nb postes maximal	domaine	entier	
mémoire maximale	domaine	nb de k octets	
unité centrale	domaine	noms d'uc	
capacité disquette	domaine	nb de k octets	
capacité disque	domaine	nb de k octets	

classe d'associations Classe matériel entre classes d'entités

Ordinateur	rôle	appartient
	cardinalité	[1,1]
	représentée par	référence
Type d'ordinateur	rôle	regroupe
	cardinalité	[1,n]
	représentée par	type

attributs (référence, type) : clé

classe d'associations Compatibilité entre classes d'entités

Type d'ordinateur	rôle	est référence
	cardinalité	[0,n]
	représentée par	type
Type d'ordinateur	rôle	est compatible
	cardinalité	[1,n]
	représentée par	type

attributs (type, type) : clé

2. Modèle de Chen

classe d'entités Système

Attributs

nom système	domaine	noms de systèmes	clé
concepteur système	domaine	concepteurs de logiciels	
provenance	domaine	pays	

classe d'associations Logiciel de base

entre classes d'entités

Type d'ordinateur	rôle	supporte
	cardinalité	[1,n]
	représentée par	type
Système	rôle	est écrit
	cardinalité	[1,n]
	représentée par	nom système

attributs (type, nom système) : clé

classe d'entités Logiciel

Attributs

nom logiciel	domaine	noms de logiciels	clé
classe	domaine	types de logiciels	
concepteur logiciel	domaine	concepteurs de logiciels	
origine	domaine	pays	
prix logiciel	domaine	somme en francs	

classe d'associations Configuration

entre classes d'entités

Logiciel	rôle	est conçu
	cardinalité	[1,n]
	représentée par	nom logiciel
Système	rôle	supporte
	cardinalité	[1,n]
	représentée par	type
Type d'ordinateur	rôle	accueille
	cardinalité	[1,n]
	représentée par	type

attributs (nom logiciel, nom système, type) : clé

mémoire centrale min	domaine	nb de k octets
nb disque min	domaine	entier

classe d'associations Installation

entre classes d'entités

Logiciel	rôle	est installé
	cardinalité	[1,n]
	représentée par	nom logiciel

Ordinateur	rôle	est doté
	cardinalité	[1,n]
	représentée par	référence
Systeme	rôle	accepte
	cardinalité	[1,n]
	représentée par	nom système
attributs (nom logiciel,référence,nom système) : clé		

La figure 2.15 propose un diagramme entite-association représentant cet exemple.

2.11. Le langage de modélisation de données EXPRESS

EXPRESS est un langage de modélisation de données permettant la représentation formelle de modèles de données. Historiquement, EXPRESS a été défini puis normalisé par le groupe ISO en charge du développement de la norme STEP (*Standard for Exchange of Product data*) pour l'échange de données techniques entre différents systèmes de CAO. EXPRESS est donc un standard ISO sous le numéro ISO 10303-11.

EXPRESS est un langage de modélisation à objets avec toutes les caractéristiques des approches à objets (entités, liens, liens inverses, héritage,...). Il est fondé sur la description d'un ensemble fini d'entités, décrivant le modèle, regroupées dans une structure de schéma qui joue le rôle de module. Un schéma peut importer des entités définies dans un autre schéma. Cette structure permet de représenter les différentes catégories de connaissances identifiées par Minsky, à savoir la connaissance structurelle, la connaissance descriptive et la connaissance comportementale.

2.11.1. Les entités

Une entité est caractérisée par un nom ainsi qu'un ensemble d'attributs typés.

Syntaxe :

```
ENTITY E ;
  Att_1 : Type_1 ;
  Att_2 : Type_2 ;
  ...
  Att_n : Type_n ;
END_ENTITY ;
```

□

Les types `Type_i` peuvent être des types de base (**integer**, **real**, **boolean**, ou **string**) ou des collections ou agrégats (**list**, **array**, **bag**, **set**). Par exemple, on peut écrire :

```
ENTITY Eleves ;
  Nom : STRING ;
```

2. Modèle de Chen

```
Prenom : STRING ;
Age : INTEGER ;
Cours_suivis : LIST[1 : ?] OF STRING ;
END_ENTITY ;
```

Les agrégats peuvent s'utiliser avec un intervalle (qui, lorsqu'il est absent, est équivalent à [0 : ?]) dont les bornes indiquent le minimum et le maximum des valeurs des index de ces agrégats.

Notons que la déclaration `une_liste : LIST OF INTEGER ;` est équivalente à `une_liste : LIST[0 : ?] OF INTEGER ;`

et ? désigne la valeur indéterminée. On peut aussi écrire une liste dont les index minimum et maximum sont contraints (par exemple `LIST [1 : 6] of ...`). Les agrégats permettent la description de cardinalités dans les associations.

Enfin, toujours pour le typage des attributs, signalons qu'un attribut peut être typé par une autre entité. Cette possibilité est très importante, elle permet la conception de modèles complexes. Par exemple :

```
ENTITY Eleves_bis ;
  Nom, Prenom : STRING ;
  Age : INTEGER ;
  Cours_suivis : LIST[1 : ?] OF cours ;
END_ENTITY ;
```

L'entité `cours` est définie par :

```
ENTITY Cours ;
  Titre : STRING ;
  Volume_horaire : INTEGER ;
END_ENTITY ;
```

2.11.2. Les instances

Les instances d'entités définissent les occurrences. Elles sont définies « comme » en programmation orientée objet. Chaque instance possède un identificateur (adresse ou pointeur) d'un ensemble de valeurs, pouvant elles mêmes être des identificateurs d'autres instances.

Ainsi l'ensemble d'instances suivant :

```
#1=ELEVES_BIS('Dupond', 'Alain', 25, [#2, #3]) ;
#2=COURS('bases de donnees', 20),
#3=COURS('architecture des ordinateurs', 30) ;
#4=ELEVES_BIS('Durand', 'Céline', [#3]) ;
```

décrit deux élèves, Dupond et Durand, qui suivent les cours de bases de données et d'architecture pour Dupond et le cours de bases de données pour Durand.

L'ensemble des instances d'un schéma ou d'un modèle EXPRESS correspond à une description en extension du modèle. La description en intension est donnée par le code EXPRESS source. L'utilisation d'instances est essentielle pour la validation de modèles.

2.11.3. Les contraintes

Il est possible de décrire formellement et rigoureusement des contraintes sur les entités en utilisant des expressions bien formées de la logique du premier ordre à l'aide de la clause **WHERE** du langage EXPRESS. Plusieurs types de contraintes peuvent être définis. Nous en citons trois.

Contraintes locales

Les contraintes locales sont associées à une entité. Par exemple :

```
ENTITY Eleves_bis ;
  Nom, Prenom : STRING ;
  Age : INTEGER ;
  Cours_suivis : LIST[1 : ?] OF cours ;
WHERE
  wr1 : age <= 50 and 18<=age
  wr2 : SIZEOF(cours_suivis) >= 1 ;
END_ENTITY ;
```

La contrainte étiquetée par **wr1** indique que l'âge d'un élève est compris entre 18 et 50 ans alors que **wr2** indique que le nombre de cours suivis par un élève est au moins égal à deux. Elle utilise la fonction prédéfinie **SIZEOF** qui donne le nombre d'éléments d'un agrégat.

Les contraintes locales décrivent des contraintes d'intégrité sur chaque entité.

Contraintes globales

Les contraintes globales portent sur toutes les instances des entités définies dans un schéma. Par exemple, la règle nommée **pas_de_cours_de_bdd** indique que l'ensemble des instances de l'entité **cours** dont le titre vaut '**bases de donnees**' est vide.

```
RULE pas_de_cours_de_bdd FOR (cours)
WHERE
  QUERY(x < * cours | x.titre = 'bases de donnees')=[] ;
END_RULE ;
```

Ici l'expression **QUERY(x < * E | expression_logique)** renvoie un agrégat d'instances de *E* tel que l'expression logique **expression_logique** est vraie. Notons au passage que **expression_logique** peut être décrit comme une fonction logique dans le langage algorithmique (langage impératif proche de Ada et de Pascal) associé au langage EXPRESS. **QUERY** est un itérateur sur les instances. Il permet le codage des quantificateurs existentiels et universels en logique.

2. Modèle de Chen

Les contraintes globales sont utilisées pour la description des contraintes d'intégrité de schémas.

Contraintes d'unicité

Une contrainte d'unicité introduite par le mot clé **UNIQUE** définit la notion de clé pour les instances. C'est une forme de contrainte globale.

Par exemple :

```
ENTITY Eleves_ter ;  
  Nom : UNIQUE STRING ;  
  Prenom : STRING ;  
  Age : INTEGER ;  
  Cours_suivis : LIST[1 : ?] OF cours ;  
WHERE  
  wr1 : age <= 50 and 18<=age  
  wr2 : SIZEOF(cours_suivis) >= 1 ;  
END_ENTITY ;
```

Indique que les noms associés à chaque élève seront uniques dans l'ensemble des instances de **Eleves_ter**.

Les contraintes d'unicité sont utilisées pour la description de clés.

Cette brève description du langage EXPRESS permet de coder les schémas entité-association vus précédemment et de les valider à partir d'un ensemble d'instances. De plus, cette approche utilisant EXPRESS autorise la description formelle des contraintes d'intégrité.

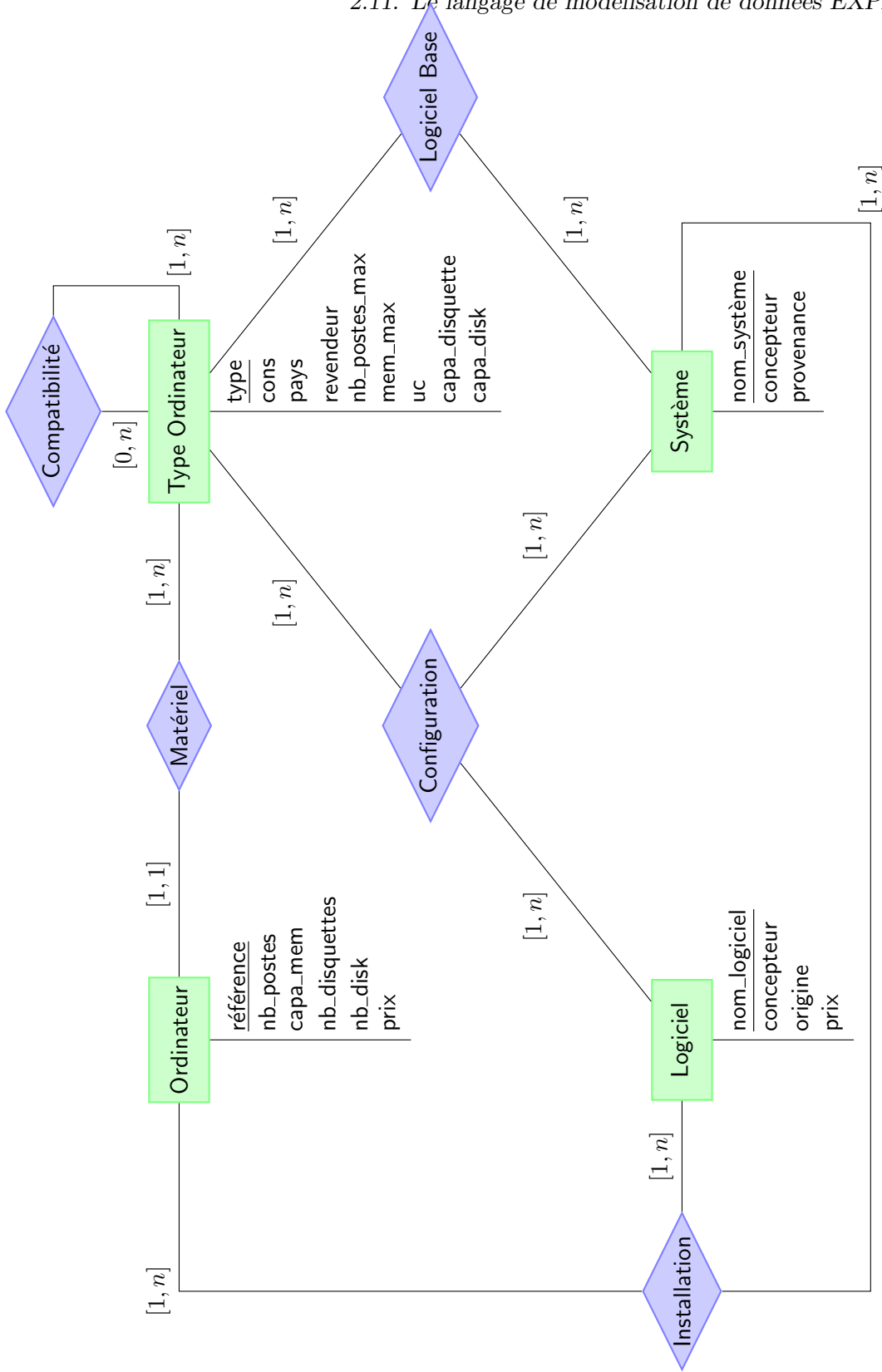


FIGURE 2.15. – Un diagramme entité-association présentant l'exemple de la gestion d'un parc informatique

3. Stockage des informations

Les systèmes que nous considérons doivent permettre de stocker les informations contenues dans les bases de façon pérenne et de retrouver et de manipuler rapidement ces informations suivant différents critères. Dans ce chapitre, nous allons survoler les problèmes posés par le stockage des informations et nous intéresser plus particulièrement aux fichiers de données.

On pourra remarquer que pour l'utilisateur (et même l'administrateur) d'une base de données, la manipulation et la structure des fichiers sont transparentes : ce chapitre permet simplement de mieux appréhender les notions qui peuvent contraindre l'implantation d'une base de données ou l'écriture de requêtes.

Le lecteur souhaitant approfondir le sujet pourra consulter [18, 40, 41, 17].

3.1. Généralités sur le stockage des données

Dans un ordinateur classique, il existe plusieurs composants physiques qui permettent de stocker des informations. Nous allons rapidement détailler ces composants et leurs caractéristiques.

3.1.1. Mémoires à accès aléatoire

Les premiers types de composants que l'on peut rencontrer sont les composants dits à mémoire à accès aléatoire (RAM pour *Random Access Memory* en anglais)¹. Ces composants ont un coût d'accès à une information qui est identique quel que soit l'endroit où est stockée cette information (par opposition aux disques durs ou aux CDROM par exemple qui ont besoin d'une tête de lecture pour pouvoir lire/écrire des données). En particulier, il n'y a aucune corrélation entre le temps d'accès à une donnée et la ou les données qui ont été récupérées précédemment.

La plupart du temps, ces composants sont dits *volatiles*, car ils ont besoin d'être maintenus sous tension pour conserver leurs informations. Nous verrons qu'il existe également des mémoires à accès direct non volatiles, comme les mémoires flash.

Les mémoires DRAM

Les mémoires DRAM (*Dynamic Random Access Memory*) sont des mémoires très simples : l'état de la mémoire (1 bit) est conservé dans un condensateur de faible capacité dont l'accès est gardé par un transistor (cf. figure 3.1a). Le principal problème de ces composants est qu'il faut constamment rafraîchir le condensateur à cause de sa décharge.

1. On fait souvent l'amalgame entre ces composants et la mémoire principale d'un ordinateur.

3. Stockage des informations

Ce sont cependant des composants peu onéreux et le temps d'accès à une information est de l'ordre de quelques dizaines de ns. On les utilise normalement pour la mémoire principale.

Les mémoires SRAM

Les mémoire SRAM (*Static Random Access Memory*) sont des mémoires qui sont plus compliquées d'un point de vue architecturale : il faut en effet 4 ou 6 transistors (cf. figure 3.1c). Le principal avantage de ces composants est qu'il n'y a pas besoin de cycle de rafraîchissement pour conserver l'état et que l'état est accessible quasi instantanément. Par contre, ces composants sont très gourmands en énergie et chers. Ils sont essentiellement utilisés pour les mémoires cache.

Les mémoires flash

Les mémoires flash sont des mémoires EEPROM (*Electrically Erasable Programmable Read-Only Memory*) qui peuvent être programmées et effacées. Elles n'ont pas besoin d'être sous tension pour conserver les informations. Les cellules d'une mémoire flash sont composées de transistors à porte flottante : ils disposent d'une porte dite flottante en plus de la porte de contrôle habituelle. Cette porte permet de conserver une information, car elle est isolée électriquement (cf. figure 3.1e). On peut modifier la valeur d'une cellule en lui appliquant une tension positive ou négative. Le principal défaut de ces composants est le nombre de cycles écriture/effacement limité (entre 100000 et 1000000 actuellement).

3.1.2. Mémoires à accès séquentiel

Le terme « mémoire à accès séquentiel » que nous utilisons ici n'est pas une appellation consensuelle. Il nous permet cependant de distinguer les mémoires à accès aléatoire, où l'accès à un bit d'information a un coût constant, d'un type de composants de stockage d'information où l'accès à un bit d'information dépend de l'endroit où est stockée l'information et de l'information qui a été accédée précédemment. Cette non uniformité d'accès à l'information est due à la présence d'éléments physiques (tête de lecture par exemple). Par contre, ces systèmes de stockage ont des capacités plus importantes que les mémoires à accès aléatoires. Parmi ces composants, on trouve principalement les disques durs magnétiques, les disques optiques (CD, DVD) et les bandes magnétiques (dont nous ne parlerons pas ici du fait de leur obsolescence).

Les disques durs magnétiques

Les disques durs magnétiques sont des moyens de stockage très utilisés actuellement : ils ont en effet de bonnes capacités de stockage, un temps d'accès raisonnable à l'information et un coût peu onéreux.

Un disque dur magnétique est composé de deux éléments principaux :

- un assemblage de disques magnétiques ou plateaux en rotation autour d'un axe ;

3.1. Généralités sur le stockage des données

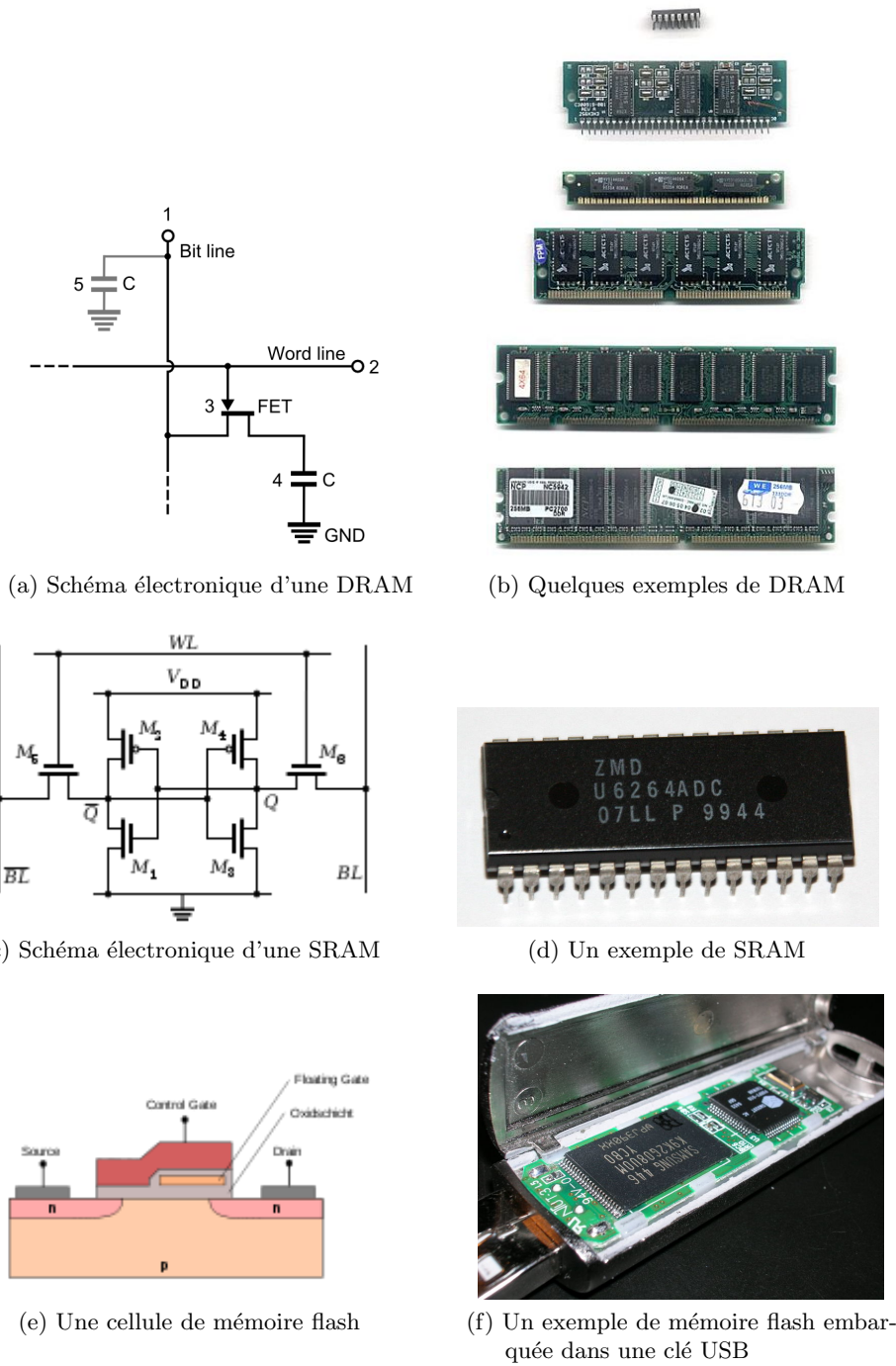


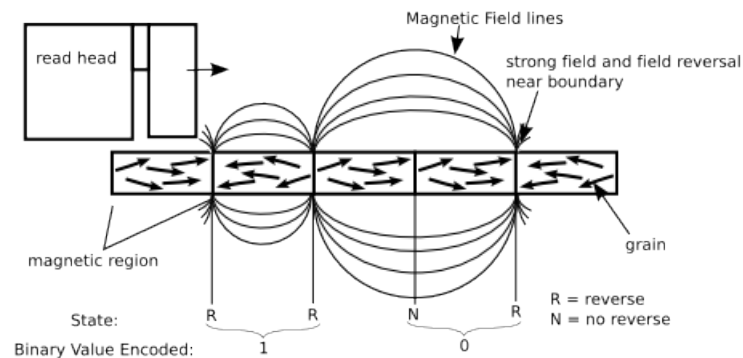
FIGURE 3.1. – Différents types de mémoires à accès direct

3. Stockage des informations

— un assemblage de têtes de lecture magnétiques sur un bras mobile pouvant « lire » et « écrire » sur la surface des disques.

Le principe de stockage de l'information est simple : la tête de lecture/écriture est capable de magnétiser ou de détecter les champs magnétiques sur la surface du disque qui comporte une couche externe magnétique.

La figure 3.2a présente une coupe de la surface magnétique d'un disque dur et les champs magnétiques en jeu et la figure 3.2b présente un disque dur ouvert où l'on peut voir le premier plateau du disque et une tête de lecture.



(a) Coupe de la section magnétique d'un disque dur



(b) Disque dur ouvert

FIGURE 3.2. – Principe de fonctionnement d'un disque dur magnétique et « intérieur » d'un disque dur

Chaque disque magnétique est séparé en pistes (*tracks*) qui sont des cercles concentriques. Chaque piste est elle-même découpée en secteurs, de taille variable suivant la position de la piste, mais contenant la même quantité d'information (512 octets normalement pour un disque magnétique). Le secteur représente l'unité *atomique* du disque : on ne peut lire ou écrire qu'un secteur, et si une partie d'un secteur est corrompue, le

3.2. Hiérarchie de composants de stockage dans un ordinateur

secteur entier l'est aussi (cf. figure 3.3).

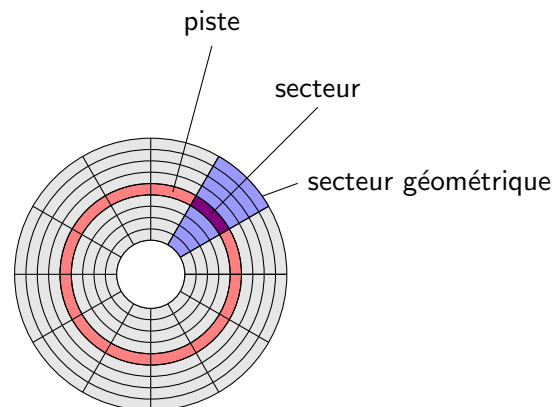


FIGURE 3.3. – Vue d'un plateau avec une piste, un secteur géométrique et un secteur

Disques optiques

Les disques optiques offrent une capacité de stockage plus importante que les disques durs magnétiques (en ne considérant bien sûr qu'un seul plateau du disque dur magnétique). La lecture d'un disque optique se fait en utilisant un laser de faible puissance qui est réfléchi par des creux et des bosses disposés sur la surface du disque. On peut maintenant réécrire des informations sur un même support optique.

On trouve dans cette catégorie de périphériques de stockage les CD (*Compact Disc*), les DVD (*Digital Versatile Disc*), les *mini-discs*, les disques Blu-ray etc.

3.2. Hiérarchie de composants de stockage dans un ordinateur

Les composants de stockage utilisés dans un ordinateur sont organisés suivant une hiérarchie reflétant les capacités de stockage de ces composants, mais également le temps d'accès aux données contenues. On peut résumer rapidement cette hiérarchie sur le diagramme 3.4. Nous allons les détailler rapidement dans ce qui suit.

3.2.1. Mémoires volatiles : le cache et la mémoire principale

Les deux premiers niveaux de stockage dans un ordinateur sont des composants de stockage *volatiles* : dès que l'ordinateur est mis hors-tension, les informations contenues dans ces composants disparaissent.

Le *cache* est un composant de stockage qui se situe habituellement sur la même puce que le microprocesseur. Les informations s'y trouvant sont en fait des copies de la mémoire principale pour un accès plus rapide. Les caches actuels ont une capacité de 1 Mo² et

2. On rappelle qu'en informatique, l'unité de base est l'octet (*byte* en anglais) qui correspond à un nombre binaire à 8 chiffres.

3. Stockage des informations

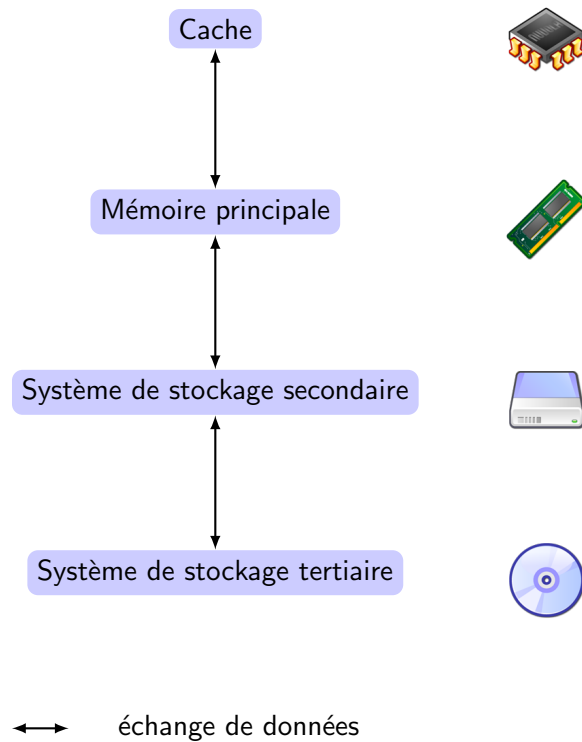


FIGURE 3.4. – Hiérarchie de composants de stockage de données

les données peuvent être lues ou écrites entre le microprocesseur et le cache en 1 ns. Par contre, pour faire transiter une donnée entre le cache et la mémoire principale, il faut plus de temps (environ 10-100 ns). Les caches sont souvent réalisés avec des SRAM.

La *mémoire principale* (que l'on appelle souvent abusivement RAM) est l'endroit où l'on stocke toutes les informations et les instructions des programmes en cours. Actuellement, les ordinateurs ont une mémoire principale d'environ 1 Go (on trouve bien évidemment des machines avec des capacités plus importantes). Le temps d'accès à la mémoire principale est toujours le même : il varie entre 10 et 100 ns. On accède à un octet de la mémoire principale par une *adresse*. La mémoire principale est souvent réalisée avec de la DRAM.

3.2.2. Système de stockage secondaire

Le stockage secondaire est un système de stockage de plus grande capacité que les deux précédents. C'est un système *non volatile*. Il est constitué la plupart du temps de disques magnétiques. Le système de stockage secondaire « contient » deux éléments : la mémoire virtuelle et un ou plusieurs systèmes de fichiers.

Média de stockage pour la mémoire secondaire

La plupart du temps, le système de stockage secondaire s'appuie sur un disque magnétique, encore appelé disque dur³. Ce système de stockage est bien évidemment permanent : les données enregistrées sur un disque dur seront conservées après la mise hors tension du système.

Pour travailler avec les informations contenues dans le disque, il faut les transférer en mémoire principale. Ce transfert se fait par *blocs*, dont la taille dépend du système d'exploitation ou du système gérant les données considéré. La taille typique d'un bloc est de 4 Ko.

La capacité actuelle des disques dur est de plusieurs centaines de Go. Il faut environ 15 ms pour écrire ou lire un bloc sur le disque dur.

Nous considérerons dans la suite que le système de stockage secondaire est constitué d'un ou plusieurs disques durs.

La mémoire virtuelle

Supposons que l'on veuille faire tourner plusieurs applications sur un même ordinateur, par exemple un système d'exploitation, un atelier de développement logiciel, un navigateur internet, un lecteur de courrier électronique. Il se peut très bien que la mémoire nécessaire pour ces applications (code à exécuter, pile, données à manipuler) soit supérieure à la mémoire principale disponible sur l'ordinateur. Dans ce cas, on ne peut pas utiliser en même temps toutes ces applications.

Les ordinateurs disposent depuis une vingtaine d'années d'un mécanisme de mémoire virtuelle qui permet de résoudre (en partie) ce problème.

Lorsque l'on exécute un programme sur un ordinateur, ce programme manipule un certain nombre de données : variables, résultats de lecture de fichiers etc. Ces données occupent un espace d'adressage *virtuel*, qui s'étend pour la plupart des machines sur 32 bits, soit environ 4 milliards d'adresses. On peut donc représenter la mémoire virtuelle par un espace de 4 Go.

Comme la mémoire virtuelle est beaucoup plus importante que la mémoire principale disponible sur la plupart des machines, une partie de son contenu est stocké sur le système de stockage secondaire. C'est ce que l'on appelle le *swap* sur les machines Unix par exemple. Les informations sont échangées entre le système de stockage secondaire et la mémoire principale par blocs entiers, que l'on appelle des *pages*.

Les fichiers et les systèmes de fichiers

En informatique, un fichier est une collection de données enregistrée sur un support physique. Ces informations restent inchangées et disponibles pour un usage ultérieur. On peut considérer deux catégories de fichiers. La première concerne les fichiers de données et la seconde les fichiers de programmes (source, objet, programmes du système

3. On peut également utiliser une disquette, un système de stockage sur mémoire Flash etc.

3. Stockage des informations

d'exploitation ou d'application). Dans notre cas, on s'intéressera principalement aux fichiers de données.

On peut distinguer deux types de fichiers :

- les fichiers *physiques* qui sont les données enregistrées sur un support physique. Ces fichiers sont indépendants de tout traitement et ont une existence matérielle. Ils peuvent être transportés, lus ou faire l'objet de traitements, archivés et détruits.
- les fichiers *logiques* qui sont des collections de données, éventuellement identiques à celles contenues dans des fichiers physiques, utilisées à un moment donné par un programme. Ils n'ont d'existence que pendant la durée de leur utilisation, que l'on peut représenter par l'instant entre deux primitives : OPEN, qui permet d'ouvrir un fichier, et CLOSE, qui permet de le fermer.

Les fichiers sont organisés par un *système de fichiers*, qui permet par exemple d'associer un nom à un fichier physique particulier. Parmi les grands types de systèmes de fichiers, on peut citer ext3, NTFS, NFS, FAT32, HFS etc. Le système de fichiers fournit un certain nombre de primitives permettant de travailler avec les fichiers :

- création d'un fichier
- mise à jour d'un fichier
- destruction d'un fichier
- lecture d'un fichier

Certains systèmes de fichiers peuvent proposer directement d'autres primitives (recherche dans un fichier par exemple).

Le lecteur souhaitant approfondir le sujet pourra consulter [40, 41].

Gérer les erreurs sur un disque dur

Les données que nous allons utiliser seront la plupart du temps stockées sur un disque dur. Malheureusement, les disques durs sont des supports physiques avec une durée de vie courte par rapport à la pérennité que l'on veut assurer pour les données (cf. par exemple l'étude présentée dans [34]). Les erreurs sur un disque dur peuvent aller d'une erreur de lecture/écriture temporaire à une corruption complète du disque qui le rend illisible. Il faut donc trouver des moyens de protéger les informations contenues dans un disque et de savoir si elles ont été corrompues. Pour cela, on dispose de plusieurs moyens :

- utilisation de *checksums* qui permettent de détecter l'éventuelle corruption des informations en stockant une indication condensée sur l'information. Par exemple, on peut compter le nombre de 1 apparaissant dans une collection de bits et stocker comme information la parité de ce nombre.
- duplication des informations sur plusieurs disques. On peut dupliquer chaque disque ou utiliser des approches plus complexes, comme par exemple RAID. On se référera à [18, 10] pour plus de détails.

3.2.3. Système de stockage tertiaire

Il se peut enfin que le volume de données à stocker dépasse la capacité du ou des disques durs que l'on a à disposition. On peut également vouloir faire une copie de sauvegarde

des données sur un support plus sûr. Pour cela, on peut utiliser plusieurs médias de stockage :

- des bandes magnétiques, peu pratiques car encombrantes et lentes. On peut les regrouper en silos de stockage.
- des disques optiques (CD ou DVD) montés en « *juke-box* ».

3.3. Impact du stockage secondaire

Nous pouvons résumer les temps d'accès et capacité de stockage des différents systèmes sur la figure 3.5 (les unités des axes représentent des puissances de 10).

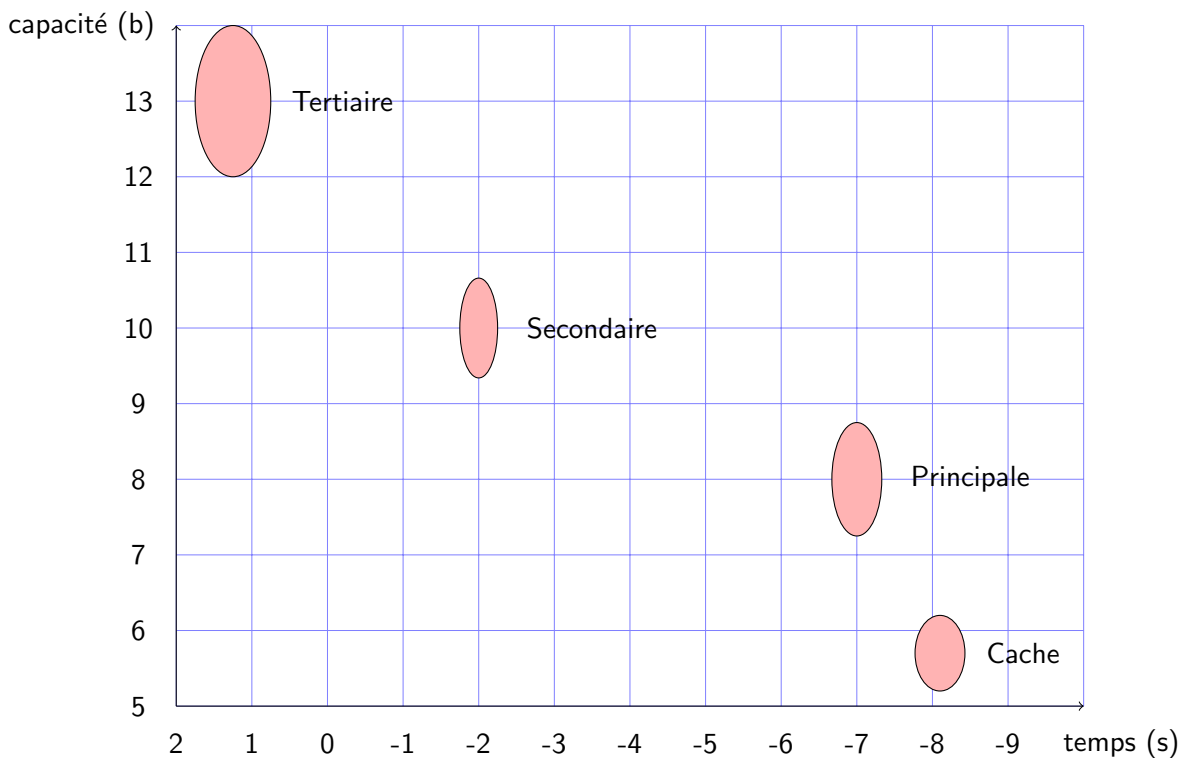


FIGURE 3.5. – Temps d'accès et capacité de stockage des différents systèmes

Le problème qui se pose est le suivant : les données que l'on manipule ne peuvent pas être contenues la plupart du temps dans la mémoire principale (qui est assez rapide, cf. 3.2.1). Pour traiter ces données, il va donc falloir transférer fréquemment des blocs du disque dur vers la mémoire principale. Or un transfert de bloc est coûteux : si on prend une moyenne de 15 ms comme temps de transfert, une machine aurait le temps d'exécuter des millions d'instructions pendant le transfert !

Le fait de stocker les données à manipuler sur un disque dur va donc amener à faire un certain nombre d'optimisations, soit à bas niveau (organisation des données sur les différents disques par exemple), soit parfois à haut niveau. En effet, lorsque l'on conçoit un

3. Stockage des informations

algorithme et que l'on calcule sa complexité en temps, on a souvent tendance à considérer que les données manipulées sont accessibles à moindre coût en mémoire principale. Mais dans un système de taille conséquente, les données ne « tiennent » pas en mémoire principale. Certains algorithmes très efficaces en mémoire principale ne le seront plus si on doit utiliser la mémoire secondaire.

Prenons par exemple un tri de données. On suppose que l'on dispose d'une collection de données indexées par une clé sur laquelle on peut construire un ordre et on cherche à trier ces données grâce à cet ordre. Normalement, les algorithmes les plus efficaces pour cela sont les algorithmes de type « *Quicksort* ». Ces algorithmes ne seront pas du tout efficaces pour des données n'étant pas contenues en mémoire principale (car on ferait alors beaucoup de transferts entre le disque dur et la mémoire principale). On utilise alors des variantes de l'algorithme de tri/fusion 3.1, dont le principe est de trier des sous-ensembles des données avant de fusionner les sous-listes ainsi triées.

Par exemple, le tableau 3.1 présente l'utilisation de cet algorithme sur deux listes déjà triées.

La complexité de cet algorithme est bien connue : $O(n \log(n))$. Il est donc assez efficace. Dans le cas de tri de données stockées sur un système de stockage secondaire, on utilise un algorithme de tri/fusion en deux phases dans lequel on trie d'abord des parties des données qui peuvent être contenues totalement dans la mémoire principale avant de les fusionner. On pourra consulter [43] pour plus de détails.

On peut également optimiser l'accès aux données contenues dans le disque dur en utilisant une technique « miroir » consistant à dupliquer les disques pour « paralléliser » les entrées/sorties ou utiliser des techniques de *buffering* prédictif pour précharger des données que l'on sait nécessaires pour la suite de l'application. Ces techniques ne seront pas abordées ici, mais on pourra consulter par exemple [21] pour approfondir le sujet.

3.4. Représentation des données

Nous allons dans cette section nous intéresser brièvement à la représentation des données sur le disque dur. Dans un premier temps, nous regarderons comment sont représentés et stockés les éléments de base, comme les attributs et les enregistrements. Nous aborderons ensuite la notion d'index et de collection d'enregistrements.

3.4.1. Stockage des attributs

Nous avons vu dans le chapitre 2 que les informations sont représentées par des entités et des associations. En particulier, les entités ont un certain nombre d'attributs, qui représentent les caractéristiques de la classe d'entités décrite⁴. Chaque type d'attribut a normalement un type informatique qui lui correspond. Ainsi, un entier est représenté la plupart du temps par 4 octets. En particulier, nous verrons dans le chapitre 8 que le langage SQL propose des types pour représenter les domaines des attributs. On parlera de *champ* pour parler de l'implantation d'un attribut.

4. Nous reviendrons sur ces notions dans le chapitre 6 lorsque nous présenterons le modèle relationnel

Algorithme 3.3.1 : Algorithme de tri/fusion d'une liste L de n éléments

entrées : une liste L d'éléments non triés de longueur n

sortie : une liste de longueur n contenant les éléments de L triés

```

1 si  $n = 1$  alors
2   | retourner  $l$  ;
3 sinon
4   | pour  $i \leftarrow 1$  to  $\frac{n}{2}$  ou  $\frac{n+1}{2}$  faire
5     |  $S[i] \leftarrow L[i]$  ;
6   fin
7   | pour  $i \leftarrow \frac{n}{2}$  ou  $i \leftarrow \frac{n+1}{2}$  to  $n$  faire
8     |  $S'[i] \leftarrow L[i]$  ;
9   fin
10  Trier( $S$ ) ;
11  Trier( $S'$ ) ;
12  tant que  $S \neq \emptyset$  et  $S' \neq \emptyset$  faire
13    |  $e1 \leftarrow$  premier élément de  $S$  ;
14    |  $e2 \leftarrow$  premier élément de  $S'$  ;
15    | si  $e1 \leq e2$  alors
16      |  $output = output \cup e1$  ;
17      | retirer  $e1$  de  $S$  ;
18    | sinon
19      |  $output = output \cup e2$  ;
20      | retirer  $e2$  de  $S'$  ;
21    fin
22  fin
23  si  $S = \emptyset$  alors
24    |  $output = output + S'$  ;
25  sinon
26    |  $output = output + S$  ;
27  fin
28  retourner  $output$  ;
29 fin

```

3. Stockage des informations

étape	L_1	L_2	sortie
0	{2, 4, 8, 10}	{1, 5, 12, 13}	{}
1	{2, 4, 8, 10}	{5, 12, 13}	{1}
2	{4, 8, 10}	{5, 12, 13}	{1, 2}
3	{8, 10}	{5, 12, 13}	{1, 2, 4}
4	{8, 10}	{12, 13}	{1, 2, 4, 5}
5	{10}	{12, 13}	{1, 2, 4, 5, 8}
6	{}	{12, 13}	{1, 2, 4, 5, 8, 10}
7	{}	{}	{1, 2, 4, 5, 8, 10, 12, 13}

TABLE 3.1. – Un exemple d'utilisation de l'algorithme tri/fusion

3.4.2. Stockage des enregistrements

Un enregistrement est un groupement logique d'informations élémentaires décrit et connu par le programme qui manipule le fichier logique. Par exemple, on peut considérer que les instances d'entités du chapitre 2 constituent des enregistrements. Dans ce cas, un enregistrement va donc être constitué d'un ensemble de champs représentant des attributs.

Un enregistrement peut être de longueur fixe ou variable. Par exemple, si l'on considère des enregistrements représentant chacun un numéro de sécurité sociale, tous les enregistrements auront la même taille. Par contre, si on considère des enregistrements contenant des noms de personnes, on pourra choisir de les stocker avec une taille variable.

Les enregistrements sont stockés dans des blocs. Il se peut qu'un enregistrement soit plus petit qu'un bloc. Dans ce cas, on peut choisir d'ajouter d'autres enregistrements dans le bloc pour gagner de la place. Il se peut également qu'un enregistrement soit plus gros qu'un bloc. Dans ce cas, il faut être capable de le découper pour pouvoir le stocker efficacement sur plusieurs blocs.

Enfin, on ajoute en début d'enregistrement un certain nombre d'informations dans une entête (*header* en anglais), comme par exemple la longueur de l'enregistrement ou une date de dernière modification.

3.4.3. Accès aux données

On peut maintenant se demander comment accéder aux enregistrements (ou plus « physiquement » aux blocs) dont on a besoin. Si le bloc est contenu en mémoire principale, on utilise le mécanisme de mémoire virtuelle pour obtenir l'adresse réelle de ce bloc, puis on peut alors trouver l'adresse du premier octet de l'enregistrement voulu à l'intérieur du bloc. Comment accéder à un bloc qui ne se trouve pas en mémoire principale ?

Il existe des supports de stockage dits *séquentiels*. C'est le cas par exemple des bandes magnétiques dont nous avons parlé pour les systèmes de stockage tertiaires. Dans ce cas, les blocs sont enregistrés les uns à la suite des autres. Pour pouvoir accéder au bloc numéro n , on est obligé de parcourir les $n - 1$ blocs précédents.

Les disques durs (qui sont les moyens principaux de stockage secondaire) sont des supports *adressables*. On peut ainsi déterminer la position d'un bloc ou d'un enregistrement sur le disque dur grâce à une adresse. Il existe deux types d'adresses :

- une adresse *physique* du bloc qui est composée des éléments suivants :
 - la machine sur laquelle se trouve le disque ;
 - un identifiant du disque ;
 - le numéro du cylindre du disque ;
 - le numéro de la piste dans le cylindre ;
 - le numéro du bloc dans la piste ;
 - éventuellement, un décalage (*offset*) ou une *clé* permettant de trouver le début de l'enregistrement dans le bloc.
- une adresse *logique* qui est une chaîne d'octets.

Les adresses logiques sont reliées aux adresses physiques via une table de correspondance stockée sur le disque. L'utilisation d'adresses logiques permet d'avoir un système souple : on peut par exemple déplacer des adresses logiques sans réellement déplacer des blocs, ce qui serait coûteux. De plus, l'accès à une adresse logique peut se faire rapidement au niveau du processeur.

Enfin, les enregistrements comportent de plus en plus de références vers d'autres enregistrements. Cette référence, parfois appelée *pointeur*, est une information contenant l'adresse d'un autre enregistrement. Dans ce cas, la traduction des adresses physiques en adresses logiques est plus compliquée. Par exemple, lorsque l'on charge en mémoire principale un bloc contenant un enregistrement, on a son adresse logique. Si ce bloc contient lui-même un pointeur vers un autre enregistrement, quelle va être la nature de cette adresse en mémoire principale, physique ou logique ? Pour résoudre cela, on utilise des mécanismes dit de *pointer swizzling*.

3.4.4. Ajout, destruction et mise à jour d'enregistrements

Les modifications d'enregistrements (ajout, destruction, mise à jour) sont des opérations qui vont être simples si l'on considère que les enregistrements ne sont ordonnés sur le disque. Cependant, en pratique, on utilisera au moins une clé pour classer les enregistrements. On trouvera dans les enregistrements sur le disque par ordre de clé croissante.

Ceci peut évidemment poser problème. On peut être quasiment certain que l'insertion d'un nouvel enregistrement par exemple va nous obliger à déplacer un certain nombre d'autres enregistrements contenus dans le bloc visé. Il se peut que le bloc puisse accueillir le nouvel enregistrement, mais également que l'on soit obligé d'utiliser un nouveau bloc (bloc de débordement) pour déplacer les enregistrements du bloc visé de plus grande clé. Dans ce cas, il faudra bien sûr régler le problème des adresses pouvant pointer sur ces blocs déplacés.

Lorsque l'on efface un enregistrement, il faut être capable de gérer l'espace ainsi libéré, soit en rendant tous les enregistrements du bloc contigus, soit en maintenant une table contenant toutes les adresses disponibles. Là encore, il faudra gérer les adresses extérieures sur les enregistrements effacés.

Enfin, la mise à jour se ramène à un effacement et une création.

3. Stockage des informations

Évidemment, tous ses problèmes sont rendus plus compliqués dès lors que l'on travaille avec des enregistrements qui n'ont pas de longueur fixe.

3.4.5. Collection d'enregistrements et index

Supposons maintenant que nous ayons une collection d'enregistrements stockée sur le disque dur et que nous souhaitons dans un premier temps retrouver tous les enregistrements de cette collection. Sans plus d'informations, nous serions obligés de parcourir *tous* les blocs du disque pour pouvoir retrouver les enregistrements voulus. Pour pallier ce problème, on peut imposer qu'un certain nombre de cylindres du disque soient réservés pour cette collection et ainsi ne pas être obligés de parcourir l'ensemble du disque.

Cette organisation simple n'est pas très souple : que se passe-t-il si la collection devient plus importante ? Ou si des cylindres réservés ne sont pas utilisés ? De plus, comment trouver facilement tous les enregistrements de la collection dont une valeur d'attribut est 5 par exemple ? Pour cela, nous devons faire appel à la notion d'*index*.

Un index est une structure de données qui prend en paramètre une propriété d'enregistrement et renvoie *rapidement* les enregistrements d'une collection ayant la propriété. Par exemple, si l'on stocke un dictionnaire sous forme d'enregistrement, on pourra avoir un index qui partitionne le dictionnaire avec la première lettre du mot. Si l'on cherche l'enregistrement correspondant au mot « base de données » dans le dictionnaire, en utilisant l'index on ne va parcourir que les mots commençant par la lettre « b ».

En informatique, il existe plusieurs façons de créer des index et de les utiliser : index simples (cas des enregistrements déjà classés), index secondaires, b-arbres, tables de hachage etc.

Prenons un exemple simple. Supposons que nous ayons des enregistrements stockés dans un fichier séquentiel, i.e. un fichier dans lequel les enregistrements sont classés en utilisant une clé. Nous supposons ici que cette clé est un entier. Un exemple de fichier de ce type est représenté sur la figure 3.6.

10	
20	
20	
40	
50	
60	

FIGURE 3.6. – Un exemple de fichier séquentiel

On suppose ici que les valeurs autres que la clé contenues dans l'enregistrement (représentées sur la figure en blanc) sont telles que leur taille excède largement un bloc. Lorsque l'on veut chercher par exemple l'enregistrement correspondant à la valeur 60 dans un tel fichier, on est obligé sans index de le parcourir en entier pour trouver le bon enregistrement, ce qui est coûteux en accès disque.

On peut alors construire un index comme présenté sur la figure 3.7.

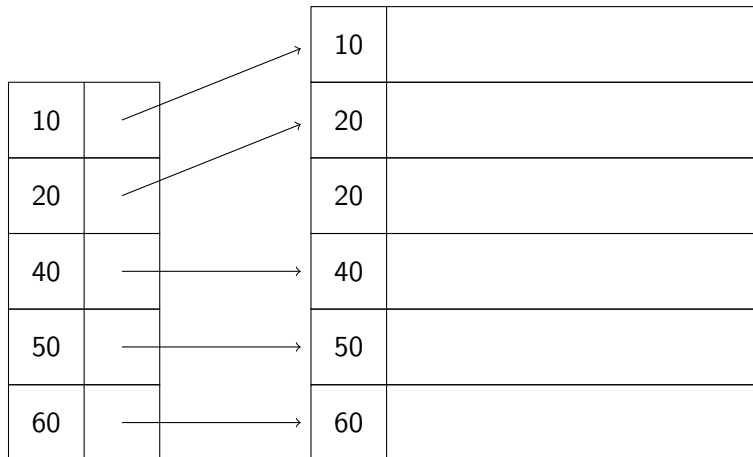


FIGURE 3.7. – Un exemple de fichier séquentiel avec index

Cette fois-ci, l'index nous permet de trouver facilement l'enregistrement que nous cherchons. De plus, la table d'index est de petite taille et tient (normalement) en mémoire principale. Dans notre recherche, on effectuera qu'un seul accès disque pour récupérer l'enregistrement au lieu d'en effectuer six si on n'avait pas d'index.

Nous reviendrons sur la notion d'index dans le chapitre 8.

3.5. Conclusion

Nous venons de présenter brièvement dans ce chapitre les moyens de stockage de données sur un ordinateur. Nous avons exposé les différences de temps d'accès et de taille de stockage entre mémoire principale, système de stockage secondaire et système de stockage tertiaire. Enfin, nous nous sommes rapidement intéressés au stockage d'enregistrement dans un fichier situé sur un disque dur.

Évidemment, il n'était pas question dans ce cours chapitre de regarder en détail tous les problèmes liés au stockage des données (optimisation de l'utilisation du disque dur, accès efficace aux données, index particuliers, index multidimensionnels etc.). Le lecteur intéressé pourra lire les chapitres 11 à 14 de [18] comme introduction.

4. Apports des bases de données. Système de Gestion de Bases de Données (SGBD)

Les bases de données ont été définies dans le but d'éliminer les problèmes qui se posent dans le cas d'une gestion de volumes d'informations lorsque des fichiers de données sont utilisés.

Les fichiers représentent le seul moyen de mémorisation des informations sur un ordinateur. Ils constituent une structure de représentation pour les données comme nous l'avons vu dans le chapitre 3. Par contre, ils ne fournissent pas une structure de représentation pour les informations dans le sens où la sémantique des informations mémorisées dépend du programme qui accède à ce fichier.

Les fichiers présentent des inconvénients liés au fait que chaque utilisateur ou chaque groupe d'utilisateurs qui met en œuvre une application particulière utilise ses propres programmes et parfois ses propres fichiers.

Lorsque plusieurs applications manipulent les mêmes informations ou bien des informations similaires, on obtient :

- une redondance des informations pouvant conduire à des incohérences entre différentes représentations d'une même information ;
- une dépendance logique concernant les programmes d'application. En effet, ces programmes ont besoin de tenir compte de la structure des fichiers, y compris ceux qui ne les concernent pas (par exemple pour des raisons de compatibilité dans le cas de représentations d'informations déjà stockées dans d'autres fichiers) ;
- une dépendance physique concernant les programmes d'application : ces programmes doivent être modifiés chaque fois que la structure physique du fichier est modifiée.

On peut remarquer que la simple juxtaposition de fichiers ne constitue pas un système de gestion de bases de données comme nous le verrons plus loin dans ce chapitre. Les problèmes relevés ci-dessus restent entiers.

4.1. Bases de données

D'un point de vue logique, les bases de données permettent de représenter et d'exploiter les informations logiques qu'elles représentent indépendamment de toute application déjà développée.

D'un point de vue physique, les bases de données sont des systèmes de fichiers élaborés, associés à des programmes d'accès spécialisés qui peuvent faciliter les commandes des échanges entre disques et unités centrales en mettant à la disposition des programmeurs

4. Apports des bases de données et SGBD

des primitives plus puissantes que les simples ordres de lecture et d'écriture (qui existent pour les fichiers).

4.2. Système de Gestion de Bases de Données

Les Systèmes de Gestion de Bases de Données (SGBD) ont été introduits à la suite de la définition du concept de bases de données dans les années 1970. La mission principale des SGBD est d'assurer la séparation la plus grande possible entre les données d'une part et les programmes d'autre part afin de gagner en souplesse.

Pour atteindre les objectifs précédents, le CODASYL (COncference on DATA SYstems Languages) a défini des critères que les SGBD doivent satisfaire. Ces critères, également appelés spécifications du groupe CODASYL sont les suivants :

- le SGBD doit permettre l'accès à tout ou à une partie des fichiers suivant des critères donnés,
- le SGBD doit permettre des accès partagés aux données,
- le SGBD doit éviter (et éventuellement supprimer) les redondances,
- le SGBD doit réaliser une indépendance la plus grande possible entre les programmes qui exploitent la base de données et les fichiers qui représentent les données,
- le SGBD doit permettre une structuration optimale des données par rapport aux traitements qui seront effectués,
- le SGBD doit enfin permettre une réglementation de l'accès aux données.

4.2.1. Structure à trois niveaux

Les différents apports des bases de données par rapport aux systèmes de gestion de fichiers ont pu être réalisés en définissant un système à trois niveaux distincts devant être géré entièrement par le SGBD (cf. figure 4.1).

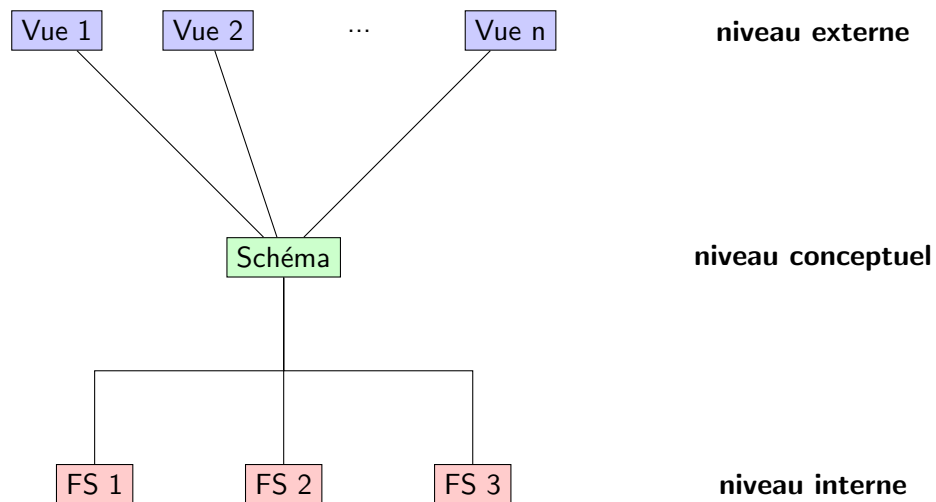


FIGURE 4.1. – Les trois niveaux de représentation d'une base de données

Les trois niveaux ainsi définis sont détaillés dans ce qui suit :

- le niveau externe concerne l'application. Il est constitué des schémas externes, encore appelés vues ou sous-schémas. Pour chaque application, ces vues décrivent le sous-ensemble des informations qu'elles exploitent. Ainsi, chaque utilisateur ou groupe d'utilisateurs se trouve affranchi de la prise en compte des structures qui ne le concernent pas.

Ce niveau permet d'assurer l'indépendance *logique*.

- le niveau conceptuel concerne l'ensemble des applications. Il permet de regrouper toutes les informations utilisées par l'ensemble des utilisateurs de la base de données. Il assure la centralisation des données et donc permet de traiter les éventuels problèmes d'incohérence.

Ce niveau permet de représenter la totalité de la base de données et d'en assurer la cohérence.

- le niveau interne concerne la représentation physique des données. Il permet de gérer les structures physiques de stockage, *a priori* de manière totalement transparente pour l'utilisateur sauf du point de vue des temps de réponse.

Ce niveau permet d'assurer l'indépendance physique des données.

4.2.2. Langages associés aux bases de données

La structure à trois niveaux précédente sert à la représentation et à la gestion des informations. Pour cela, elle nécessite la définition d'un langage. Ce langage est divisé en plusieurs sous groupes relatifs à la création et à l'exploitation :

- le *langage de définition de données*, encore appelé LDD ou DDL (*Data Definition Language*), permet de décrire des structures, conformément au modèle choisi, y compris les sous-schémas ou les vues. Il permet également de définir des règles ou contraintes de cohérence que les informations de la base de données doivent vérifier.
- le *langage de manipulation de données*, encore appelé LMD ou DML (*Data Manipulation Language*), permet la manipulation des informations, c'est-à-dire l'ajout, la suppression, la modification (considérée dans certains cas comme une suppression suivie d'un ajout) et surtout l'interrogation. Pour cette dernière opération, le langage de manipulation (ici d'interrogation) doit être complet, c'est-à-dire permettre de poser n'importe quelle question. Cet aspect est très important dans le cadre de la mise en œuvre des bases de données.
- le *langage de description des structures physiques* permet de définir le niveau interne de la base de données, c'est-à-dire les structures physiques de stockage utilisées pour ranger les informations physiques sur les supports physiques. Ce langage est principalement utilisé par l'administrateur de la base de données. Il sert indirectement à améliorer les performances de la base de données et notamment en termes de temps de réponse lors de l'interrogation.
- le *langage d'expression de sélection* sert essentiellement à l'interrogation et constitue de ce fait un sous-ensemble du LMD. Il a été particulièrement étudié car il représente la partie du langage d'exploitation d'une base de données dans lequel les requêtes sont exprimées. Il peut prendre deux formes :

4. Apports des bases de données et SGBD

- *procédurale* pour indiquer comment sélectionner des informations dans la base de données,
- *assertionnelle* pour indiquer quelles sont les informations à sélectionner.

4.3. Caractéristiques des SGBD

Nous présentons dans cette section les grandes caractéristiques des SGBD.

4.3.1. Objets, relations et schémas

Trois grandes notions sont toujours présentes dans les bases de données : les objets manipulés, les relations et le schéma de la base.

1. *les objets de la base de données* sont les contenants ultimes, décrits par l'ensemble de leurs composants. Par exemple, l'objet véhicule se décompose en marque, type, immatriculation, etc. De même, l'objet employé se décompose en nom, prénom, date de naissance, et cette dernière en jour, mois et année.
2. *les relations entre les objets* se décrivent en général séparément. Elles sont représentées de façons différentes suivant les SGBD et peuvent être de plusieurs types.
3. *le schéma d'une base de données* est la représentation de sa structure. Cette représentation concerne les objets de la base, leurs composants, les relations qui les unissent, les méthodes d'accès à ces objets, les contraintes d'intégrité etc.

4.3.2. Les fonctions de base

Les bases de données peuvent également comporter des objets de service, gérés automatiquement par le SGBD : dates, statistiques d'accès aux objets, etc. Elles peuvent avoir aussi des systèmes de sécurité (autorisations d'accès, chiffrement, . . .) ou bien des primitives pour l'accès à distance par exemple.

4.3.3. Indépendance des données et des programmes

Réaliser l'indépendance des données et des programmes revient à accéder aux données sans pour autant fournir leurs adresses physiques. Si une donnée change d'adresse physique sur le support, ou même si elle change de support, le programme continuera à fonctionner normalement sans qu'il soit nécessaire de le modifier.

Pour réaliser cette indépendance, les descriptions logicielles de tous les fichiers sont rassemblées dans un programme unique, appelé schéma. Tous les accès aux fichiers transiteront par ce schéma. Les programmes ne connaîtront que les noms symboliques des variables et ne connaîtront plus, à la limite, la notion de fichier. Dès lors, le schéma organisera comme il l'entend la disposition des données sur les supports, sans que les programmes s'en préoccupent.

4.3.4. Non redondance

L'existence d'un schéma permet d'éviter la duplication des données. Pour cela, et afin de réaliser une base de données, il est nécessaire de disposer d'un dictionnaire des données qui est la traduction externe du schéma. Ce dictionnaire comporte la liste de toutes les variables de la base de données, avec leur description physique (binaire, chaîne de caractères de telle longueur, ...). Chaque variable conservera le même nom dans chaque programme qui l'utilise.

4.3.5. Partageabilité

De multiples accès (qui peuvent être simultanés) peuvent concerner une même donnée, et donc converger sur le même bloc physique. Le SGBD doit gérer cette simultanéité, essentiellement pour la prévention d'interblocages.

4.3.6. Efficacité des échanges

Il s'agit de l'amélioration des performances des SGBD en termes de temps de réponse indépendamment de la puissance du processeur. Cette performance est obtenue par le choix des structures de données adéquates et d'algorithmes rapides.

4.3.7. Cohérence

Il s'agit d'assurer la cohérence des données. Par exemple, à la suite d'une panne, il faut que les données retrouvent un état cohérent à la reprise du système. La gestion de tels problèmes est également dévolue au SGBD.

4.3.8. Sécurité et fiabilité

Les SGBD doivent prévoir des protections contre les accès non souhaités par la mise en place de mots de passe ou par cryptographie. La fiabilité est directement liée à la nature du logiciel qui représente le SGBD.

4.4. Les composants d'un SGBD

Cette section est un résumé de ce que l'on peut trouver dans [18].

Nous présentons sur la figure 4.2 les différents composants que l'on trouve dans un SGBD. Nous détaillons ensuite rapidement chacun de ces composants et en particulier nous nous attachons à montrer quelle(s) caractéristique(s) présentée(s) en section 4.3 ils implantent.

On s'aperçoit sur la figure qu'il existe deux types d'utilisateurs émettant des commandes sur le SGBD :

- l'administrateur du SGBD ;
- les utilisateurs « normaux » du SGBD.

4. Apports des bases de données et SGBD

L'administrateur du SGBD peut utiliser le langage de description de données pour modifier le schéma de la base de données. Un *compilateur LDD* traduit les commandes de l'administrateur écrites en DDL et celles-ci sont passées au *moteur d'exécution*. Celui-ci demande ensuite au *gestionnaire d'index et de fichiers* de modifier le schéma. C'est ce dernier composant qui contient les informations sur le schéma de la base puisque c'est lui qui gère les index, les enregistrements etc. L'indépendance des données et des programmes est donc assurée par ce composant.

Lors d'une requête par un utilisateur, cette requête est transformée par le *compilateur de requête* en un plan (une séquence d'actions) que le SGBD va appliquer pour répondre à la requête. Le *moteur d'exécution* va ensuite préparer des requêtes pour des enregistrements, des index etc. qui sont des informations de petite taille. Le *gestionnaire de buffers* se charge ensuite de demander les pages correspondantes et le *gestionnaire de stockage* lit et écrit les pages sur le disque dur (ou autre système de stockage secondaire). L'efficacité des échanges est donc assurée par ces derniers éléments.

Enfin, le *gestionnaire de transaction*, le module de *logging et de reprise* et le module de *contrôle de la concurrence* permettent de gérer les aspects transactionnels que nous aborderons dans le chapitre 12. Ce sont ces éléments qui sont chargés la cohérence et la partageabilité des données.

4.5. Conclusion et remarques

Nous venons de présenter rapidement les avantages des bases de données par rapport à un système de stockage d'informations fondé sur l'utilisation de simples fichiers. Nous nous sommes intéressés ensuite aux systèmes de gestion de bases de données, en particulier à leurs caractéristiques et à leur architecture. On peut émettre un certain nombre de remarques sur ce qui a été écrit :

- le système de gestion de bases de données est un système *ouvert* : on peut mettre à jour aussi bien les données que les relations qu'il y a entre elles.
- une base de données sera toujours enregistrée sur des supports d'accès aléatoire. Les programmes de gestion de ces ensembles de données, qui font largement appels à des techniques de recherche sur tables d'index et de clés, utilisent souvent des systèmes de fichiers chaînés qui exigent l'accès direct.
- une base de données est envisagée lorsque la masse et la complexité des informations à traiter deviennent importantes i.e. lorsque les problèmes inhérents aux fichiers classiques apparaissent (rigidité, encombrements des supports).
- la mise en place d'une base de données est moins immédiate que la création d'un fichier supplémentaire dans une architecture logicielle classique. Il faut en effet analyser les relations qui existent entre toutes les données contenues dans les différents fichiers.
- un SGBD associe deux types de concepts différents et complémentaires :
 1. un modèle de structuration des données qui doit être le mieux adapté au système d'informations (notion *statique*),
 2. une méthode d'accès, traduite en programmes d'échange, efficaces et complets,

4.5. Conclusion et remarques

entre les supports périphériques de la base de données et les programmes d'utilisation (notion *dynamique*).

Nous allons maintenant nous intéresser à différents modèles de bases de données, avant de détailler plus précisément le modèle relationnel dans le chapitre 6.

4. Apports des bases de données et SGBD

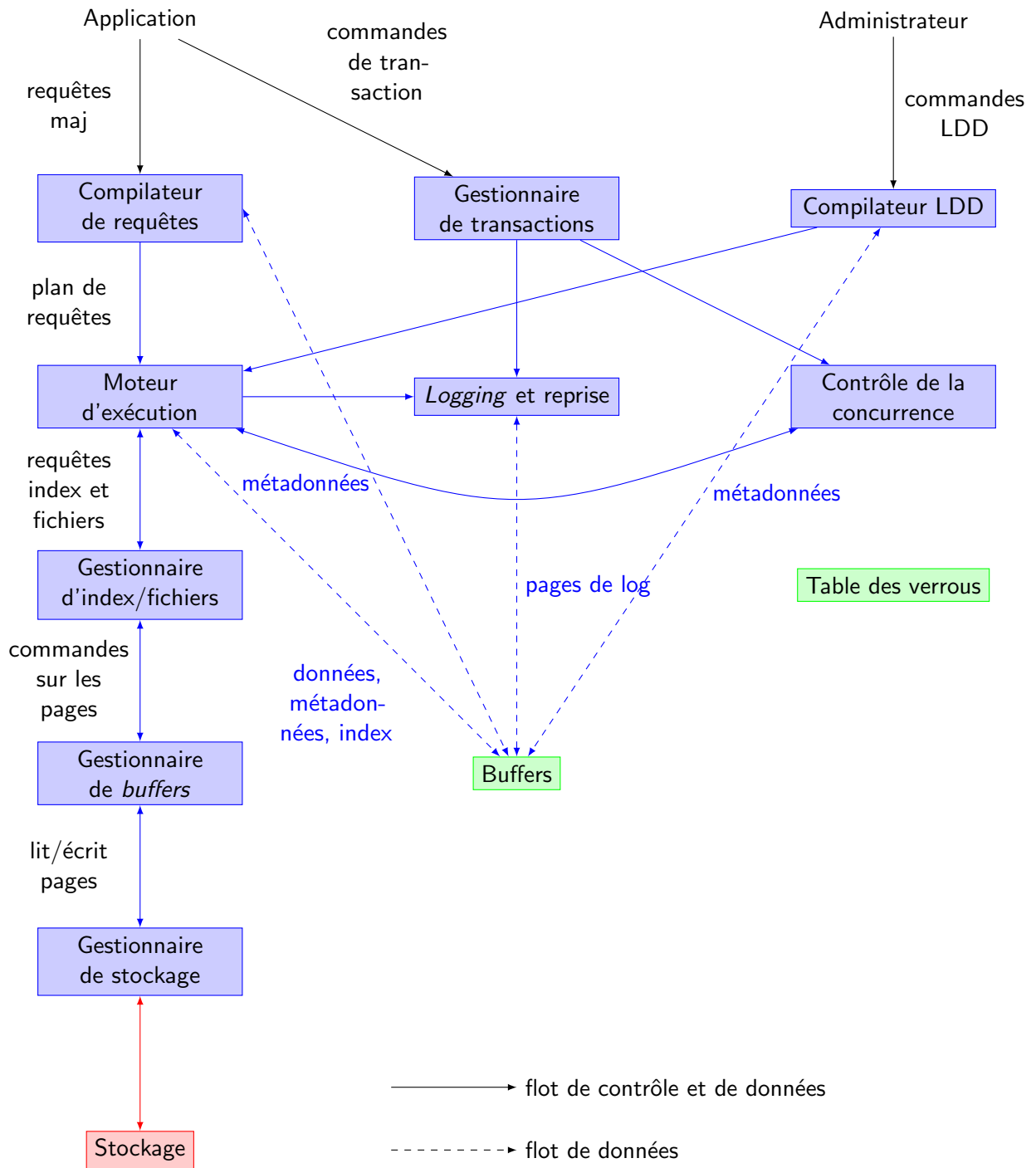


FIGURE 4.2. – Les différents composants d'un SGBD

5. Quelques modèles classiques de bases de données

L'organisation des données est représentée par un modèle de données, outil utilisé pour représenter l'organisation logique de ces données. Ce modèle permet la représentation des objets et des relations. Les modèles de données utilisés dans les SGBD se distinguent par la façon de représenter les relations entre les données.

Les principaux modèles de données utilisés pour les bases de données sont les modèles hiérarchique, réseau, relationnel et à objets. Ces modèles diffèrent dans le traitement des associations. Nous présentons ici rapidement les modèles hiérarchiques et réseau, les modèles relationnels et à objets étant présentés plus en détail dans les chapitres 6 et 13.

5.1. Le modèle hiérarchique

Pour construire les premières bases de données, les informaticiens ont utilisé la structure la plus simple pour la manipulation de données en gestion : la structure hiérarchique inspirée de l'approche utilisée par le langage COBOL pour la description des enregistrements.

5.1.1. Contexte et définitions

Le modèle hiérarchique correspond à la représentation la plus courante et la plus naturelle de la structuration des variables d'un système d'information en gestion. Il est une généralisation des structures de données en COBOL (variables de niveaux 01, 02, 03, ...).

Le modèle hiérarchique repose sur l'établissement d'un arbre de définition hiérarchique dans lequel les relations entre les éléments d'un niveau et du niveau immédiatement inférieur sont de type « un à plusieurs », noté 1..N.

Définition 5.1.1 (modèle hiérarchique).

Le modèle hiérarchique est un modèle de données qui organise logiquement les données selon des relations structurelles. Ces relations (qui sont des graphes) sont des arbres de définitions hiérarchiques (un arbre est un graphe particulier).

Chaque arbre comprend une racine et des branches qui permettent l'accès aux différents niveaux de données. Le niveau d'une donnée mesure sa distance à la racine.

La racine est située au niveau supérieur. Les autres données, qualifiées de dépendantes, se situent au niveau des nœuds de l'arbre (on parle de parents et d'enfants). On obtient alors des chemins d'accès hiérarchique. □

5. Modèles classiques de BD

Exemple 5.1.1 : La figure 5.1 présente une structuration hiérarchique d'informations concernant la représentation d'une adresse. Dans cet exemple, « Adresse », la racine de l'arbre, est le parent de « Numéro », « Rue », « Ville » et « Code Postal ». « Code Postal » a lui-même deux enfants, « Numéro département » et « Numéro bureau distributeur ».

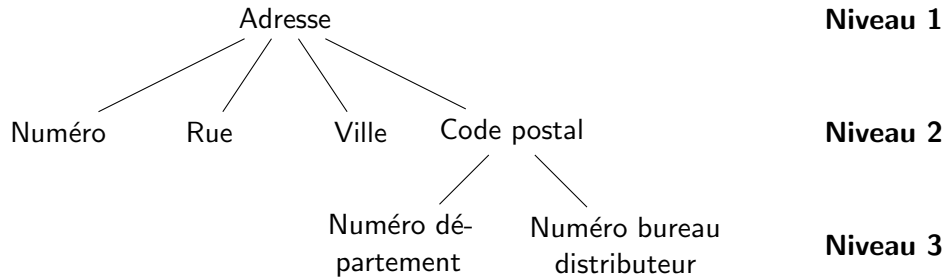


FIGURE 5.1. – Un exemple de structuration hiérarchique

Définition 5.1.2 (attribut).

Un attribut (ou item) est la plus petite unité de donnée manipulable. Elle peut prendre toutes les valeurs compatibles avec son identité.

Les attributs sont caractérisés par un nom unique et un ensemble de valeurs ou domaine de valeurs (type). □

Un attribut très important dans la base est celui qui joue le rôle de clé. Nous reparlerons de cette notion lorsque nous aborderons le modèle relationnel dans le chapitre 6.

Définition 5.1.3 (groupe simple).

Un groupe simple est formé d'un ensemble d'attributs représentés sous la forme d'une arborescence. □

Exemple 5.1.2 : La figure 5.2 présente un exemple de groupe simple.

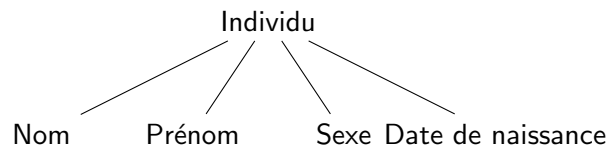


FIGURE 5.2. – Un exemple de groupe simple

Définition 5.1.4 (groupe composé).

Un groupe composé est un groupe dont au moins un de ses attributs est lui-même un groupe. □

Exemple 5.1.3 : La figure 5.3 présente un exemple de groupe composé.

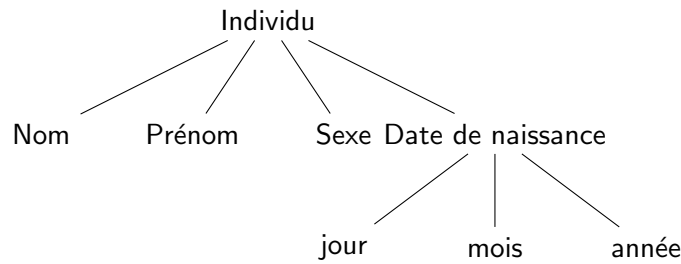


FIGURE 5.3. – Un exemple de groupe composé

Dans une base de données utilisant le modèle hiérarchique, seuls les attributs peuvent avoir des valeurs.

Définition 5.1.5 (groupe répétitif).

Un groupe répétitif est un groupe dont au moins un des groupes constitutifs peut se répéter plusieurs fois. Ce dernier est placé entre parenthèses. □

Exemple 5.1.4 : La figure 5.4 présente un exemple de groupe répétitif. En effet, un représentant peut avoir plusieurs affaires (sinon il n'est pas très bon...).

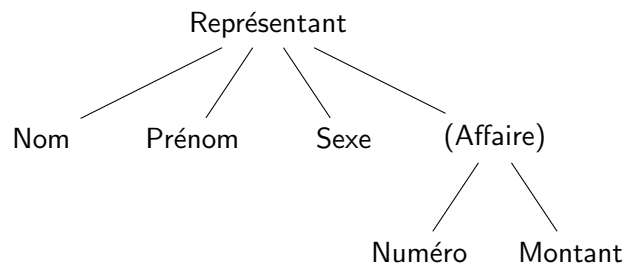


FIGURE 5.4. – Un exemple de groupe répétitif

Il faut noter que la racine d'un groupe composé doit être toujours considérée comme un groupe répétitif car la base de données représente un ensemble de données pour lesquelles on a un arbre. Les parenthèses sont donc omises au niveau de la racine.

5.1.2. Techniques de transformation entre le niveau logique et le niveau physique

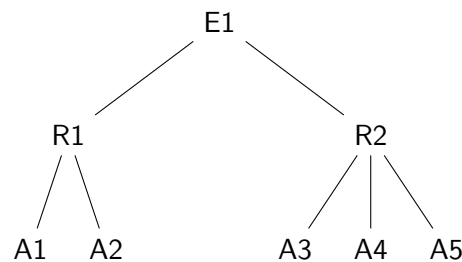
La mise en œuvre d'un SGBD hiérarchique implique l'emploi de pointeurs logiques qui peuvent être combinés avec des éléments physiques afin d'élaborer une base de données hiérarchique.

Dans ce type de SGBD, l'utilisateur dispose de deux organisations physiques de base : l'organisation séquentielle hiérarchisée et l'organisation directe hiérarchisée. Chacune se présente sous deux formes.

5. Modèles classiques de BD

L'organisation séquentielle hiérarchisée implique que les enregistrements sont reliés sur le plan physique de façon adjacente. Ses deux sous-organisations sont désignées par HSAM (*Hierarchical Sequential Access Method*) et HISAM (*Hierarchical Indexed Sequential Access Method*).

- avec HSAM, les enregistrements sont regroupés dans des blocs physiques selon une séquence hiérarchique préétablie. Cette structure respecte l'ordre séquentiel de parcours préfixé d'une structure hiérarchique (de haut en bas et de gauche à droite). Considérons la structure hiérarchique représentée sur la figure 5.5. On voit alors que sa représentation physique stocke les enregistrements en utilisant l'ordre donné ci-dessus : E1 puis R1 puis A1 puis A2 puis R2 etc.



E1	R1	A1	none
A2	R2	none	none
A3	A4	A5	none

FIGURE 5.5. – Une structure hiérarchique et sa représentation HSAM

- avec HISAM, on accède de façon indexée à des arbres de bases de données, chaque arbre étant stocké dans des emplacements physiquement contigus en séquence hiérarchique. Le type élément racine doit contenir un élément clé pour indexer chaque arbre de la base.

Dans le cas de l'organisation directe hiérarchisée, les enregistrements sont reliés par des pointeurs de deux types :

- pointeurs-fils (reliant une racine à ses enfants) ;
- pointeurs-frères (reliant des enregistrements ayant le même père).

L'enregistrement parent est relié au premier de ses enfants. Les deux sous-organisations utilisées pour cette organisation directe sont appelées HDAM (*Hierarchical Direct Access Method*) et HIDAM (*Hierarchical Indexed Direct Access Method*) :

- avec HDAM, on a accès aux enregistrements racines par un algorithme d'adressage sur les éléments-clés de ces enregistrements (par *hash-coding* et zone de débordement).
- avec HIDAM, on dispose d'une zone index (répertoire) qui contient les valeurs clés des enregistrements racines et des pointeurs liant chaque valeur clé à l'enregistrement correspondant.

5.1.3. Caractéristiques du modèle hiérarchique

Les principales caractéristiques du modèle hiérarchique sont décrites dans ce qui suit :

1. il existe un ensemble de types d'enregistrements, qui représentent les ensembles d'entités du monde réel.
2. il existe un ensemble de relations reliant tous les types d'enregistrements dans un diagramme de structures de données arborescent, et qui représente les associations entre ensembles d'entités.
3. il n'existe qu'une seule relation entre deux types d'enregistrements. Cette relation ne peut être que de type 1..N. Il n'est pas nécessaire de la nommer.
4. les relations exprimées dans le diagramme de structures de données forment un arbre dont les « branches » pointent vers les « feuilles », qui sont les éléments de données.

On en déduit la définition du modèle hiérarchique de données.

5.1.4. Définition du modèle hiérarchique

Définition 5.1.6 (modèle hiérarchique).

Le modèle hiérarchique est un modèle de base de données qui organise logiquement les données selon les relations structurelles des arbres de définition hiérarchique. □

Un SGBD de type hiérarchique permet d'insérer, de modifier, de détruire et de recouvrer les occurrences d'enregistrements au sein de la base de données hiérarchique.

Les contraintes d'ajout et de destruction d'enregistrement sont les suivantes :

- chaque nouvel enregistrement ajouté à la base doit être relié à une occurrence d'un enregistrement parent.
- lorsqu'un enregistrement est détruit, toutes les occurrences des types d'enregistrements enfants le sont aussi.

Le modèle hiérarchique permet une représentation approchée du réel perçu au prix d'une redondance plus ou moins forte des données. Il impose un chemin d'accès unique et figé à chaque élément de données. Il n'y a donc pas indépendance des traitements et des données.

Il faut donc retenir :

- les relations entre parents et enfants sont obligatoirement 1..N (cf. figure 5.6).
- les structures qui traduisent ces relations sont fonction du type de requêtes à réaliser (cf. figure 5.7).

5.1.5. Limitations du modèle hiérarchique

On peut résumer les limitations du modèle hiérarchique par les points suivants :

- difficulté d'exprimer des relations de type $M..N$ (cf. figure 5.8) ;
- une modification de l'utilisation peut entraîner une perte d'efficacité de ce système ;
- l'espace de stockage est très important ;
- impossibilité d'exprimer des parents multiples ;

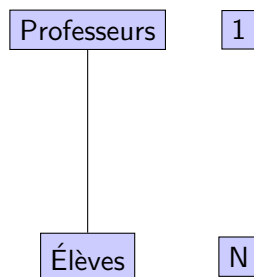


FIGURE 5.6. – Représentation hiérarchique parents-enfants

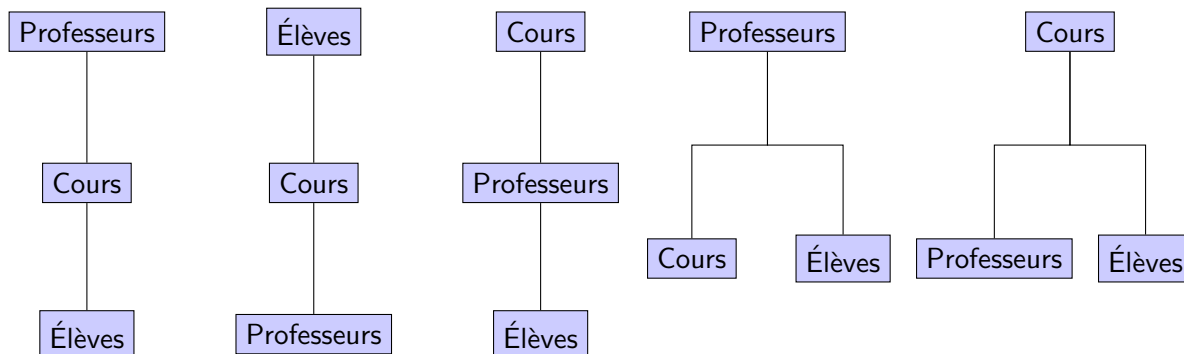


FIGURE 5.7. – Différentes représentations hiérarchiques associées à différentes requêtes

— difficulté d’exprimer des relations cycliques.

Par contre, ces bases de données sont réputées efficaces en consultation, lorsque peu de modifications surviennent. En effet, les algorithmes de consultation sont des algorithmes d’arbre qui sont efficace en termes de temps d’exécution.

5.1.6. Réalisations techniques

Les SGBD hiérarchiques les plus utilisés sur le marché sont les systèmes IMS et SYSTEM-2000.

IMS (*Integrated Management System*, IBM, 1966) fonctionne sur les systèmes d’exploitation MVS. Le LDD est constitué de macro instructions. Le LMD est autonome (et s’appelle DL/1) ou inclus dans COBOL, PL1 ou le langage d’assemblage. Des appels (CALL) à des modules DL/1 sont effectués à partir de ces langages. Ce système a servi pour le programme Apollo et pour suivre les factures des matériels sur le programme Saturne V. On le retrouve encore actuellement dans les systèmes utilisés par les distributeurs de billets de banque.

SYSTEM 2000 est aussi appelé S2K. Il fonctionne sur la plupart des gros systèmes BULL (sous GCOS), IBM (sous MVS), UNIVAC (sous EXEC 8) et CONTROL DATA (sous NON). Le LDD est autonome et s’appelle DEFINE. Le LMD est autonome ou inclus dans COBOL, FORTRAN ou le langage d’assemblage.

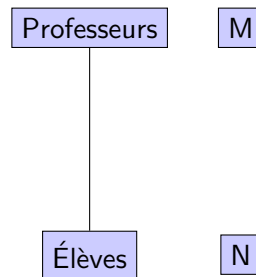


FIGURE 5.8. – Une association maillée qui ne peut pas être directement représentée dans le modèle hiérarchique

5.2. Le modèle réseau

Le modèle réseau, issu des travaux du CODASYL (*CO*nference on *DA*ta *SY*stems *LA*nguages), généralise le modèle hiérarchique [2, 32, 26]. Des relations multiples deviennent possibles et l'accès aux données n'est pas seulement ascendant ou descendant. Il est donc plus puissant que le modèle hiérarchique, mais également plus difficile à gérer.

5.2.1. Les spécifications du CODASYL

Le groupe de travail DBTG (*Data Base Task Group*) du CODASYL a proposé un certain nombre de spécifications qui ont servi de base à la définition du modèle réseau. Selon ce groupe, les objectifs d'un SGBD sont de :

- permettre de structurer les données de la manière la plus adaptée pour chaque programme en évitant la redondance de données ;
- de permettre à plusieurs programmes d'utiliser en concurrence la même base de données ;
- de permettre l'emploi de plusieurs techniques de recouvrement dans l'ensemble de la base ou dans un de ses sous-ensembles ;
- de protéger la base de données contre les accès non autorisés ;
- de centraliser l'emplacement physique des données ;
- de rendre les programmes indépendants de la structure physique des données ;
- de permettre l'utilisation de structures variées, allant d'enregistrements sans lien entre eux à des enregistrements regroupés sous forme de réseau ;
- de permettre aux utilisateurs de se servir des données sans se préoccuper des détails techniques liés aux relations entre enregistrements ;
- de rendre les programmes d'application aussi indépendants que possible de la structure de données ;
- de séparer la description de l'ensemble des données de la base des données effectivement accessibles à un utilisateur ;
- de fournir une description de la base qui puisse être utilisée par plus d'un langage de traitement ;
- de posséder une architecture permettant un interface entre la base de données et

5. Modèles classiques de BD

plusieurs langages.

A partir de ces propositions de spécifications pour un SGBD, le DBTG a défini :

1. un langage de définition des données (DDL) dont l'objet est de définir l'ensemble des données à stocker dans la base. Cette définition se nomme schéma. Il est le résultat d'un texte source écrit en DDL et est indépendant des applications.
2. la possibilité de créer à partir du schéma des sous-schémas, qui sont des sous-ensembles logiques du schéma. Le sous-schéma ne décrit que les données propres à une application et ignore celles qui n'y sont pas impliquées. Il sert à adapter la description des données à différents langages de traitement.
3. un langage de manipulation des données (DML) lié au langage utilisé dans les programmes d'application. Le DML proposé par le DBTG est un langage hôte du COBOL.
4. la notion d'ensemble, liée aux notions d'enregistrements propriétaire et membre.

5.2.2. Description

L'approche qui conduit à construire le modèle réseau consiste à introduire dans le modèle hiérarchique le concept de référence. Ce concept introduit plus de souplesse dans la description des données.

Ce modèle permet de représenter des relations d'attributs dans un ensemble d'entités et des associations entre ensembles d'entités. Formellement, un lien L_{ij} définit une connexion entre deux types d'enregistrements E_i et E_j . Pour chaque enregistrement x de type E_i , le lien identifie un ensemble d'enregistrements de type E_j et réciproquement. C'est de cette façon que sont représentées les associations entre ensembles d'entités. Ces liens pourront être de type 1..1, 1.. N ou M .. N . Dans le cas du modèle réseau, ce dernier type de lien est représenté comme étant deux relations de type 1.. N .

Une référence sera un pointeur sur un enregistrement. Cela permet d'éliminer les redondances qui auraient été présentes dans le modèle hiérarchique.

Par ailleurs, on pourra supprimer une référence par simple suppression du pointeur et sans supprimer les enregistrements référencés. Pour l'utilisateur, le réseau est transparent et selon le traitement à effectuer, l'utilisateur travaillera sur plusieurs arborescences qui peuvent varier.

5.3. Définitions

On peut caractériser un SGBD de type réseau par les définitions ci-dessous :

— élément de données (*data item*)

L'élément de données est la plus petite unité de données logique dans la base qui se caractérise par un nom et des attributs (type, longueur).

— enregistrement (*record*)

Un enregistrement est le regroupement d'un ou plusieurs éléments de données.

- type d'enregistrement (*record type*)
Un type d'enregistrement est un ensemble d'enregistrements de même nature qui décrivent les éléments d'un même ensemble d'entités.
- ensemble (*set*)
Un ensemble représente le lien entre deux types d'enregistrements. Chaque ensemble possède un nom unique et plusieurs ensembles d'un même type peuvent exister. Un ensemble relie un type d'enregistrement propriétaire (*owner*), situé en « tête » du lien et un type d'enregistrement membre situé en « bout » de lien.
- type d'ensemble (*set type*)
Un type d'ensemble possède un type d'enregistrement propriétaire et un ou plusieurs types d'enregistrement membre. Chaque type d'ensemble est défini indépendamment de tout autre type d'ensembles.
- espace (*area*)
L'espace désigne un sous-ensemble de la base, ce qui implique le stockage de chaque type d'enregistrements dans un espace. Plusieurs types d'enregistrement peuvent être stockés dans le même espace.
- schéma (*schema*)
Le schéma décrit la base de données en fonction de son contenu et de sa structure. Il y a un schéma de base, décrit par le DDL, indépendamment des langages utilisés pour accéder aux données (DML).
- sous-schéma (*subschema*)
Le sous-schéma est un sous-ensemble logique du schéma décrivant uniquement les types d'enregistrements nécessaires à une application particulière, ainsi que les types d'ensembles et les espaces devant être accessibles à cette application. Il peut y avoir plusieurs programmes d'application utilisant le même sous-schéma et plusieurs sous-schémas dans la base.
- le LMD (Langage de Manipulation de Données)
Le langage de manipulation est un langage souvent procédural. Il permet le parcours du réseau. Il se compose de deux types d'instructions :
 - déplacement d'un pointeur à l'intérieur du réseau avec positionnement d'autres pointeurs appelés pointeurs « courants »,
 - manipulation des contenus d'enregistrements par transfert dans une zone de travail définie à l'intérieur du programme.
 Par ailleurs, nous y trouvons les instructions classiques d'ajout, de suppression et de recherche d'une occurrence d'enregistrements.
Deux autres instructions (propres au modèle réseau) permettent la connexion et la déconnexion d'une occurrence d'enregistrement par rapport à une occurrence d'ensemble.

5.3.1. Réalisations techniques

De nombreux SGBD ont été réalisés en appliquant le modèle réseau : IDS, IDMS, ADABAS, DMS/1100, DBMS, TOTAL. Ces produits existent encore pour la plupart.

- ADABAS (*Adaptable DATA Base System*)

5. Modèles classiques de BD

Les relations collatérales et circulaires y sont possibles. Il n'y a pas d'équivalence totale avec le schéma CODASYL mais il y a indépendance des données et des programmes. Le LDD et le LMD (ADASCRIPT) sont autonomes. Le LMD peut être inclus dans les langages COBOL, FORTRAN ou PL/1 ou le langage d'assemblage sous forme de *call*.

— IDS (*Integrated Data Store*)

Il fonctionne sur les machines BULL et s'utilise principalement à partir de COBOL.

— IDMS (*Integrated Database Management System*)

Son LDD est autonome. Il offre des relations parentales, collatérales et circulaires. Son LMD est inclus dans COBOL.

— DMS/1100

Il est exactement conforme aux recommandations du CODASYL. Le LDD est autonome. La mise à jour de la base de données est possible avec restructuration. Le LMD est inclus dans COBOL, FORTRAN ou le langage d'assemblage.

— TOTAL

Ses exigences de taille mémoire sont modestes. Le LDD (DDL) est autonome, le LMD est dans COBOL, FORTRAN ou PL/1 ou le langage d'assemblage

5.4. Exemple d'un SGBD de type DBTG

Le modèle que nous présentons maintenant a été proposé par le groupe DBTG (*Database Task Group*) du CODASYL. Il permet la séparation entre :

- le langage de définition des données ;
- le langage de manipulation des données ;
- et la séparation entre les niveaux interne, conceptuel et externe.

5.4.1. Les notions de base

Le modèle réseau proposé utilise les notions de base suivantes :

- le *set* ou *coset* (pour *CODASYL set*). Un *set* exprime une relation 1..N entre enregistrements ;
- l'enregistrement ou *record* ;
- le champ ou *data item*.

Il y a distinction entre types et occurrences. Les occurrences représentent les valeurs correspondant aux types.

Les types sont structurés, i.e. on utilise certains types pour en construire d'autres :

- le type enregistrement est un ensemble de types de champs ;
- le type *set* est un ensemble de types d'enregistrements.

Pour obtenir la structure des occurrences, il suffit de remplacer le mot « type » par « occurrence ».

Enfin, quelques autres notions sont définies :

- appartenance d'une occurrence de *set* ;
- déconnexion et connexion d'une occurrence de *set* ;
- zone ou *area* : regroupe un ensemble d'enregistrements.

5.4.2. Le langage de définition des données

Ce langage se décompose en trois parties principales qui sont :

1. la définition du schéma de la base de données qui consiste à définir :
 - les types d'enregistrements
 - les types de set
2. la définition de sous-schémas qui représentent des sous-ensembles de types d'enregistrements et de types de *set*.
3. la définition des structures physiques et leur répartition dans chaque zone.

5.4.3. Généralités sur le langage de manipulation des données

Le langage de manipulation de données est un langage de type procédural permettant un parcours en réseau (comme dans les listes chaînées). Ce langage permet :

- de déplacer un pointeur (*currency*) dans le réseau ;
- de manipuler les contenus des enregistrements ;
- d'autres fonctions classiques d'ajout, de suppression et de modification existent aussi dans ce type de langage.

5.4.4. Exemple de construction de modèle et LDD

Considérons maintenant l'exemple des fournisseurs et des produits. Le schéma entité-association associé est donné sur la figure 5.9.

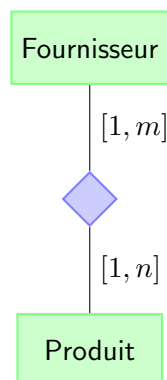


FIGURE 5.9. – Diagramme entité-association fournisseur/produit

Après décomposition, le modèle DBTG donne le diagramme présenté sur la figure 5.10. L'enregistrement « FP », dit *connection record*, a été introduit pour représenter l'association maillée. Dans ce cas nous obtenons deux autres liens qui sont :

1. F-FP dont le propriétaire est Fournisseur et qui est membre de FP ;
2. P-FP dont le propriétaire est Produit et qui est membre de FP.

5. Modèles classiques de BD

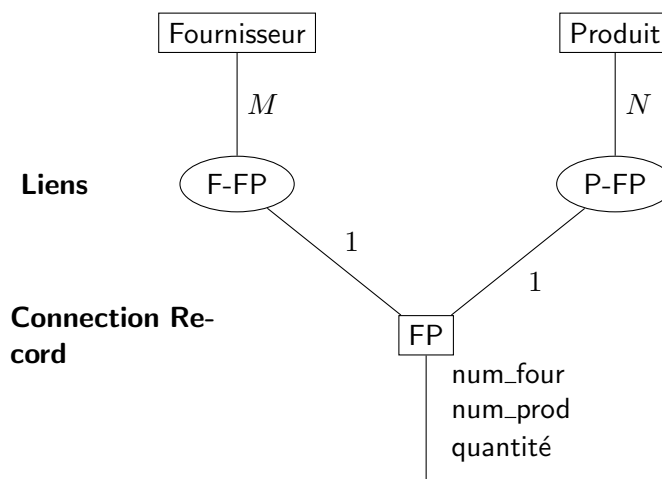


FIGURE 5.10. – Représentation du schéma entité-association de la figure 5.9 dans le modèle réseau

Déclaration en LDD

En utilisant le schéma de la figure 5.10, nous obtenons la définition des enregistrements et des *sets*.

Description des enregistrements

```
SCHEMA NAME IS FOUR_PROD.
```

```
RECORD NAME IS FOUR.
```

```
DUPLICATES ARE NOT ALLOWED
```

```
FOR FNO IN FOUR.
```

```
FNO; TYPE IS CHARACTER 5.
```

```
FNOM; TYPE IS CHARACTER 20.
```

```
STATUT; TYPE IS FIXED DECIMAL 3.
```

```
VILLE; TYPE IS CHARACTER 20.
```

```
RECORD NAME IS PROD.
```

```
DUPLICATES ARE NOT ALLOWED
```

```
FOR PNO IN PROD.
```

```
PNO; CHAR (6).
```

```
PNOM; CHAR (20).
```

```
COULEUR; CHAR (10).
```

```
POIDS; TYPE IS FIXED DECIMAL 4. DEFAULT IS -1.
```

```
VILLE; TYPE IS CHAR 20.
```

```
RECORD NAME IS FP
```

```

DUPLICATES ARE NOT ALLOWED
                FOR PNO IN FP , FNO IN FP.
FNO;           TYPE IS CHARACTER  5.
PNO;           CHAR (6).
QTE;           TYPE IS FIXED DECIMAL 5.
```

Description des sets

Les ensembles associés aux différents liens que sont P-FP et F-FP sont définis par :

```
SET NAME IS F-FP.
```

```

OWNER IS FOUR.
ORDER IS SORTED BY DEFINED KEYS.
DUPLICATES ARE NOT ALLOWED.
MEMBER IS FP.
INSERTION IS AUTOMATIC.
RETENTION IS FIXED.
KEY IS ASCENDING PNO IN FP.
SET SELECTION IS BY VALUE OF FNO IN FOUR.
```

```
SET NAME IS P-FP.
```

```

OWNER IS PROD.
ORDER IS SORTED BY DEFINED KEYS.
DUPLICATES ARE NOT ALLOWED.
MEMBER IS FP.
INSERTION IS AUTOMATIC.
RETENTION IS FIXED.
KEY IS ASCENDING FNO IN FP.
SET SELECTION IS BY VALUE OF PNO IN PROD.
```

Remarque : Sur la définition des sets nous remarquons les utilisations de :

- **INSERTION AUTOMATIC**. Elle indique la possibilité d'insertion automatique d'occurrences. Sinon, elle est manuelle avec **INSERTION MANUAL**.
- **RETENTION IS FIXED**, pour indiquer que les membres du *set* ne peuvent être membre que de ce *set*. Elle peut être **MANDATORY** (pour indiquer que les membres doivent être obligatoirement membre d'un *set*) ou **OPTIONAL**. □

5.5. Exemples de manipulation de données

5.5.1. Manipulation des données

Deux concepts fondamentaux permettent la navigation dans le réseau créé par le schéma défini dans le LDD :

5. Modèles classiques de BD

- le *curseur*, appelé également *CURRENCY*. Il possède les caractéristiques suivantes :
 - il désigne le pointeur courant ;
 - il représente un indicateur ;
 - il est une clé dans la base de données ;
 - il existe une table de *currency indicators* qui nous renseigne sur les valeurs des pointeurs courants pour toutes les navigations en cours ;
 - les curseurs sont gérés par le SGBD.
- la *navigation* CODASYL. Elle caractérise essentiellement le langage de manipulation des données via l'utilisation du curseur défini ci-dessus. Elle est représentée par des algorithmes puissants permettant de parcourir le graphpe qui implémente le réseau représentant les données de la base de données.

La navigation se fait grâce à quatre types de commandes qui sont :

- la recherche d'articles introduite par le mot clé **FIND** ;
- les échanges d'articles introduits par les mots clés **GET** et **STORE** ;
- les mises à jour introduites par les mots clés **ERASE** **CONNECT**, **DISCONNECT** et **MODIFY** ;
- le contrôle introduit par les mots clés **READY** et **FINISH**.

On appelle **RUN UNIT** la zone de données qui joue le rôle de buffer courant, c'est-à-dire celle sur laquelle pointe **CURRENCY**.

5.5.2. Quelques mots clés du langage de manipulation de données

Voici quelques mots clés du LMD. Nous les détaillerons sur des exemples dans les sections suivantes.

- **FIND** localise une occurrence de *record*. Elle positionne le pointeur **CURRENCY**.
- **GET** récupère la donnée dans la **RUN UNIT**.
- **STORE** crée un nouveau *record* dans la **RUN UNIT**.
- **ERASE** efface le *record* dans la **RUN UNIT**.
- **MODIFY** modifie le *record* dans la **RUN UNIT**.
- **CONNECT** connecte la **RUN UNIT** courante dans une occurrence de set.
- **DISCONNECT** déconnecte la **RUN UNIT** courante d'une occurrence de set.
- **RECONNECT** déconnecte une occurrence de set, puis la reconnecte à une autre occurrence de set.

Utilisation de **GET**

```
MOVE 'F4' TO FNO IN FOUR.  
FIND ANY FOUR USING FNO IN FOUR.  
GET FOUR.
```

ou bien

```
FIND FOUR OCCURRENCE FOR 'F4'.  
GET FNOM IN FOUR, STATUT IN FOUR.
```

5.5. Exemples de manipulation de données

permettent de récupérer les fournisseurs ayant pour numéro 'F4'. Dans le deuxième cas, on ne sélectionne que le nom et le statut.

Utilisation de STORE

Soit à créer l'occurrence de l'enregistrement FP suivante :

```
F5      P6      700
```

Il faut transférer les différents éléments dans le conteneur de données (la *current unit*) par l'ordre MOVE puis ranger les données dans la base par l'ordre STORE.

```
MOVE 'F5' TO FNO IN FP.  
MOVE 'P6' TO PNO IN FP.  
MOVE 700 TO QTE IN FP.  
  
MOVE 'F5' TO FNO IN FOUR.  
MOVE 'P6' TO PNO IN PROD.  
  
STORE FP.
```

Utilisation de ERASE

Soit à supprimer 'F3' dans FOUR.

```
FIND FOUR OCCURRENCE FOR 'F3'.  
ERASE [ALL] FOUR.
```

Attention, l'option [ALL] détruit tous les descendants même dans un *set*.

Utilisation de MODIFY

Soit à ajouter 10 au statut du fournisseur F3.

```
FIND FOUR OCCURRENCE FOR 'F3'.  
GET FOUR.  
ADD 10 TO STATUT IN FOUR.  
MODIFY FOUR.
```

Utilisation de CONNECT, DISCONNECT et RECONNECT

Les commandes précédentes agissent sur des occurrences de *records*. Les commandes CONNECT, DISCONNECT et RECONNECT agissent sur des occurrences de *set*. Voici leur syntaxe :

```
CONNECT [type_d_article] TO nom_d_ensemble.  
DISCONNECT [type_d_article] FROM nom_d_ensemble.  
RECONNECT [type_d_article] WITHIN nom_d_ensemble.
```


5. Modèles classiques de BD

Par exemple, on cherche à connecter une occurrence de S pour une valeur S4 dans l'occurrence SETX possédée par l'occurrence x de X.

```
FIND ... x.  
FIND S OCCURRENCE FOR S4.  
CONNECT S TO SETX.
```

```
FIND ... x.  
FIND S OCCURRENCE FOR S4.  
RECONNECT S WITHIN SETX.
```

```
FIND S OCCURRENCE FOR S4.  
DISCONNECT S FROM SETX.
```

Navigation avec les différents FIND

La forme générale d'une instruction FIND est donnée par :

```
FIND ANY rec USING rub IN rec.
```

Par exemple, si l'on recherche un fournisseur correspondant au numéro F4 :

```
MOVE 'F4' TO FNO IN FOUR.  
FIND ANY FOUR USING FNO IN FOUR.
```

Les enregistrements dupliqués

Pour obtenir les enregistrements dupliqués, on utilise la construction FIND DUPLICATES combinée avec une structure itérative. Par exemple :

```
MOVE 'NO' TO NOTFOUND.  
PERFORM UNTIL NOTFOUND='YES'.  
  GET S  
  ...  
  FIND DUPLICATES FOUR USING FNO IN FOUR.  
END PERFORM.
```

Recherche du propriétaire d'un ensemble (owner d'un set)

La recherche du propriétaire d'un *set* est réalisée par l'utilisation du mot clé OWNER.

```
FIND OWNER F-PP.
```

Accès séquentiel dans un set

Considérons l'exemple suivant où l'on souhaite trouver les valeurs de numéros de produits PNO pour le fournisseur de numéro F3.

```
MOVE 'F3' TO FNO IN FOUR.  
FIND ANY FOUR USING FNO IN FOUR.  
MOVE 'NO' TO EOF.  
FIND FIRST FP WITHIN F-FP.  
PERFORM UNTIL EOF ='YES'.  
GET FP.  
ADD PNO IN FP TO RESULT_LIST.  
...  
FIND NEXT FP WITHIN F-FP.  
END PERFORM.
```

RESULT_LIST est agrégat qui permet de représenter les résultats intermédiaires de la recherche.

Recherche avec duplication

Considérons l'exemple consistant à obtenir tous les produits fournis par le fournisseur de numéro F1 dont la quantité est égale à 100.

```
MOVE 'F1' TO FNO IN FOUR.  
FIND ANY FOUR USING FNO IN FOUR.  
MOVE 100 TO QTE IN FP.  
FIND FP WITHIN F-FP CURRENT USING QTE IN FP.  
MOVE 'NO' TO NOTFOUND.  
PERFORM UNTIL NOTFOUND='YES'  
GET FP.  
...  
FIND DUPLICATE WITHIN F-FP USING QTE IN FP.  
END PERFORM.
```

Imbrication de boucles

Considérons l'exemple consistant à trouver un autre produit de même fournisseur pour tous les fournisseurs du produit P4, et à imprimer le nom du fournisseur et le numéro de ce produit.

```
MOVE 'P4' TO PNO IN PROD.  
FIND ANY P USING PNO IN P.  
MOVE 'NO' TO EOF.  
PERFORM UNTIL EOF='YES'.  
    FIND NEXT FP WITHIN P-FP.  
    IF EOF NOT ='YES'
```

5. Modèles classiques de BD

```
FIND OWNER WITHIN F-FP.  
GET FOUR.  
MOVE 'NO' TO FOUND..  
PERFORM UNTIL FOUND ='YES'.  
    FIND NEXT FP WITHIN F-FP.  
    GET FP.  
    IF PNO IN FP NOT ='P4'  
        MOVE 'YES' TO FOUND.  
    END_IF.  
END_PERFORM.  
PRINT FNO IN FOUR, NOMFOUR IN FOUR, PNO IN FP.  
END_IF.  
END PERFORM.
```

6. Le modèle relationnel

Le modèle réseau est une évolution logique du modèle hiérarchique. Par rapport à ce dernier, le modèle réseau a permis l'élimination des redondances de données et la création de chemins d'accès multiples à une même donnée. Ces SGBD sont répandus dans les organisations qui utilisent des bases de données, car ils permettent une assez bonne adéquation entre les contraintes de fidélité au réel perçu et de simplicité d'utilisation. Cependant, il ne permet pas la prise en compte de phénomènes plus complexes et ne résout pas tous les problèmes d'indépendance des structures de données et des traitements. C'est pourquoi une troisième approche pour la résolution de ces problèmes a conduit aux modèles de quatrième génération : les modèles relationnels.

6.1. Description générale

Le modèle relationnel est dû à Codd [11, 12] et a pris une importance grandissante dans le domaine du traitement et le stockage des données informatiques. La principale raison de son succès est sa simplicité technique : les relations manipulées par ce modèle sont des *tableaux à deux dimensions*, au sens le plus ordinaire du terme, avec lignes et colonnes. C'est pourquoi ce modèle est qualifié de *plat*.

Les éléments contenus à l'intersection d'une ligne et d'une colonne sont les données stockées. Un sous ensemble d'une ligne ou d'une colonne constitue une information. Tout le problème est de bien définir les variables qui constituent ces tableaux. Cela relève de l'analyse fonctionnelle et qui peut être parfois difficile à résoudre (car on veut modéliser le monde réel).

Ce modèle dispose lui aussi d'un langage de manipulation de données (LMD) et un langage de description de données (LDD). Un vocabulaire spécifique, issu en grande partie de la théorie des ensembles, accompagne la description de ce type de bases de données. Dans ce chapitre, nous étudierons en détail ce modèle.

6.2. Un exemple intuitif

Nous allons illustrer les bases de données relationnelles en étudiant le modèle relationnel associé à la base de données gérant un parc informatique. Nous obtenons les relations suivantes représentées sous la forme de tableaux (ou tables) :

Ordinateur

6. Le modèle relationnel

<i>ref</i>	<i>nb_pos</i>	<i>capa_mem</i>	<i>nb_lect</i>	<i>nb_disk</i>	<i>prix</i>	<i>type</i>
10	1	512	2	1	10000	Micral 75
12	1	256	2	0	8000	Goupil G4
25	1	128	1	1	30000	Mac II

Type Ordinateur

<i>type</i>	<i>cons</i>	<i>pays</i>	<i>rev</i>	<i>nb_postes</i>	<i>mem_max</i>	<i>uc</i>	<i>capa_lect</i>	<i>capa_disk</i>
Micral 75	Bull	France	Camif	1	512	IN80486	1044	40
Mac II	Apple	USA	Apple	1	256	Mo68020	1044	60

Système

<i>nom_sys</i>	<i>concept</i>	<i>provenance</i>
MS_DOS	Microsoft	USA
UNIX	AT&T	USA

Logiciel de base

<i>type</i>	<i>nom_sys</i>
Micral 75	MS_DOS
Micral 75	UNIX
Micral 75	OS/2
Mac II	UNIX

Logiciel

<i>nom</i>	<i>classe</i>	<i>concept</i>	<i>provenance</i>	<i>revendeur</i>	<i>prix</i>
Turbo Pascal	Compil	Borland	USA	Camif	1000
Oracle	SGBD	Oracle	USA	Oracle	15000

Configuration

<i>nom_log</i>	<i>nom_sys</i>	<i>type</i>	<i>mem_min</i>	<i>disk_min</i>
Turbo Pascal	MS_DOS	Goupil G4	256	0
Oracle	UNIX	Micral 75	1500	1

Installation

<i>ref</i>	<i>nom_log</i>	<i>nom_sys</i>
10	Turbo Pascal	MS_DOS
10	DBaseIV	MS_DOS
25	Oracle	UNIX

Compatibilité

<i>type_ref</i>	<i>type_comp</i>
Micral 75	Goupil G4
Mac II	Mac Classic
IBM PS2	Micral 75

6.3. Définitions

Le but du modèle relationnel est de percevoir une réalité sous la forme de tableaux de valeurs appelés relations¹ :

- une relation a un nom ;
- une colonne d'une relation est un *attribut* ;
- une ligne d'une relation est un *n-uplet* ou un *tuple* ;
- l'ordre des lignes et des colonnes n'a pas d'importance (car c'est une relation ensembliste).

On doit pouvoir interpréter chaque ligne d'un tel tableau à l'aide d'une phrase simple et cohérente.

6.3.1. Domaine

Dans un premier temps, puisque nous allons travailler avec des relations, il faut définir sur quels ensembles les relations vont porter. On introduit pour cela la notion de *domaine*.

Définition 6.3.1.

Un domaine est un ensemble de valeurs que peut prendre un élément donné (souvent un attribut). Il sera représenté par un type. □

On remarquera que pour le modèle relationnel, les types utilisés doivent être atomiques : entiers, réels, chaînes de caractères etc ou sous-ensembles de ces types. On ne peut pas utiliser par exemple comme type une structure complexe comme une liste ou un ensemble.

6.3.2. Relations

On peut fournir deux définitions équivalents d'une relation. Ces deux définitions sont dites extensionnelles, car on explicite le contenu des relations.

1. On parlera également de tables

6. Le modèle relationnel

Définition 6.3.2.

Soient \mathcal{R} une relation et $\mathcal{D}_1, \dots, \mathcal{D}_n$ les domaines utilisés pour définir les colonnes de la relation (des domaines peuvent être identiques). Alors \mathcal{R} peut être définie comme :

- un sous-ensemble du produit cartésien $\mathcal{D}_1 \times \dots \times \mathcal{D}_n$
- un ensemble $\{t_1, \dots, t_m\}$ de n -uplets de valeurs tel que $\forall i \in \{1, \dots, m\} t_i = (d_{i,1}, \dots, d_{i,n})$ où pour tout $j \in \{1, \dots, n\}$ $d_{i,j}$ prend ses valeurs dans \mathcal{D}_j .

On dira que m est la cardinalité de \mathcal{R} .

On dira que n est l'ordre de \mathcal{R} . □

Pour représenter une relation, on utilise la notation sous forme de tableau suivante :

	\mathcal{D}_1	...	\mathcal{D}_i	...	\mathcal{D}_n
t_1	$d_{1,1}$...	$d_{1,i}$...	$d_{1,n}$
...
t_j	$d_{j,1}$...	$d_{j,i}$...	$d_{j,n}$
...
t_m	$d_{m,1}$...	$d_{m,i}$...	$d_{m,n}$

Exemple 6.3.1 : La représentation de la relation *Logiciel de base* sera un sous ensemble du produit cartésien des 2 domaines *Types d'ordinateurs* et *Noms de systèmes*. Voici le produit cartésien des deux domaines (une croix indique que le couple formé appartient à la relation) :

	Micral 75	Mac II	Goupil G4
MS_Dos	×		
UNIX	×	×	
OS/2	×		

La relation peut également être représenté sous la forme d'un tableau comportant 4 n -uplets :

	<i>type</i>	<i>nom_sys</i>
t_1	Micral 75	MS_DOS
t_2	Micral 75	UNIX
t_3	Micral 75	OS/2
t_4	Mac II	UNIX

□

On peut remarquer que comme les relations sont des ensembles de n -uplets, l'ordre de représentation des tuples importe peu. La représentation d'une relation n'est donc pas unique.

6.3.3. Attributs et clés

La notion d'attribut dans le modèle relationnel est la même que celle vue dans le modèle entité-association (cf. chapitre 2). L'attribut représente donc une propriété.

Définition 6.3.3.

Un attribut est une fonction nommée d'une relation dans un de ses domaines. \square

Par exemple, dans la relation *Logiciel de base*, l'attribut *type* est une fonction vers l'ensemble des chaînes de caractères représentant les types d'ordinateurs.

Une case situé sur la colonne de l'attribut *Att* d'une relation contiendra donc une valeur d'attribut prise dans le domaine de l'attribut *Att*.

Un attribut ou un groupe d'attributs dont la connaissance d'une valeur identifie un n-uplet de la relation de façon unique est appelé *superclé* d'une relation.

Définition 6.3.4.

Soit \mathcal{R} une relation possédant n attributs Att_1, \dots, Att_n . L'ensemble non vide d'attributs $\{Att_{c_1}, \dots, Att_{c_m}\} \subseteq \{Att_1, \dots, Att_n\}$ est une superclé de \mathcal{R} si et seulement si :

Étant donnés deux tuples $t_1 = (d_{1,1}, \dots, d_{1,n})$ et $t_2 = (d_{2,1}, \dots, d_{2,n})$ de \mathcal{R} , alors $t_1 = t_2$ si et seulement si $\forall i \in \{1, \dots, m\} d_{1,c_i} = d_{2,c_i}$. \square

D'après cette définition, on peut en déduire les propriétés suivantes :

- toute relation possède au moins une superclé. Cette clé peut être éventuellement constituée de l'ensemble de ses attributs ;
- tous les attributs d'une relation dépendent fonctionnellement des superclés de la relation (cf. chapitre 11 sur les dépendances fonctionnelles) ;
- si deux n-uplets contiennent la même valeur de clé, alors ils sont identiques.

Si une superclé est minimale pour l'inclusion, on parle alors de *clé* de la relation.

Définition 6.3.5.

Soit $\{Att_{c_1}, \dots, Att_{c_m}\}$ une superclé d'une relation \mathcal{R} . Si $\forall i \in \{1, \dots, m\} \{Att_{c_1}, \dots, Att_{c_m}\} - \{Att_{c_i}\}$ n'est pas une superclé de \mathcal{R} , alors $\{Att_{c_1}, \dots, Att_{c_m}\}$ est une clé de \mathcal{R} . \square

Une relation peut bien évidemment avoir plusieurs clés.

6.3.4. Schéma relationnel d'une base de données

Avant de donner la définition du schéma d'une base de données, nous définissons le schéma d'une relation de façon intensionnelle.

Définition 6.3.6.

Le schéma d'une relation est définie de façon intensionnelle par l'énoncé de :

- son nom ;
- ses attributs ;
- des domaines de valeurs de ses attributs ;

6. Le modèle relationnel

On note des deux façons suivantes la relation \mathcal{R} , d'attributs Att_1, \dots, Att_n de domaines respectifs $\mathcal{D}_1, \dots, \mathcal{D}_n$:

$$\begin{aligned} \mathcal{R} \{Att_1 : \mathcal{D}_1, \dots, Att_n : \mathcal{D}_n\} \\ \mathcal{R}(\{Att_1 : \mathcal{D}_1, \dots, Att_n : \mathcal{D}_n\}) \quad \square \end{aligned}$$

On peut remarquer que le fait d'utiliser des ensembles pour les attributs d'une relation montre bien que l'ordre importe peu. Il peut y avoir plusieurs schémas pour une même relation. Lorsque les domaines sont omis, on dit qu'on obtient un schéma simplifié.

Lors de la construction du schéma d'une relation, on spécifie également la clé primaire de la relation.

Définition 6.3.7.

Soit \mathcal{R} une relation. La clé primaire de \mathcal{R} est une clé $\{Att_1, \dots, Att_m\}$ de \mathcal{R} . Pour représenter cette clé primaire, on souligne les attributs correspondants dans la relation. \square

Exemple 6.3.2 : Par exemple, le schéma simplifié associé à la relation *Configuration* est le suivant :

$$\text{Configuration } \{\underline{type}, \underline{nom_syst}, \underline{nom_log}, \underline{mem_min}, \underline{disk_min}\} \quad \square$$

Nous venons donc de voir la définition en intension du schéma d'une relation. Une définition en extension d'une relation est définie par l'ensemble de ses tuples.

Nous pouvons maintenant définir le schéma relationnel d'une base de données.

Définition 6.3.8.

Le schéma relationnel d'une base de données est défini par l'ensemble des schémas des relations qui composent la base et un ensemble de contraintes appelé contraintes d'intégrité. \square

Les contraintes d'intégrité seront définis plus en détail dans le chapitre 9. On peut toutefois citer les contraintes correspondant aux *clés étrangères* qui spécifient qu'un ou plusieurs attributs d'une relation doivent prendre leurs valeurs non pas dans leur domaine en entier, mais dans les valeurs d'attributs apparaissant dans une autre relation.

Définition 6.3.9.

Soit $R(\Delta)$ une relation. On dit que $\{Att_1, \dots, Att_n\} \subseteq \Delta$ sont des clés étrangères référant les attributs $\{Att'_1, \dots, Att'_n\}$ d'une relation R' ssi les valeurs prises par le tuple $\langle Att_1, \dots, Att_n \rangle$ ne peuvent être que des valeurs du tuple $\langle Att'_1, \dots, Att'_n \rangle$ apparaissant dans des n -uplets de R' . On le note de la façon suivante :

$$\begin{aligned} R \{ \dots, \underline{Att_1}, \dots, \underline{Att_n} \} \\ \{Att_1, \dots, Att_n\} \text{ référence } R' \{Att'_1, \dots, Att'_n\} \quad \square \end{aligned}$$

Exemple 6.3.3 : Par exemple, les valeurs d'attributs apparaissant dans la colonne *nom_sys* de la relation *Logiciel de base* ne peuvent apparaître que comme valeurs d'attribut de la colonne *nom_sys* de la relation *Système*. \square

Nous verrons dans 9 qu'un certain nombre de contraintes sont imposées sur les clés étrangères (en particulier, elles sont souvent des clés primaires de la relation référencée).

6.4. Conception du schéma relationnel à partir d'un diagramme entité-association

Nous avons vu dans le chapitre 2 que le modèle entité-association permettait de représenter facilement un ensemble de données structuré. Or, le modèle relationnel est celui qui est actuellement utilisé dans les SGBD actuels (en particulier parce qu'il repose sur des fondements mathématiques solides). Le but de cette section est donc de présenter une approche formelle pour passer du schéma entité-association au modèle relationnel traduisant ce schéma.

6.4.1. Principe initial

Le schéma entité-association présente deux concepts fondamentaux : l'entité et l'association. A ces deux concepts nous associons deux types de relations :

- les *relations-entités* issues des classes d'entités ;
- les *relations-associations* issues des classes d'associations.

On propose une première règle de traduction dans ce qui suit.

Principe 6.4.1.

1. pour chaque classe d'entités, on définit une relation composée
 - des attributs et des domaines de la classe d'entités ;
 - de la clé primaire de la classe d'entités.
2. pour chaque classe d'associations, on définit une relation composée
 - des attributs et des domaines de la classe d'associations ;
 - de la clé primaire de la classe d'associations. Cette clé est composée des clés primaires de chacune des classes d'entités en liaison et des attributs propres de l'association. □

Exemple 6.4.1 : Considérons la base de données du parc informatique présentée comme exemple dans le chapitre 2. On trouve 4 relations-entités :

- *Ordinateur* { ref, nb_pos, ..., prix }
- *Type d'ordinateur* { type, cons, ..., capa_disk }
- *Logiciel* { nom, ..., prix }
- *Système* { nom_sys, ..., provenance }

On trouve 4 relations-associations :

- *Classe matériel* { ref, type }
- *Compatibilité* { type_ref, type_comp }
- *Logiciel de base* { type, nom_sys }
- *Configuration* { nom_log, nom_sys, type, ..., disk_min }
- *Installation* { ref, nom_log, nom_sys } □

6.4.2. Simplification du schéma

L'application immédiate de la règle initiale introduit souvent plus d'informations que ce qui est nécessaire. C'est pourquoi le schéma relationnel obtenu doit être simplifié.

Reprenons l'exemple du parc informatique. Dans la relation *Classe matériel*, une ligne de la relation peut être identifiée par la connaissance de la seule valeur de l'attribut *ref* sans la valeur de *type*. Cela veut dire que la superclé minimale de cette relation est *ref* qui est la même clé que dans la relation *Ordinateur*.

Pour cette raison, nous effectuons le regroupement des deux relations *Classe matériel* et *Ordinateur* en une seule. On obtient :

Ordinateur {*ref*, ..., *prix*, *type*}

Les questions qui se posent alors sont les suivantes :

- quand et comment effectuer les regroupements ?
- quels sont les attributs et les relations concernés par le regroupement ?

On peut proposer un certain nombre d'interprétations sur l'exemple précédent :

- *type* dépend *fonctionnellement* de *ref* et se comporte comme les autres attributs d'*Ordinateur*. C'est pour cette raison que *type* est un nouvel attribut d'*Ordinateur*.
- *type*, clé de *Type ordinateur*, dépend *fonctionnellement* de *ref*, clé d'*Ordinateur*, mais la cardinalité du rôle joué par la classe d'entités *Ordinateur* dans la classe d'associations *Classe matériel* est fonctionnelle, alors que celle du rôle joué par *Type ordinateur* est hiérarchique.
- on regroupe dans une même relation toutes les informations attribuées à un *Ordinateur* qui se présentent sous la forme de propriétés monovaluées.

On remarquera que *Ordinateur* n'est plus une relation entité, ni une relation association. De plus, toujours dans *Ordinateur*, *type* est une clé étrangère.

6.4.3. Règles de traduction

Nous sommes en mesure de donner complètement les règles de traduction permettant de passer du schéma entité-association à un schéma de base de données relationnelle. Pour cela, nous allons définir pour chaque classe du modèle entité-association la traduction vers le modèle relationnel en la présentant sous forme d'algorithme.

Algorithme 6.4.1 : Algorithme permettant de traduire une classe d'entités en relation

entrée : une classe d'entités de nom *E*

sortie : une relation représentant la classe d'entités

1 construire une relation *R* :

- de nom *E*
- possédant pour attributs ceux de la classe d'entités *E*
- ayant pour clé primaire celle de la classe d'entités *E*

retourner *R* ;

L'algorithme 6.4.1 nous permet de traduire les classes d'entités en relations. Il faut

maintenant s'occuper des classes d'associations, en tenant compte des problèmes évoqués dans la section précédente. L'algorithme 6.4.2 présente la traduction associée.

Algorithme 6.4.2 : Algorithme permettant de traduire une classe d'associations en relation(s)

entrées : une classe d'associations A entre des classes d'entités de noms E_1, \dots, E_p
sortie : une relation correspondant à A ou une relation correspondant à une classe d'entité modifiée

```

1 pour  $i \in \{1, \dots, p\}$  faire
2   |  $r_i \leftarrow$  nom du rôle joué par  $E_i$  dans  $A$  ;
3 fin
4 si aucun des  $r_i$  n'est de cardinalité  $[0, 1]$  ou  $[1, 1]$  alors
5   | construire une relation :
6     | — de nom  $A$ 
7     | — d'attributs ceux de la classe d'association  $A$ 
8     | — possédant pour clé primaire le regroupement des clés étrangères que constituent les
9     |   clés des relations  $E_1, \dots, E_p$ 
6 sinon
7   | il existe  $r_j$  tq sa cardinalité soit  $[0, 1]$  ou  $[1, 1]$ . Dans ce cas :
8     | — ajouter dans  $E_j$  les attributs de la classe d'associations  $A$ 
9     | — ajouter dans  $E_j$  les clés des relations  $E_1, \dots, E_{j-1}, E_{j+1}, \dots, E_p$  comme clés
10    | étrangères
8 fin
9 retourner la relation  $A$  si elle existe ou la relation  $E_j$ 

```

On peut remarquer que s'il existe plusieurs classes d'entités dont le rôle a une multiplicité $[0, 1]$ ou $[1, 1]$, il suffit de choisir une de ces entités pour appliquer le bloc « sinon » de l'algorithme précédent.

Exemple 6.4.2 : Supposons que nous ayons le schéma entité-association présenté sur la figure 6.1.

On obtient alors comme schémas de relations :

$$\begin{aligned}
 E1 & \{ \underline{Att_{1,1}}, \dots, \underline{Att_{1,c_1}}, Att_{1,(c_1+1)}, \dots, Att_{1,n_1} \} \\
 E2 & \{ \underline{Att_{2,1}}, \dots, \underline{Att_{2,c_2}}, Att_{2,(c_2+1)}, \dots, Att_{2,n_2} \} \\
 E3 & \{ \underline{Att_{3,1}}, \dots, \underline{Att_{3,c_3}}, Att_{3,(c_3+1)}, \dots, Att_{3,n_3} \} \\
 E4 & \{ \underline{Att_{4,1}}, \dots, \underline{Att_{4,c_4}}, Att_{4,(c_4+1)}, \dots, Att_{4,n_4} \} \\
 A & \{ \underline{Att_{A,1}}, \dots, \underline{Att_{A,n_A}}, \underline{Att_{1,1}}, \dots, \underline{Att_{1,c_1}}, \underline{Att_{2,1}}, \dots, \underline{Att_{2,c_2}}, \underline{Att_{3,1}}, \dots, \underline{Att_{3,c_3}}, \\
 & \quad \underline{Att_{4,1}}, \dots, \underline{Att_{4,c_4}} \}
 \end{aligned}$$

□

Exemple 6.4.3 : Supposons que nous ayons le schéma entité-association présenté sur la figure 6.2.

On obtient alors comme schémas de relations :

6. Le modèle relationnel

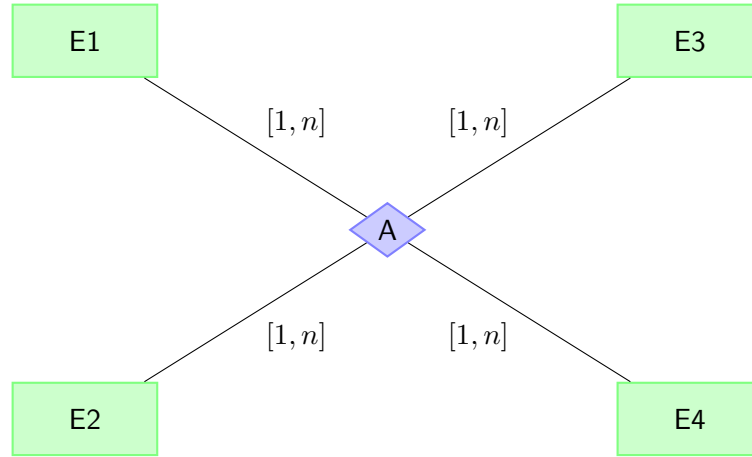


FIGURE 6.1. – Schéma entité-association sans cardinalité [1, 1]

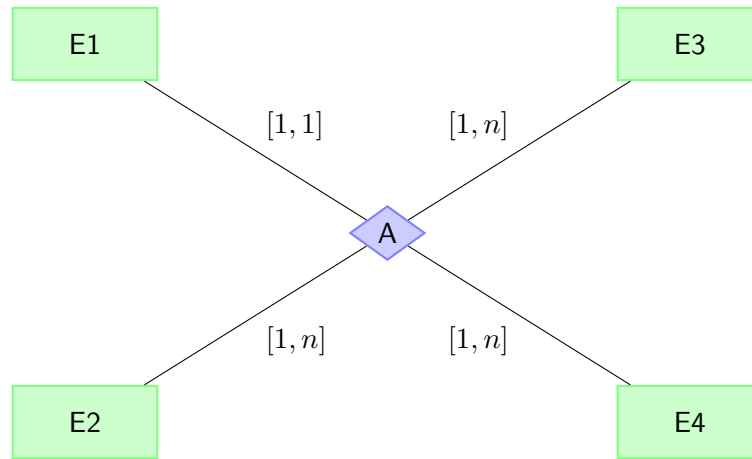


FIGURE 6.2. – Schéma entité-association avec cardinalité [1, 1]

- $E1$ $\{\underline{Att_{1,1}}, \dots, \underline{Att_{1,c_1}}, Att_{1,(c_1+1)}, \dots, Att_{1,n_1}, Att_{A,1}, \dots, Att_{A,n_A}, Att_{2,1}, \dots, Att_{2,c_2}, Att_{3,1}, \dots, Att_{3,c_3}, Att_{4,1}, \dots, Att_{4,c_4}\}$
- $E2$ $\{\underline{Att_{2,1}}, \dots, \underline{Att_{2,c_2}}, Att_{2,(c_2+1)}, \dots, Att_{2,n_2}\}$
- $E3$ $\{\underline{Att_{3,1}}, \dots, \underline{Att_{3,c_3}}, Att_{3,(c_3+1)}, \dots, Att_{3,n_3}\}$
- $E4$ $\{\underline{Att_{4,1}}, \dots, \underline{Att_{4,c_4}}, Att_{4,(c_4+1)}, \dots, Att_{4,n_4}\}$

□

Il se peut qu'en utilisant ces principes on obtienne des clés composées de plusieurs attributs qui soient simplifiables. Dans ce cas, il faut simplifier les clés et regrouper les éventuelles relations possédant des clés identiques issues de ces simplifications. L'algorithme 6.4.3 présente cette simplification.

Attention, il ne faut pas pour autant regrouper systématiquement deux relations ayant des attributs de noms identiques. Par exemple, les locations et les ventes de voitures sont

Algorithme 6.4.3 : Simplification d'un schéma relationnel

entrée : un schéma relationnel SR **sortie** : un schéma relationnel dont les clés sont simplifiées et les relations éventuellement regroupées

```

1 pour chaque relation  $r$  dont la clé primaire est composée de plusieurs attributs
  faire
2   | si la clé de  $r$  n'est pas minimale alors
3   |   | simplifier la clé de  $r$  ;
4   | fin
5 fin
6 si deux relations  $E_1$  et  $E_2$  ont la même clé primaire  $\{Att_1, \dots, Att_n\}$  alors
7   | regrouper  $E_1$  et  $E_2$  en une seule relation possédant :
8   |   — tous les attributs de  $E_1$  et  $E_2$  autres que la clé primaire
9   |   —  $\{Att_1, \dots, Att_n\}$  comme clé primaire
10 fin
11 retourner  $SR$ 

```

représentables par deux relations semblables :

- *Location* $\{Num_employ, Num_voiture, date\}$
- *Vente* $\{Num_employ, Num_voiture, date\}$

Le regroupement n'a ici aucun sens.

Enfin, nous présentons l'algorithme permettant de traduire les entités *faibles* en relations.

Algorithme 6.4.4 : Traduction d'une entité faible en relation

entrée : une entité faible E liée à des entités supports E_1, \dots, E_n par des associations supports A_1, \dots, A_p **sortie** : une relation R représentant E

```

1 placer les attributs de  $E$  dans  $R$  ;
2 pour chaque  $i \in \{1, \dots, n\}$  faire
3   | ajouter la clé de  $E_i$  à  $R$  ;
4 fin
5 retourner  $R$  ;

```

Cet algorithme correspond bien à l'idée intuitive que l'on se fait d'une entité faible. On remarquera que l'on ne s'occupe pas de la traduction des associations supports, car celles-ci ne seront pas traduites en vertu de l'algorithme 6.4.2.

7. Algèbre relationnelle

Le but de ce chapitre est de présenter les fondements des langages de manipulation de données dans le cadre du modèle relationnel. Nous présenterons les outils d'extraction et de modification du contenu d'une base de données relationnelle.

À cet effet, deux classes de langages sont définies :

- les langages procéduraux (impératifs) dans lesquels l'utilisateur exprime une requête¹ sous la forme d'un enchaînement d'instructions ou d'une procédure qui construit le résultat cherché ;
- les langages assertionnels (déclaratifs) dans lesquels l'utilisateur exprime une requête sous la forme d'une description du résultat recherché sans se préoccuper de la manière de le construire.

L'algèbre relationnelle est un langage procédural permettant d'exprimer des requêtes sur le contenu de relations. Ce langage n'est pas complet au sens de Turing : il existe des opérations que l'on ne peut pas exprimer en algèbre relationnelle (par exemple une « boucle » **for**). Cette caractéristique n'est pas rédhibitoire : la limitation du nombre d'opérations disponibles permet d'optimiser les requêtes écrites dans un langage déclaratif de haut niveau comme SQL, qui sera étudié au chapitre 8.

L'algèbre relationnelle est le langage de référence sur lequel s'appuient tous les autres langages de manipulation de bases de données relationnelles. Il fournit à ces langages une sémantique formelle. La définition des relations ayant été abordée au chapitre 6, nous nous intéresserons essentiellement au problème de la recherche d'informations ou de la consultation dans une base de données relationnelle.

7.1. Principe et définitions

L'algèbre relationnelle est une algèbre au sens mathématique « général » : il s'agit donc d'un ensemble d'éléments atomiques muni d'opérateurs. Les éléments atomiques de l'algèbre relationnelle sont des variables qui représentent des relations, et des constantes qui sont des relations finies. Les opérateurs ont pour opérandes des relations et *fourniront toujours une relation comme résultat*. On peut répartir ces opérateurs en quatre classes :

- les opérateurs ensemblistes classiques (union, intersection, différence) appliqués aux relations (cela ne pose pas problème, car une relation peut être représentée par un ensemble de tuples²). Ce sont des opérateurs d'arité deux ;
- les opérateurs permettant de retirer des parties de la relation : la *sélection* qui élimine certaines lignes de la relation et la *projection* qui élimine certaines colonnes.

1. La définition d'une requête sera donnée plus loin.

2. Tuple est un synonyme de n-uplet.

7. Algèbre relationnelle

Ce sont des opérateurs d'arité un ;

- les opérateurs qui combinent les tuples de deux relations, comme le produit cartésien ou les *jointures*. Ce sont des opérateurs d'arité deux ;
- un opérateur de renommage, qui permet de renommer une relation ou les attributs d'une relation.

Tous ces opérateurs sont *sans effets de bord* (ils ne modifient pas la valeur de leurs opérandes).

Les expressions de l'algèbre relationnelle sont appelées *requêtes*.

Dans ce qui suit, la syntaxe des opérateurs sera présentée de deux façon : une « mathématique » (notée (MA)) et une « informatique » (notée (IN)). Les deux notations sont tout autant valables l'une que l'autre. On définira la sémantique de ces opérateurs le plus clairement possible, même si on n'utilisera pas de notation formelle pour le faire.

Dans la suite du chapitre, nous considérerons les deux relations suivantes :

- R de schéma $\{A_1, A_2, \dots, A_n\}$;
- S de schéma $\{B_1, B_2, \dots, B_p\}$.

Pour tous les exemples de ce chapitre, nous nous appuyerons sur les relations développées dans le chapitre 6 (*Ordinateur, Logiciel* etc.).

7.2. Opérateurs ensemblistes

Les opérations ensemblistes classiques peuvent s'appliquer aux relations. Celles-ci peuvent en effet être représentées par l'ensemble des tuples qui les caractérisent. Pour pouvoir appliquer des opérations ensemblistes aux relations, nous allons poser l'hypothèse suivante :

Hypothèse 7.2.1.

Soient R et S deux relations. Pour appliquer une opération ensembliste à R et S , il faut et il suffit que :

- R et S aient des schémas avec des attributs identiques et de même domaine ;
- les colonnes de R et S doivent être ordonnées de la même façon. □

7.2.1. Union

Le résultat de l'union de R et de S est l'ensemble des tuples qui sont dans R ou dans S .

Syntaxe :

(MA)	$R \cup S$
(IN)	UNION (R, S)

□

7.2.2. Intersection

Le résultat de l'intersection de R et de S est l'ensemble des tuples qui sont à la fois dans R et dans S .

Syntaxe :

- (MA) $R \cap S$
 (IN) INTER(R, S)

□

7.2.3. Différence

Le résultat de la différence de R et de S est l'ensemble des tuples qui sont dans R mais pas dans S .

Syntaxe :

- (MA) $R - S$
 (IN) DIFF(R, S)

□

7.3. Projection

La projection est un opérateur qui permet à partir d'une relation R de créer une relation identique à R mais qui ne possède que certaines colonnes de R .

Syntaxe :

- (MA) $\pi_{A_i, A_j, \dots, A_l}(R)$
 (IN) PROJ(R, A_i, A_j, \dots, A_l)

□

La valeur de la requête $\pi_{A_i, A_j, \dots, A_l}(R)$ est donc une relation de schéma $\{A_i, A_j, \dots, A_l\}$ qui est la restriction de R aux colonnes A_i, A_j, \dots, A_l (dans cet ordre). Comme on ne travaille qu'avec des ensembles, les tuples identiques sont éliminés.

Exemple 7.3.1 : Considérons la requête ayant pour résultat la liste des numéros de référence et des types de tous les ordinateurs ?

Cette requête est la projection de la relation *Ordinateur* sur les attributs *ref* et *prix*, soit $\pi_{ref, prix}(Ordinateur)$. On obtient la relation suivante :

<i>ref</i>	<i>type</i>
10	Micral 75
11	Goupil G4
25	Mac II

□

7.4. Sélection

L'opérateur de sélection appliqué à une relation R produit une nouvelle relation définie à partir d'un sous-ensemble des tuples de R . Les tuples sélectionnés sont ceux qui vérifient une propriété P impliquant les attributs de R . P est une expression logique, dont la valeur est soit **vraie**, soit **faux**. On peut définir une expression logique formellement de la façon suivante :

7. Algèbre relationnelle

Définition 7.4.1.

Soient a et b deux attributs ou constantes. Alors :

- $a < b$, $a \leq b$, $a = b$, $a \geq b$, $a > b$, $a \neq b$ sont des termes ;
- si t_1 et t_2 sont des termes, t_1 ET t_2 et t_1 OU t_2 sont des expressions logiques ;
- si e_1 et e_2 sont des expressions logiques, e_1 ET e_2 et e_1 OU e_2 sont des expressions logiques. □

Nous avons utilisé ici les opérateurs arithmétiques classiques, mais on peut également utiliser d'autres opérateurs qui correspondent aux types des attributs de la relation.

Syntaxe : Soit P une expression logique impliquant les attributs de R . La syntaxe de l'opérateur de sélection est la suivante :

(MA) $\sigma_P (R)$
(IN) $\text{SEL}(R, P)$

□

Le résultat de la requête $\sigma_P (R)$ est une relation de schéma identique à celui de R , soit $\{A_1, A_2, \dots, A_n\}$. Les tuples présents dans le résultat de la requête sont ceux qui vérifient P .

Exemple 7.4.1 : Considérons la requête ayant pour résultat les caractéristiques des SGBD qui coûtent moins de 1000 F.

Cette requête est la sélection de tous les logiciels de classe SGBD et tels que leur prix est inférieur à 1000 F, soit $\sigma_{\text{classe} = \text{"SGBD"} \text{ ET } \text{prix} < 1000}$ (*Logiciel*). On obtient une relation de même schéma que *Logiciel* qui ne contient aucun tuple. □

On peut remarquer que les opérations de projection et de sélection sont des opérations d'extraction respectivement verticale et horizontale.

7.5. Produit cartésien

Le produit cartésien de deux ensembles « classiques » R et S est l'ensemble des paires formées en choisissant le premier élément de la paire dans R et le second élément dans S . Pour des relations, qui sont des ensembles de tuples, la paire créée à partir d'un tuple $t_R = (a_1, \dots, a_n)$ de R et d'un tuple $t_S = (b_1, \dots, b_p)$ de S est le tuple $(a_1, \dots, a_n, b_1, \dots, b_p)$ (on choisit arbitrairement de placer d'abord les éléments de t_R).

Syntaxe :

(MA) $R \times S$
(IN) $\text{PROD}(R, S)$

□

Exemple 7.5.1 : Considérons les deux tables R et S suivantes :

R	A	B	C
	a_1	b_1	c_1
	a_2	b_2	c_2

S	D	E
	d_1	e_1
	d_2	e_2
	d_3	e_3

Le résultat du produit cartésien $R \times S$ est la table suivante :

$R \times S$	A	B	C	D	E
	a_1	b_1	c_1	d_1	e_1
	a_1	b_1	c_1	d_2	e_2
	a_1	b_1	c_1	d_3	e_3
	a_2	b_2	c_2	d_1	e_1
	a_2	b_2	c_2	d_2	e_2
	a_2	b_2	c_2	d_3	e_3

□

Si deux attributs dans les relations R et S ont le même nom, par exemple A , on peut les distinguer en les préfixant par le nom de la relation : $R.A$ et $S.A$.

7.6. Jointures

Les opérateurs de jointures sont des opérateurs qui permettent de construire une relation à partir de deux relations qui ont une propriété commune (représentée par un attribut « commun »). Nous allons examiner différents types de jointures.

7.6.1. Jointure naturelle

La jointure naturelle de deux relations R et S consiste à ne sélectionner dans le produit cartésien de R et de S que les tuples tels que les attributs communs³ à R et S aient la même valeur.

Syntaxe :

- (MA) $R \bowtie S$
- (IN) **JOIN**(R , S)

□

Exemple 7.6.1 : Considérons les deux tables R et S suivantes :

3. On dit que deux attributs sont communs s'ils portent le même nom.

7. Algèbre relationnelle

$$R \quad \begin{array}{|c|c|} \hline A & B \\ \hline 1 & 2 \\ \hline 32 & 4 \\ \hline \end{array}$$

$$S \quad \begin{array}{|c|c|c|} \hline B & C & D \\ \hline 2 & 5 & 6 \\ \hline 4 & 7 & 8 \\ \hline 9 & 10 & 11 \\ \hline \end{array}$$

Le résultat de la jointure naturelle $R \bowtie S$ est la table suivante :

$$R \bowtie S \quad \begin{array}{|c|c|c|c|} \hline A & B & C & D \\ \hline 1 & 2 & 5 & 6 \\ \hline 3 & 4 & 7 & 8 \\ \hline \end{array}$$

□

7.6.2. θ -jointure

La jointure naturelle nous permet de « fusionner » deux relations possédant des attributs communs. Il peut être également intéressant de grouper deux relations n'ayant pas d'attributs communs en utilisant une propriété à vérifier par les tuples. Pour cela, on utilise une θ -jointure.

Syntaxe : Soit P une expression logique impliquant les attributs de R ou de S . La syntaxe de l'opérateur de θ -jointure est la suivante :

(MA) $R \bowtie_P S$
 (IN) **JOIN**(R, S, P)

□

Le résultat de $R \bowtie_P S$ est une relation de schéma $\{A_1, \dots, A_n, B_1, \dots, B_p\}$. Une jointure naturelle est une θ -jointure particulière (la condition porte sur l'égalité entre les attributs identiques de R et de S) à laquelle on rajoute une projection (pour éliminer les colonnes redondantes).

Théorème 7.6.1.

$$\bowtie_P = \sigma_P \circ \times$$

□

Ainsi, le résultat de $R \bowtie_P S$ est en fait équivalent au résultat de $\sigma_P(R \times S)$. Cette équivalence permet de mieux comprendre comment fonctionne la jointure.

Exemple 7.6.2 : Par exemple, considérons la requête ayant pour résultat la provenance des systèmes d'exploitation installables sur Micral 75.

La provenance des systèmes d'exploitation est définie par l'attribut *prov* de la relation *Système*. Les systèmes d'exploitation installables sur Micral 75 vont être définis par l'attribut *nom_sys* de la relation *Logiciel de Base*.

La requête considérée est donc :

$Système \bowtie_{type = "Micral75"} AND Système.nom_sys = Logiciel\ de\ Base.nom_sys\ Logiciel\ de\ Base$

7.7. Opérateur de renommage

à laquelle il faut « ajouter » une projection sur l'attribut *provenance*.

Si l'on utilise le théorème 7.6.1, on peut détailler l'obtention du résultat.

1. construction de *Système* × *Logiciel de Base* :

<i>Système.nom_sys</i>	<i>concept</i>	<i>provenance</i>	<i>type</i>	<i>Logiciel de Base.nom_sys</i>
MS_DOS	Microsoft	USA	Micral 75	MS_DOS
MS_DOS	Microsoft	USA	Micral 75	UNIX
MS_DOS	Microsoft	USA	Mac II	UNIX
UNIX	AT&T	USA	Micral 75	MS_DOS
UNIX	AT&T	USA	Micral 75	UNIX
UNIX	AT&T	USA	Mac II	UNIX

2. sélection des tuples suivant le critère *type* = "Micral75" AND *Système.nom_sys* = *Logiciel de Base.nom_sys* :

<i>Système.nom_sys</i>	<i>concept</i>	<i>provenance</i>	<i>type</i>	<i>Logiciel de Base.nom_sys</i>
MS_DOS	Microsoft	USA	Micral 75	MS_DOS
UNIX	AT&T	USA	Micral 75	UNIX

3. projection sur l'attribut *provenance* :

<i>provenance</i>
USA

□

7.7. Opérateur de renommage

L'opérateur de renommage permet de renommer une relation ou de changer le nom des attributs d'une relation *R*.

Syntaxe :

- (MA) $\rho_{S(A_{i_1}, \dots, A_{i_n})}(R)$
 (IN) $REN(R, S, (A_{i_1}, \dots, A_{i_n}))$

□

Le résultat de $\rho_{S(A_{i_1}, \dots, A_{i_n})}(R)$ est une relation de nom *S* qui possède les mêmes tuples que *R*, mais dont les attributs sont nommés A_{i_1}, \dots, A_{i_n} dans cet ordre.

7.8. Opérations de mise à jour

On peut, en utilisant les opérations ensemblistes, effectuer des mises à jour sur une relation :

- l'insertion d'un tuple se fera au moyen de l'opérateur d'union. Par exemple :

Logiciel de Base ∪ (MacII), MS_DOS)

insère un nouveau tuple dans la relation *Logiciel de Base* ;

7. Algèbre relationnelle

- la suppression d'un tuple se fera au moyen de l'opérateur de différence. Par exemple :
 $\text{Logiciel de Base} - (\text{MacII}, \text{UNIX})$
supprime un tuple dans la relation *Logiciel de Base* ;
- la modification de tuple se fait au moyen d'une suppression suivie d'une insertion.

Les opérateurs algébriques n'ont normalement pas d'effet de bord : on obtient donc une nouvelle relation à chaque opération de mise à jour. Il faut alors renommer la relation pour obtenir la relation initiale.

7.9. Représentation d'une requête

Une requête n'utilisant qu'un seul opérateur n'est pas souvent suffisante pour exprimer la question à laquelle on souhaite répondre. Il faut souvent utiliser la composition pour pouvoir y parvenir. Prenons comme exemple la question « *quels sont les types d'ordinateurs et les concepteurs associés tels que la mémoire maximale soit supérieure à 256 Mo et la capacité du disque soit supérieure à 30 Go ?* ». On peut écrire cette requête grâce à une projection, une intersection et deux sélections :

1. sélection sur la table *Type d'Ordinateur* des ordinateurs dont la mémoire maximale est supérieure à 256 Mo, soit $\sigma_{\text{mem_max} > 256}(\text{Type d'Ordinateur})$;
2. sélection sur la table *Type d'Ordinateur* des ordinateurs dont la capacité du disque est supérieure à 30 Go, soit $\sigma_{\text{capa_disk} > 30}(\text{Type d'Ordinateur})$;
3. intersection des deux relations obtenues en 2 et 3 ;
4. projection de la relation obtenue en 4 sur les attributs *type* et *concept*.

7.9.1. Représentation sous forme d'arbre

La requête précédente a donc pour forme complète :

$\pi_{\text{type, concept}}(\sigma_{\text{mem_max} > 256}(\text{Type d'Ordinateur}) \cap \sigma_{\text{capa_disk} > 30}(\text{Type d'Ordinateur}))$
ce qui est peu lisible.

La figure 7.1 propose une représentation sous forme d'arbre de la requête précédente beaucoup plus agréable à lire (mais beaucoup moins efficace en terme de calcul pour une machine).

7.9.2. Notation linéaire

En utilisant la notation « informatique » présentée dans la syntaxe des opérateurs, on peut également proposer une notation linéaire pour représenter une séquence de requêtes. Il suffit pour cela d'ajouter un opérateur d'affectation `:=` au langage. Par exemple :

```
R := SEL(Type Ordinateur, mem_max>256)
S := SEL(Type Ordinateur, capa_disk>30)
T := INTER(R,S)
Reponse := PROJ(T, type, concept)
```

On suppose ici que les variables R, S, T et Reponse ont les schémas adéquats.

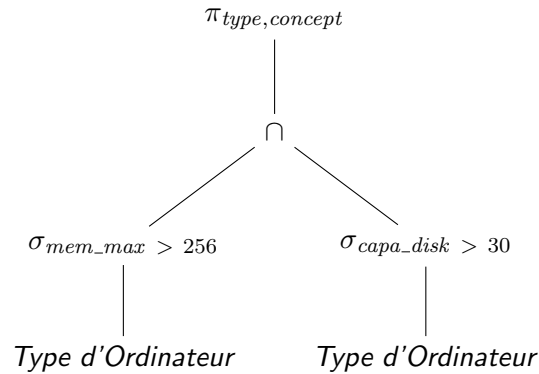


FIGURE 7.1. – Représentation sous forme d'arbre d'une requête

7.10. Opérateurs étendus

Nous présentons ici quelques opérateurs étendus de l'algèbre relationnelle. Ces opérateurs permettront de mieux comprendre les opérateurs du langage SQL (cf. chapitre 8).

7.10.1. Opérateurs d'agrégation

Les opérateurs d'agrégation permettent d'agréger (ou de « résumer ») les valeurs contenues dans une colonne d'une relation. Les opérateurs les plus utiles sont :

- **SUM** qui permet de sommer une colonne contenant des valeurs numériques ;
- **AVG** qui permet de moyenniser une colonne contenant des valeurs numériques ;
- **MIN** et **MAX** qui retournent respectivement la plus petite et la plus grande valeur d'une colonne ;
- **COUNT** retourne le nombre de valeurs d'une colonne.

Ces opérateurs *ne sont pas des opérateurs de l'algèbre relationnelle*. Ils ne renvoient pas une relation mais une valeur (numérique la plupart du temps). On parle d'« agrégation » car ils réduisent une colonne entière d'une relation à une valeur.

Exemple 7.10.1 : Considérons la relation suivante :

<i>A</i>	<i>B</i>
1	2
3	4
1	2

Voici les résultats de quelques agrégations appliquées à cette relation :

- **SUM**(*B*) = 8
- **AVG**(*A*) = 2,5
- **MIN**(*A*) = 1
- **COUNT**(*A*) = 3

□

7.10.2. Regroupements

Les opérateurs d'agrégation permettent de réduire une colonne d'une relation à une valeur. Il se peut parfois que l'on ne veuille pas appliquer un opérateur d'agrégation à l'ensemble d'une relation, mais l'appliquer séparément à des ensembles de tuples de la relation. Par exemple, on pourrait chercher quel est le prix le moins cher dans chaque classe de logiciel. Il faut regrouper les tuples de la relation *Logiciel* grâce à l'attribut *classe*, puis appliquer **MIN** sur les colonnes désignées par *prix* dans chacune des classes obtenues. L'opérateur de regroupement permet de grouper les tuples d'une relation suivant différentes classes en utilisant les attributs de la relation.

Syntaxe :

$$\gamma_L (R)$$

où L est une liste d'éléments qui peuvent être :

- un attribut de la relation R . On dit que c'est un *attribut de regroupement* ;
- un opérateur d'agrégation appliqué à un attribut de la relation. Le nom de l'attribut correspondant dans la relation résultat est précisé par le signe \rightarrow . L'attribut ainsi créé est appelé *attribut d'agrégat*.

□

Définition 7.10.1.

La relation retournée par $\gamma_L (R)$ est construite de la façon suivante :

1. *partitionnement des tuples de R . Chaque partition est composée des tuples ayant un assignement particulier pour les attributs de regroupement de L ;*
2. *pour chaque groupe, construire un tuple constitué :*
 - a) *des attributs de regroupement de L ;*
 - b) *des agrégations calculées sur les tuples du groupe des attributs d'agrégats de L*

□

Il faut bien remarquer que l'on ne construit qu'un seul tuple par groupe.

Exemple 7.10.2 : La requête présentée en début de section peut s'écrire :

$\gamma_{classe, \text{MIN}(prix) \rightarrow prix_bas} (Logiciel)$. On obtient alors :

<i>classe</i>	<i>prix_bas</i>
Compil	1000
SGBD	15000

□

7.10.3. Jointures externes

Lorsque l'on fait une jointure sur deux relations, les tuples des relations qui ne satisfont pas la propriété demandée n'apparaissent pas dans le résultat de la jointure. En effectuant une jointure externe, ces tuples apparaissent également dans le résultat. On complète la valeurs des attributs de la seconde relation par une valeur particulière notée \perp . Il existe plusieurs types de jointures externes :

Syntaxe :

- $R \overset{\circ}{\bowtie}_P S$ est une jointure externe sur R et S suivant la condition P ;
- $R \overset{\circ}{\bowtie}_L P S$ est une jointure externe sur R et S suivant la condition P . On ne considère que les tuples de R pour la jointure externe ;
- $R \overset{\circ}{\bowtie}_R P S$ est une jointure externe sur R et S suivant la condition P . On ne considère que les tuples de S pour la jointure externe

□

Exemple 7.10.3 : Considérons les deux relations suivantes :

	<i>A</i>	<i>B</i>	<i>C</i>
<i>R</i>	1	2	3
	4	5	6
	7	8	9

	<i>B</i>	<i>C</i>	<i>D</i>
<i>S</i>	2	3	10
	2	3	11
	6	7	12

Le résultat de la jointure naturelle externe $R \overset{\circ}{\bowtie} S$ est la table suivante :

	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
$R \overset{\circ}{\bowtie} S$	1	2	3	10
	1	2	3	11
	4	5	6	⊥
	7	8	9	⊥
	⊥	6	7	12

□

7.11. Conclusion

L'algèbre relationnelle permet d'exprimer de façon très simple des requêtes sur des relations. On ne dispose en effet que d'une seule structure de contrôle, la composition (on n'a ni branchement conditionnel, ni répétition).

Par contre, il existe plusieurs solutions possibles pour construire une requête. Par exemple, considérons la question suivante : « *quel sont les nombres de lecteurs de disquettes des ordinateurs en possédant plus de 2 ?* ». La réponse à cette question peut être exprimée comme le résultat de deux requêtes différentes :

$$\pi_{nb_lec} (\sigma_{nb_lec \leq 2} (Ordinateur)) \quad (7.1)$$

$$\sigma_{nb_lec \leq 2} (\pi_{nb_lec} (Ordinateur)) \quad (7.2)$$

Considérons que les opérations ont un coup d'exécution unitaire. Supposons qu'il y ait n tuples dans *Ordinateur* et m tuples vérifiant l'expression logique ($m \leq n$). Dans le cas

7. Algèbre relationnelle

de la requête (7.1), on va effectuer n opérations de comparaison pour la sélection, puis m opérations de projection (il ne reste que m tuples). Le « coût » de la requête (7.1) est donc $m + n$. Dans le cas de la requête (7.2), on va effectuer n opérations de projection, puis n opérations de comparaison pour la sélection (il reste encore n tuples). La requête (7.1) est donc moins « coûteuse » que la requête (7.2), alors qu'elle fournit le même résultat. En utilisant l'algèbre relationnelle, il faudrait donc choisir parmi toutes les requêtes possibles la plus efficace. Nous allons voir dans le chapitre 8 que des langages de haut niveau comme SQL permettent d'optimiser automatiquement le coût computationnel des requêtes.

8. Étude d'un langage assertionnel : SQL (Structured Query Language)

L'algèbre relationnelle présentée dans le chapitre 7 est un moyen puissant de calcul sur les relations. Elle permet de construire une relation contenant les éléments sur lesquels porte une requête sur une base de données. L'algèbre relationnelle peut être assimilée à un langage impératif : on doit décrire *comment* est obtenue la relation « réponse ».

L'objectif d'un langage assertionnel est d'offrir à l'utilisateur un moyen d'exprimer une requête en ne décrivant que le résultat recherché à l'aide d'une assertion (expression logique) sans devoir expliquer la façon de le trouver. En d'autres termes, l'assertion exprime le « quoi » et non le « comment ». Dans ce cas, l'utilisateur qualifie les résultats et le système compose la procédure de recherche.

Les objets manipulés par SQL sont les suivants :

- des tables : relations ou vues externes (cf. section 8.3.6) ;
- des colonnes (*columns*) attributs de « base » des relations ou des vues externes ou des attributs calculés à partir des attributs de base (via des agrégations par exemple) ;
- des lignes (*rows*) qui représentent les tuples.

Remarque : L'algèbre relationnelle que nous avons présentée précédemment manipulait les relations comme des ensembles de tuples. Dans SQL, l'algèbre relationnelle manipule en fait des multi-ensembles (*bags* en anglais) qui sont des ensembles dans lesquels les éléments peuvent avoir plusieurs occurrences. L'utilisation de multi-ensembles permet d'« optimiser » les requêtes. Par exemple, si l'on considère l'union de deux ensembles R et S , on est obligé de comparer tout tuple de S avec ceux de R pour vérifier qu'il n'apparaît pas déjà dans R . L'utilisation des multi-ensembles dans les opérations intermédiaires permet donc d'accélérer les calculs. □

Des compléments sur ce chapitre peuvent être trouvés dans [7, 8, 18, 6].

8.1. Historique et présentation

Il existe plusieurs dialectes de SQL. Tout d'abord, il existe trois versions standards de SQL :

- une version ANSI (*American National Standard Institute*) ;
- un standard mis à jour en 1992 appelé SQL-92 ou SQL2 ;
- une dernière version, notée SQL :1999, qui étend SQL2 avec des notions « objets » et d'autres caractéristiques.

8. SQL (*Structured Query Language*)

La plupart des versions commerciales de SQL fournies par les SGBD commerciaux respectent le standard ANSI et se conforment à la plupart des extensions de SQL2. Une petite partie de SQL :1999 est intégrée et, la plupart du temps, ces versions de SQL possèdent leurs propres extensions. Dans ce chapitre, nous étudierons principalement la version ANSI de SQL et SQL2. Si nous présentons des instructions présentes dans SQL99, nous le précisons. SQL99 sera présenté rapidement dans le chapitre 13.

Le langage SQL fonctionne selon deux modes :

- un mode interprété, i.e. dans lequel SQL est vu comme un langage à part entière. Nous présenterons SQL principalement selon ce mode ;
- un mode intégré, dans lequel SQL est intégré à un autre langage de programmation (C, Java, Cobol etc.). Nous présenterons dans le chapitre 10 l'interface entre SQL et Java.

On peut distinguer dans SQL trois sous-langages particuliers :

- le langage de manipulation de données (DML pour *Data Manipulation Language*), qui permet d'obtenir des informations et de les mettre à jour ;
- le langage de description de données (DDL pour *Data Description Language*), qui permet de créer des relations, de modifier leur schéma etc.
- le langage de contrôle des données (DCL pour *Data Control Language*) qui permet de restreindre l'accès des données.

Les principaux mots-clés de ces trois langages sont les suivants :

DDL	DML	DCL
ALTER	DELETE	GRANT
CREATE	INSERT	REVOKE
COMMENT	SELECT	
DESCRIBE	UPDATE	
DROP		
RENAME		

Nous allons présenter ces trois langages dans ce qui suit. Nous considérerons une relation R d'attributs A_1, \dots, A_n .

8.2. Le langage de manipulation de données

Le langage de manipulation de données permet soit d'obtenir une information à partir d'une ou plusieurs relations, soit de mettre à jour des tuples d'une relation. Dans le cas de l'obtention d'information, le résultat est édité sous la forme d'un tableau.

8.2.1. Bloc de qualification

La structure de base est le bloc de qualification ¹.

Syntaxe (bloc de qualification) :

1. SQL est un langage insensible à la casse des caractères, que ce soit pour ses mots clés ou pour les noms de relations ou d'attributs.

```

SELECT Ai, ..., An -- colonnes et agregations
FROM R             -- relation
WHERE F            -- assertion
GROUP BY A        -- regroupement
HAVING H          -- assertion
ORDER BY T        -- tri
;

```

□

Les clauses **GROUP**, **ORDER** et **HAVING** sont optionnelles. Nous pouvons détailler informellement la sémantique des clauses **FROM**, **WHERE** et **SELECT** :

- **FROM** permet de préciser quelles sont la ou les relations sur lesquelles on pose la requête ;
- **WHERE** permet de poser une assertion qui doit être vérifiée par les lignes de la table solution ;
- **SELECT** désigne les attributs des tuples qui doivent apparaître dans la solution.

Le bloc de qualification « simple » (i.e. ne contenant que les clauses **SELECT**, **FROM** et **WHERE**) est donc une combinaison d'une projection et d'une sélection.

Remarque : le bloc de qualification se termine toujours par un « ; ».

□

8.2.2. Projection

La syntaxe d'une projection simple est présentée dans ce qui suit.

Syntaxe (projection) :

```

SELECT Ai, ..., Ap
FROM R;

```

□

Exemple 8.2.1 : La requête « *quelle est la liste des numéros de référence et des types de tous les ordinateurs ?* » s'exprime de la façon suivante :

```

SELECT ref, type
FROM Ordinateur;

```

Remarquons que l'on peut obtenir des tuples identiques, en particulier si on ne sélectionne pas la clé primaire parmi les colonnes de la projection. Pour éviter cela, on peut utiliser le mot clé **DISTINCT** (on voit ici apparaître le fait qu'on travaille avec des multi-ensembles dans SQL).

Syntaxe (projection ensembliste) :

```

SELECT DISTINCT Ai, ..., Ap
FROM R;

```

□

8. SQL (Structured Query Language)

8.2.3. Sélection

Syntaxe (sélection) :

```
SELECT *  
FROM R  
WHERE F;
```

□

Ici, « * » désigne l'ensemble des colonnes de la relation R . F est une expression logique permettant de sélectionner les colonnes.

Expression logique de sélection

L'expression logique peut contenir des constantes et des noms de colonnes de la relation R . On la construit avec les opérateurs suivants :

- <, <=, =, >=, >, <>;
- **AND**, **OR**, **NOT**;
- **IN**, par exemple **IN** ('MS_DOS', 'UNIX');
- **BETWEEN**, par exemple **BETWEEN** 15 **AND** 30;
- **LIKE** en utilisant un pattern et des jokers (cf. ce qui suit).

Les chaînes de caractères peuvent être comparées en utilisant l'opérateur **LIKE** qui permet de comparer une chaîne de caractères avec un pattern utilisant les symboles :

- **_** qui remplace n'importe quel caractère;
- **%** qui remplace n'importe quelle séquence de caractères;

Par exemple 'DUPONT'**LIKE** 'D%ONT' renverra **TRUE**, 'DUPONT'**LIKE** 'D__T' renverra **FALSE**, et 'DOIT'**LIKE** 'D__T' renverra **TRUE**. Si l'on veut utiliser les caractères **_** ou **%** dans une chaîne de caractères, on peut utiliser le mot clé **ESCAPE** suivi d'un caractère qui représentera un caractère d'échappement (par exemple **pourcentage LIKE '100x%' ESCAPE 'x'**).

La valeur NULL

La valeur **NULL** est une valeur qui peut être affectée à certains attributs lorsque :

- on ne sait pas quelle est la valeur de l'attribut;
- aucune valeur ne peut convenir pour l'attribut;
- on ne peut pas connaître la valeur de l'attribut (par exemple un numéro de téléphone).

Il faut noter que **NULL** n'est pas une constante. On ne peut pas l'utiliser explicitement comme opérande dans la clause **WHERE**. Si **NULL** apparaît dans une comparaison, on peut obtenir une valeur de vérité **UNKNOWN** pour l'expression considérée (qui est considérée pour le retour de la requête comme s'il s'agissait d'une valeur de vérité **FALSE**). La sémantique exacte de **NULL** est encore de nos jours sujette à débat.

8.2.4. Entêtes de colonnes

L'entête des colonnes de la table résultat d'une requête est soit le nom de l'attribut correspondant, soit une expression de « calcul » contenant le nom d'un attribut. Pour une meilleure présentation, on peut changer le nom des colonnes de la table résultat.

Syntaxe (entêtes de colonnes) :

```
SELECT col1 AS 'Nouveau nom 1', col2 AS 'Nouveau nom 2'  
FROM R  
...  
;
```

L'entête du résultat affiché portera les noms **Nouveau nom 1** et **Nouveau nom 2** au lieu de **col1** et **col2**.

Exemple 8.2.2 : On peut demander quels sont les prix en euros des ordinateurs :

```
SELECT prix / 6.55957 AS prix_euro  
FROM Ordinateur ;
```

Une application importante de cette fonctionnalité est la personnalisation par rapport à une langue donnée (internationalisation). Un exemple classique est celui des catalogues multilingues qui sont issues d'une unique base de données.

8.2.5. Tris des colonnes

Pour fournir un résultat trié suivant certaines colonnes, il existe une clause particulière, la clause **ORDER BY**.

Syntaxe (tri) :

```
ORDER BY <listes attributs>
```

où chaque élément de la liste est un attribut suivi de **ASC** (ordre ascendant) ou **DESC** (ordre descendant). □

Exemple 8.2.3 : La requête permettant d'éditer le catalogue des ordinateurs dans l'ordre des prix décroissants et des capacités de mémoire croissantes est :

```
SELECT *  
FROM Ordinateur  
ORDER BY prix DESC, capa_mem ASC ;
```


8.2.6. Opération de produit

Nous allons maintenant nous intéresser aux opérations impliquant plus d'une relation. La première opération à laquelle nous allons nous intéresser est le produit de relation.

Syntaxe (produit) :

```
SELECT A1, ..., Ap
FROM R1, R2, ..., Rk
WHERE F;
```

□

Si l'on considère t_1, \dots, t_k des tuples respectifs de R_1, \dots, R_k , le tuple (t_1, \dots, t_k) sera qualifié s'il satisfait la formule logique F . La table résultat sera obtenue en effectuant une projection de (t_1, \dots, t_k) sur A_1, \dots, A_p . La requête est donc semblable à $\pi_{A_1, \dots, A_p}(\sigma_F(R_1 \times \dots \times R_k))$.

Dans le cas d'attributs provenant de tables différentes et portant le même nom, on les distingue en les préfixant par le nom de la relation.

Exemple 8.2.4 : La requête « *quels sont les types, prix et constructeurs de la machine numéro 10 ?* » s'écrit :

```
SELECT Ordinateur.type, Ordinateur.prix,
        TypeOrdinateur.const
FROM Ordinateur, TypeOrdinateur
WHERE (Ordinateur.ref = 10) AND
        (Ordinateur.type = TypeOrdinateur.type);
```

8.2.7. Noms d'alias

Les noms d'alias sont des noms symboliques donnés à une table. Ils sont déclarés dans la clause **FROM**. Ces noms constituent des variables de parcours d'une table. Le type d'une variable nom d'alias associé à une table est celui d'une ligne de cette même table.

Syntaxe (alias) :

```
SELECT N1.A1, ..., Nn.An
FROM R1 (AS) N1, ..., Rn (AS) Nn
WHERE (N1.A1 = ...);
```

□

On travaille toujours avec des noms de variable dans une requête. Par défaut, si l'on ne précise pas un nom d'alias, le nom utilisé est celui de la relation. Par exemple, **FROM R** est en fait **FROM R AS R**.

Exemple 8.2.5 : Considérons la requête « *quels sont les types, prix et constructeurs des ordinateurs de référence 10 ?* » est exprimée de la façon suivante :

```
SELECT O.type, O.prix, T.const
FROM Ordinateur O, TypeOrdinateur T
WHERE (O.ref = 10) AND (O.type = T.type);
```

Les variables O et T représentent respectivement un tuple de la relation **Ordinateur** et un tuple de la relation **TypeOrdinateur**. □

8.2.8. Jointures

On peut exprimer différents tous les types de jointures avec SQL (jointure naturelle, θ -jointure, jointure externe). Nous allons les détailler rapidement dans ce qui suit.

Produit cartésien

La forme la plus simple de jointure en SQL est ce que nous avons appelé le produit cartésien en algèbre relationnelle.

Syntaxe (produit cartésien) :

```
R CROSS JOIN S
```

□

Cette expression renvoie une table équivalent au produit cartésien de R et de S . En cas d'attributs portant le même nom, ils seront préfixés par le nom de la relation.

θ -jointure

La θ -jointure est la forme la plus générale de jointure. Rappelons que l'on peut la voir comme la composition d'un produit cartésien et d'une sélection.

Syntaxe (θ -jointure) :

```
R JOIN S ON F
```

□

R JOIN S ON F établit la θ -jointure de R et S sur le critère F (qui est une expression logique).

Le résultat de cette expression est évidemment une relation. On peut donc l'utiliser dans une sélection. Par exemple, la requête correspondant à l'exemple 7.6.2 est :

```
SELECT provenance
FROM Systeme JOIN LogicielBase ON
    type = 'Micral 75' AND Systeme.nom_sys = LogicielBase.nom_sys ;
```

8. SQL (Structured Query Language)

Autres types de jointures

Les autres jointures possibles sont :

- les jointures naturelles en préfixant **JOIN** par **NATURAL**. Il faut bien entendu que les deux relations aient un attribut commun et dans ce cas il n'y a pas de condition **ON**. Par exemple, la requête précédente aurait pu s'écrire :

```
SELECT provenance  
FROM Systeme NATURAL JOIN LogicielBase ;
```

- certains SGBD autorisent l'utilisation du mot-clé **USING** pour permettre de simplifier l'écriture d'une jointure dont la condition porte sur l'égalité entre attributs portant le même nom. Par exemple, la requête précédente aurait pu s'écrire :

```
SELECT provenance  
FROM Systeme JOIN LogicielBase USING nom_sys ;
```

La différence la syntaxe précédente est que l'utilisation de **NATURAL** impose une égalité sur *tous* les attributs portant le même nom.

- les jointures externes qui sont de trois types :
 - les jointures externes complètes par **FULL OUTER JOIN** ;
 - les jointures externes à gauche par **LEFT OUTER JOIN** ;
 - les jointures externes à droite par **RIGHT OUTER JOIN**.

Les jointures externes permettent d'utiliser toutes les tuples d'une ou des deux relations, même si certains tuples ne devraient pas apparaître dans la jointure normale (on dit interne). On complète alors les valeurs manquantes dans ces tuples par **NULL**.

Par exemple, si l'on considère les deux relations suivantes :

	A	B
R1	-10.5	15
	-20.1	30

	B	C
R2	15	toto
	40	titi

alors voici les résultats de quelques requêtes comportant des jointures, interne et externes :

- **SELECT * FROM R1 JOIN R2 ;**

A	B	C
-10.5	15	toto

On remarquera que l'on peut préciser explicitement que l'on fait une jointure interne en utilisant **SELECT * FROM R1 INNER JOIN R2 ;**.

- **SELECT * FROM R1 LEFT OUTER JOIN R2 ;**

A	B	C
-10.5	15	toto
-20.1	30	NULL

— **SELECT * FROM R1 RIGHT OUTER JOIN R2 ;**

A	B	C
-10.5	15	toto
NULL	40	titi

— **SELECT * FROM R1 FULL OUTER JOIN R2 ;**

A	B	C
-10.5	15	toto
-20.1	30	NULL
NULL	40	titi

8.2.9. Opérateurs ensemblistes

Les opérateurs ensemblistes s'appliquent à des ensembles dont les données sont compatibles (même type). Contrairement aux autres opérateurs, les opérateurs ensemblistes conservent la structure des relations.

Intersection, union et différence peuvent s'appliquer aux résultats de deux **SELECT** qui retournent les mêmes colonnes ou les attributs de même domaine. Ces trois opérateurs sont des opérateurs binaires. Ils prennent en paramètre deux relations compatibles ayant la même structure.

Syntaxe (opérateurs ensemblistes) :

```
R UNION S
R INTERSECT S
R EXCEPT S
```

□

Exemple 8.2.6 : Considérons la requête « *quels sont les types d'ordinateurs compatibles à la fois avec un Micral 75 et un Mac II ?* ». Cette requête s'exprime de la façon suivante :

```
(SELECT type_comp FROM Compatibilite
 WHERE type_ref = 'Micral 75')
INTERSECT
(SELECT type_comp FROM Compatibilite
 WHERE type_ref = 'Mac II');
```

Remarque : les opérateurs ensemblistes suivi du mot-clé **ALL** renvoient un multi-ensemble et non un ensemble. □

8.2.10. Opérateurs d'agrégation

Nous avons présenté les opérateurs d'agrégation dans la section 7.10.1. Ils permettent d'effectuer des opérations sur certaines colonnes d'une relation. On retrouve dans SQL les mêmes opérateurs qu'en algèbre relationnelle étendue :

- **SUM** qui permet de sommer une colonne contenant des valeurs numériques ;
- **AVG** qui permet de moyenniser une colonne contenant des valeurs numériques ;
- **MIN** et **MAX** qui retournent respectivement la plus petite et la plus grande valeur d'une colonne ;
- **COUNT** retourne le nombre de valeurs d'une colonne.

Si l'on note $OP1, OP2, \dots, OPk$ des opérateurs d'agrégations, la syntaxe de leur utilisation est la suivante :

Syntaxe (opérateur d'agrégation) :

```
SELECT OP1(Ai), ..., OPk(Aj)
FROM R
WHERE F;
```

□

Seul l'opérateur **COUNT** peut s'utiliser avec ***** : **COUNT(*)** permet de compter tous les tuples résultats d'une requête.

Exemple 8.2.7 : La requête suivante permet de récupérer le prix moyen des logiciels :

```
SELECT AVG(prix)
FROM Logiciel ;
```

□

8.2.11. Regroupements

Nous avons également vu les regroupements dans la section 7.10.2. Les regroupements permettent de créer des classes distinctes dans les tuples d'une relation suivant un critère et d'appliquer un opérateur d'agrégation à chacune de ces classes.

Syntaxe (regroupement) :

```
SELECT A1, OP1(A2)
FROM R
WHERE F
GROUP BY P ;
```

□

GROUP BY doit se trouver après la clause **WHERE**. **P** est une liste d'attributs de regroupement, définie comme suit :

Définition 8.2.1.

Soit

```
SELECT A1, ..., Ap  
FROM R  
WHERE F  
GROUP BY P ;
```

une requête SQL où chaque A_i est soit un attribut, soit un opérateur d'agrégation appliqué à un attribut. Alors seuls les attributs apparaissant dans P peuvent apparaître sans opérateur d'agrégation dans la clause **SELECT**. □

Ceci correspond bien à la définition donnée en section 7.10.2 qui définissait la relation renvoyée par un opérateur de regroupement. En effet, si l'on autorisait des attributs n'apparaissant pas dans la clause **GROUP BY** à apparaître comme opérandes de l'opérateur de sélection, quelle valeur de ces attributs choisir lorsque l'on construit la table résultat ?

Exemple 8.2.8 : Considérons la requête suivante : « quelle est la liste des types de machines installées, avec pour chaque type le nombre d'exemplaire de machines ? ». Elle s'exprime en SQL par :

```
SELECT type, COUNT(ref)  
FROM Ordinateur  
GROUP BY type;
```

□

Les agrégations sont bien entendu calculées en tenant compte des regroupements.

8.2.12. Clause HAVING

La clause **HAVING** permet d'appliquer un critère de sélection aux lignes d'une partition créée par **GROUP BY**. On peut la voir comme le pendant de **WHERE** qui travaille sur la relation entière. Similairement à la clause **SELECT**, seuls les attributs apparaissant dans **GROUP BY** peuvent apparaître non agrégés dans **HAVING**.

Syntaxe (sélection sur les regroupements) :

```
SELECT A1, ..., Ap  
FROM R  
WHERE F  
GROUP BY P  
HAVING L;
```

□

Exemple 8.2.9 : Considérons la requête suivante : « quelle est la liste des types de machines installées, avec le nombre d'exemplaires, dont la capacité mémoire de chaque exemplaire atteint ou dépasse 512 Ko et dont le nombre d'exemplaires installé atteint ou dépasse 2 ? ». En ajoutant un tri par nombre croissant d'exemplaires, la requête s'écrit :

8. SQL (Structured Query Language)

```
SELECT O.type AS 'Type', COUNT(O.ref) AS 'Nombre exemplaires'
FROM Ordinateur O
WHERE capa_mem >= 512
GROUP BY O.type
HAVING count(O.ref) >= 2
ORDER BY 2 ASC;
```

ORDER BY 2 permet de préciser d'ordonner les tuples résultats suivant la deuxième colonne. □

8.2.13. Sous-requêtes

Dans SQL, une requête peut elle-même être composée avec une autre requête. On parle alors de sous-requêtes. Il existe plusieurs types de sous-requêtes :

- les sous-requêtes qui retournent une relation et qui sont utilisées dans une clause **FROM**. Dans ce cas, il faut leur donner un nom d'alias pour pouvoir les utiliser ;
- les sous-requêtes qui retournent une constante. Dans ce cas, on peut les utiliser dans la clause **WHERE** d'une requête. Par exemple :

```
SELECT prix
FROM Ordinateur
WHERE ref =
  (SELECT ref
   FROM Installation
   WHERE nom_sys = 'UNIX');
```

- les sous-requêtes qui retournent une relation et qui sont utilisées dans une clause **WHERE**.

Dans ce dernier cas, on peut utiliser différents opérateurs, qui s'appliquent à une relation R et renvoient un booléen :

- **EXISTS** R qui est vraie si et seulement si R n'est pas vide ;
- s **IN** R est vraie si et seulement si s apparaît dans R (s peut être un tuple sous la forme $(A1, A2)$ par exemple). Le contraire de s **IN** R est s **NOT IN** R ;
- s **OP ALL** R où **OP** est un opérateur de comparaison est vraie si et seulement si s **OP** t est vraie pour tout tuple t de R ;
- s **OP ANY** R où **OP** est un opérateur de comparaison est vraie si et seulement si s **OP** t est vraie pour au moins un tuple t de R .

Exemple 8.2.10 : Considérons la requête « *quelles sont les provenances des systèmes installables sur un Micral 75 ?* ». Elle s'écrit :

```
SELECT provenance
FROM Systeme
WHERE nom_sys IN
  (SELECT nom_sys
   FROM LogicielBase
   WHERE type = 'Micral 75')
```

□

Exemple 8.2.11 : Considérons la requête « *quels sont les noms de logiciels qui ne sont installés sur aucune machine ?* ». Elle s'écrit :

```
SELECT nom, classe
FROM Logiciel
WHERE NOT EXISTS
  (SELECT nom_log
   FROM Installation
   WHERE nom = nom_log);
```

□

8.2.14. Insertion

SQL propose deux formes d'insertion de données :

— une insertion de valeurs fournies en entrée (par « extension ») :

Syntaxe (insertion en extension) :

```
INSERT
INTO R(Ai, Aj, ..., Ap)
VALUES (vi, vj, ..., vp);
```

□

v_k est une constante ou une expression calculée. La valeur v_k est celle de l'attribut A_k de la ligne en cours de remplissage. Si les A_k sont omis, les valeurs sont affectées dans l'ordre de déclaration des colonnes.

Si un attribut n'apparaît pas dans la liste, sa valeur est la valeur par défaut de son type (**NULL** la plupart du temps).

ON peut insérer plusieurs tuples avec la même instruction (attention, dans ce cas l'insertion de tous les tuples est considérée comme une transaction, cf. chapitre 12) :

Syntaxe (insertion en extension de plusieurs tuples) :

```
INSERT
INTO R(Ai, Aj, ..., Ap)
VALUES (vi1, vj1, ..., vp1),
       (vi2, vj2, ..., vp2),
       ...
       (vin, vjn, ..., vpn);
```

□

— une insertion de valeurs issues d'une autre table (par « intension ») :

Syntaxe (insertion en intension) :

8. SQL (Structured Query Language)

```
INSERT  
INTO R(Ai, Aj, ..., Ap)  
SELECT Ci, Cj, ..., Cp  
FROM ...;
```

□

8.2.15. Modification de tuples

La modification de tuples d'une table se fait par la construction **UPDATE**.

Syntaxe (mise à jour) :

```
UPDATE R  
SET Ai = vi, ..., Ap = vp  
WHERE F;
```

□

F est l'assertion qualifiant les tuples à modifier.

Exemple 8.2.12 : L'instruction

```
UPDATE Ordinateur  
SET capa_mem = capa_mem *2  
WHERE type = 'Micral 75';
```

permet de doubler la capacité mémoire de tous les ordinateurs Micral 75.

□

8.2.16. Suppression

La suppression de tuples d'une table se fait par la construction **DELETE**.

Syntaxe (suppression de tuples) :

```
DELETE  
FROM R  
WHERE F;
```

□

Cette requête supprime tous les n-uplets de R satisfaisant F .

Exemple 8.2.13 : La requête

```
DELETE  
FROM Ordinateur  
WHERE type = 'Goupil G4';
```

supprime tous les ordinateurs Goupil G4.

□

8.3. Le langage de description de données

Le langage de description de données de SQL permet de créer et d'administrer :

- les relations d'un schéma conceptuel;
- les vues externes construites sur ce schéma;
- une partie du niveau interne (c'est-à-dire le niveau physique) en définissant les index.

8.3.1. Types de données

Dans un premier temps, nous allons introduire les types de données manipulables par SQL. En effet, tous les attributs d'un schéma doivent avoir un type.

1. les chaînes de caractères peuvent être de longueur fixe ou de longueur variable. **CHAR**(n) est une chaîne de longueur n, alors que **VARCHAR**(n) est une chaîne de longueur maximale n. Par exemple, si 'bla' est affectée à un attribut de type **CHAR**(5), la valeur de l'attribut sera 'bla'. Si la même chaîne est affectée à un attribut de type **VARCHAR**(5), la valeur de l'attribut sera 'bla';
2. des chaînes d'octets représentées par les types **BIT**(n) et **BIT VARYING**(n);
3. le type **BOOLEAN**;
4. le type **INT** ou **INTEGER**;
5. les types **FLOAT** et **REAL** (ils sont synonymes). Un type réel plus précis est **DOUBLE PRECISION**. On peut fixer le format d'un réel avec **DECIMAL**(n,d) où n est le nombre de chiffres du réel et d le nombre de chiffres après la virgule de ce réel;
6. enfin, il existe des types **DATE**, **TIME** et **TIMESTAMP** qui permet de combiner les deux premiers types. Le 21 avril 1976 peut être représenté par '1976-04-21', '04/21/1976'².

Suivant le SGBD installé, de nombreux autres types sont disponibles : types géométriques, tableaux, etc.

8.3.2. Création de table

La création de tables se fait grâce à l'opérateur **CREATE TABLE**.

Syntaxe (création de table) :

```
CREATE TABLE Nom (
  ATT1 Type1,
  ATT2 Type2,
  ...
);
```

□

2. La représentation peut également être de la forme '21/04/1976'. Cela dépend du SGBD installé

8. SQL (*Structured Query Language*)

Exemple 8.3.1 : La création de la table `Configuration` se fait de la façon suivante :

```
CREATE TABLE Configuration (  
    nom          VARCHAR(20),  
    nom_sys     VARCHAR(20),  
    type        VARCHAR(20),  
    mem_min     INTEGER DEFAULT 4,  
    disk_min    INTEGER DEFAULT 1  
);
```

□

Le mot-clé **DEFAULT** permet de donner une valeur par défaut à l'attribut. Si aucune valeur par défaut n'est précisée à la création de la table, **NULL** est utilisé.

Nous verrons dans le chapitre 9 comment écrire des contraintes sur les tables à la création.

8.3.3. Destruction d'une relation

Syntaxe (suppression d'une table) :

```
DROP TABLE R;
```

□

Cette requête supprime non seulement le contenu de la relation, mais également son schéma.

8.3.4. Modification du schéma d'une relation

On peut modifier le schéma d'une relation de deux façons :

- ajouter un attribut à la relation en utilisant le mot clé **ADD**. Dans ce cas, l'attribut est également ajouté aux tuples existants et la valeur de l'attribut pour ces tuples est la valeur par défaut ;
- supprimer un attribut avec le mot-clé **DROP**.

Syntaxe (modification du schéma) :

```
ALTER TABLE R  
ADD ATT1 Type1;
```

```
ALTER TABLE R  
DROP ATT2;
```

□

La suppression de colonnes dans une relation peut entraîner de gros problèmes, en particulier au niveau des clés étrangères (cf. chapitre 9). De plus, le bon fonctionnement des applications externes utilisant la relation peut être perturbé par ces changements. On s'abstiendra donc le plus possible de modifier le schéma d'une relation.

8.3.5. Index

Nous présentons rapidement les index dans cette section. Leur utilisation permet souvent d'améliorer les performances lors de requêtes sur une base de données.

Définition 8.3.1.

Un index sur un attribut A d'une relation R est une structure de données efficace pour trouver les tuples qui ont une valeur fixée pour A . □

Lorsque les relations ont un nombre très important de tuples, il devient très coûteux lors d'une requête de vérifier que tous les tuples de la relation vérifient une propriété. Par exemple, supposons que l'on veuille les types des ordinateurs ayant un prix de 10 000 F, et que la relation `Ordinateur` possède 10 000 tuples. Dans ce cas, on va devoir faire 10 000 vérifications alors qu'il ne risque d'avoir que quelques tuples vérifiant la condition. L'utilisation d'un index sur l'attribut `prix` permet de trouver tout de suite les tuples cherchés³.

Syntaxe (création d'un index) :

```
CREATE INDEX NomIndex ON R(ATT1, ..., ATTn);
```

□

Remarquons que l'on peut créer un index sur plusieurs attributs. L'ordre des attributs est alors important pour l'efficacité de l'optimisation. Il faut mettre en premier les attributs susceptibles d'être les plus utilisés.

Syntaxe (destruction d'un index) :

```
DROP INDEX NomIndex;
```

□

Remarque : les index doivent être eux-mêmes stockés sur le disque, donc consomment de la place. De plus, ils rendent plus difficiles les opérations de mise à jour. Il faut donc réfléchir à l'usage « courant » de la base qui va être fait (beaucoup de modifications, beaucoup de requêtes) avant de créer un index sur certains attributs d'une relation. □

8.3.6. Vues externes

Une vue constitue une restriction d'une base de données à un sous-ensemble de cette base de données. Elle permet de ne rendre visible qu'une partie de cette base.

Définition 8.3.2.

On appelle vue externe une relation virtuelle qui :

- est le résultat d'une requête ;
- ne contient aucun tuple ;

3. Cette opération d'optimisation est effectuée par processeur de requêtes SQL qui ne va sélectionner que les tuples cherchés grâce à l'index, sans parcourir l'ensemble des tuples de `Ordinateur`.

8. SQL (*Structured Query Language*)

— est utilisable en consultation comme n'importe quelle autre relation de la base de données. □

Les vues ne sont donc pas stockées physiquement.

Création d'une vue

Syntaxe (création d'une vue) :

```
CREATE VIEW Nom (ATT1, ATT2, ..., ATTn)
AS
(SELECT C1, ..., Cn
 FROM R
 ...
 );
```

□

L'attribut ATT_i de la vue représente la colonne C_i de même rang de la requête qui calcule la vue.

Exemple 8.3.2 : Considérons la requête qui permet de construire la vue **Catalogue**. Elle contient le nom, la classe, le nombre d'installations et le prix des logiciels disponibles. Elle s'écrit :

```
CREATE VIEW Catalogue (nom, nombre prix)
AS
(SELECT L.nom, count(I.reference), L.prix
 FROM Logiciel L, Installation I
 WHERE L.nom = I.nom_log
 GROUP BY L.nom);
```

□

Remarque : On a jusqu'à présent utilisé trois termes différents : « tables », « relations » et « vues ». Les programmeurs SQL utilisent le terme « table » au lieu de « relation » pour parler des relations qui sont effectivement stockées (on pourrait même supposer qu'une relation est implantée par plusieurs tables). Une vue est donc une relation qui elle n'est pas stockée physiquement.

En algèbre relationnelle, on sait que toute opération a pour résultat une relation. Une vue est donc simplement une expression de l'algèbre relationnelle à laquelle on a donné un nom. □

Destruction d'une vue

Syntaxe (destruction d'une vue) :

```
DROP VIEW Nom [RESTRICT|CASCADE];
```

□

La destruction d'une vue n'affecte en rien les tables à partir desquelles elle était construite.

L'option **RESTRICT** (par défaut) permet de préciser que si l'on fait référence à cette vue dans une autre vue ou dans une contrainte d'intégrité, la suppression va échouer. Au contraire, l'option **CASCADE** va permettre de détruire *automatiquement* les vues et les contraintes d'intégrité faisant référence à la vue.

Modifications de vues

On peut exécuter des modifications sur les tuples d'une table au travers d'une vue en respectant certaines conditions.

Hypothèse 8.3.1.

Une vue peut être mise à jour si et seulement si :

- la requête **SELECT** ne porte que sur une seule relation R et ne possède pas d'agrégats ni de regroupements ;
- la clause **WHERE** ne contient pas R dans une sous-requête ;
- la liste dans la clause **SELECT** (sans **DISTINCT**) doit contenir suffisamment d'attributs pour pouvoir remplir les autres attributs de la relation avec leur valeur par défaut. □

Un exemple très simple permet de comprendre pourquoi on ne peut pas mettre à jour une vue impliquant plusieurs relations. Prenons par exemple deux relations R et S . Supposons que le critère de la clause **WHERE** porte sur une égalité entre l'attribut a_1 de R et l'attribut b_1 de S . Ces deux attributs n'apparaissent pas dans la liste de colonnes de la clause **SELECT**. À l'insertion d'un tuple via la vue, on va donc créer un tuple de R dont la valeur de l'attribut a_1 sera **NULL** et un tuple de S dont la valeur de l'attribut b_1 sera **NULL**. Or un test **NULL = NULL** n'est pas vrai, donc les deux nouveaux tuples n'apparaîtront pas dans le calcul de la vue.

On se demande comment sont gérées les insertions et les suppressions de tuples dans les relations concernées par la vue (les mises à jour n'étant qu'une séquence suppression/ajout). Par exemple, la suppression d'un tuple d'une vue construite sur **R1 INTERSECT R2** devra supprimer le tuple en question de **R1** et de **R2**, même si une seule suppression, par exemple dans **R1**, permet de supprimer le tuple de la vue.

Le tableau 8.1 présente les règles appliquées pour une vue construite à partir de deux relations en utilisant différents opérateurs (la généralisation à plusieurs relations est évidente). On suppose que les conditions présentées dans l'hypothèse précédente sont vérifiées. Dans certains cas, il se peut que les relations concernées soit en fait construites par une requête restrictive (par exemple (**SELECT * FROM R1 WHERE P1**) **UNION ...**). Dans ce cas, la restriction sera représentée par un prédicat P_i représentant la condition de restriction.

8. SQL (*Structured Query Language*)

Utilisation des vues externes

Les vues sont utilisées pour la consultation d'informations. Plus précisément, elles sont définies pour :

- donne une vision plus lisible d'un schéma, adaptée à un utilisateur ;
- permettre à un utilisateur d'être protégé contre les éventuels changements des relations de la base de données (si l'on change une relation on peut toujours construire une vue qui correspond à ce que l'utilisateur avait auparavant) ;
- fixer les accès à un schéma par la pose de droits d'accès (cf. section 8.4) ;
- simplifier une requête complexe en compilant dans une vue une partie de la requête (réutilisation).

8.4. Le langage de contrôle des données

À l'instar des systèmes de gestion de fichiers présents dans les systèmes d'exploitation, les SGBDs fondés sur SQL proposent un mécanisme de gestion de droits d'accès. Il est plus élaboré que le simple gestionnaire associé aux fichiers.

8.4.1. Utilisateurs d'une base de données

Les utilisateurs d'une base de données sont répertoriés par le SGBD avec :

- un identifiant interne ;
- un mot de passe ;
- un nom d'usage destiné aux autres utilisateurs.

8.4.2. Privilèges

SQL définit neuf types de privilèges : **SELECT**, **INSERT**, **DELETE**, **UPDATE**, **REFERENCES**, **USAGE**, **TRIGGER**, **EXECUTE** et **UNDER**. Nous nous intéresserons plus particulièrement aux quatre premiers⁴.

Comme leur nom l'indique, les privilèges **SELECT**, **INSERT**, **DELETE**, **UPDATE** permettent respectivement de poser une requête sur une relation, d'insérer un tuple dans une relation, d'effacer un tuple d'une relation et de mettre à jour une relation.

8.4.3. Création de privilèges

Dans un premier temps, toute table SQL a un propriétaire. Ce propriétaire a tous les droits sur cette table. Dans un second temps, ce propriétaire peut donner certains privilèges à d'autres utilisateurs sur cette table. Pour cela, on utilise la construction **GRANT**. On peut préciser alors si l'utilisateur auquel on accorde un privilège possède lui aussi le droit d'accorder ce privilège à d'autres utilisateurs.

Syntaxe (création de privilèges) :

4. Les mot-clés **TRIGGER** et **REFERENCES** seront abordés dans le chapitre 9

```

GRANT Privileges
ON Relation
TO liste utilisateurs
[WITH GRANT OPTION]

```

□

On peut remplacer les privilèges par **ALL** qui est un raccourci pour tous les privilèges. **PUBLIC** comme nom d'utilisateur représente l'ensemble des utilisateurs répertoriés dans la base de données.

Exemple 8.4.1 : Considérons une base de données avec quatre utilisateurs de noms d'usage **yamine**, **bernard**, **serge** et **christophe**. **yamine** est le propriétaire des relations **Ordinateur** et **Logiciel**. Il donne les privilèges **SELECT** et **INSERT** à **bernard** et **serge** sur **Ordinateur** et **SELECT** à **bernard** et **serge** sur **Logiciel** :

```

GRANT SELECT, INSERT ON Ordinateur TO bernard, serge
WITH GRANT OPTION;
GRANT SELECT ON Logiciel TO bernard, serge
WITH GRANT OPTION;

```

bernard donne à **christophe** les mêmes privilèges, mais sans l'option **GRANT OPTION** :

```

GRANT SELECT, INSERT ON Ordinateur TO christophe;
GRANT SELECT ON Logiciel TO christophe;

```

serge autorise quant à lui **christophe** à insérer un tuple dans **Logiciel** en n'utilisant que l'attribut **ref**, et à utiliser **SELECT** sur les deux relations, tout cela sans **GRANT OPTION** :

```

GRANT SELECT, INSERT(ref) ON Ordinateur TO christophe
GRANT SELECT ON Logiciel TO christophe

```

christophe a reçu ses privilèges **SELECT** sur les deux relations par deux utilisateurs différents. Il a également reçu le privilège **INSERT(ref)** deux fois : une fois indirectement par **bernard** qui lui donne un privilège **INSERT** sur toute la relation **Ordinateur** et une fois via le privilège **INSERT** donné par **serge**.

On peut résumer ce réseau de privilège via la figure 8.1. Une * représente le fait qu'un utilisateur a une **GRANT OPTION** sur le privilège en question. ** indique que le privilège dérive du fait que l'utilisateur est le propriétaire de la relation. Les traits en pointillés indique une délégation sans **GRANT OPTION**.

8.4.4. Retrait de privilèges

Pour retirer des privilèges à un ou plusieurs utilisateurs, on utilise le mot-clé **REVOKE**.

Syntaxe (retrait de privilèges) :

8. SQL (Structured Query Language)

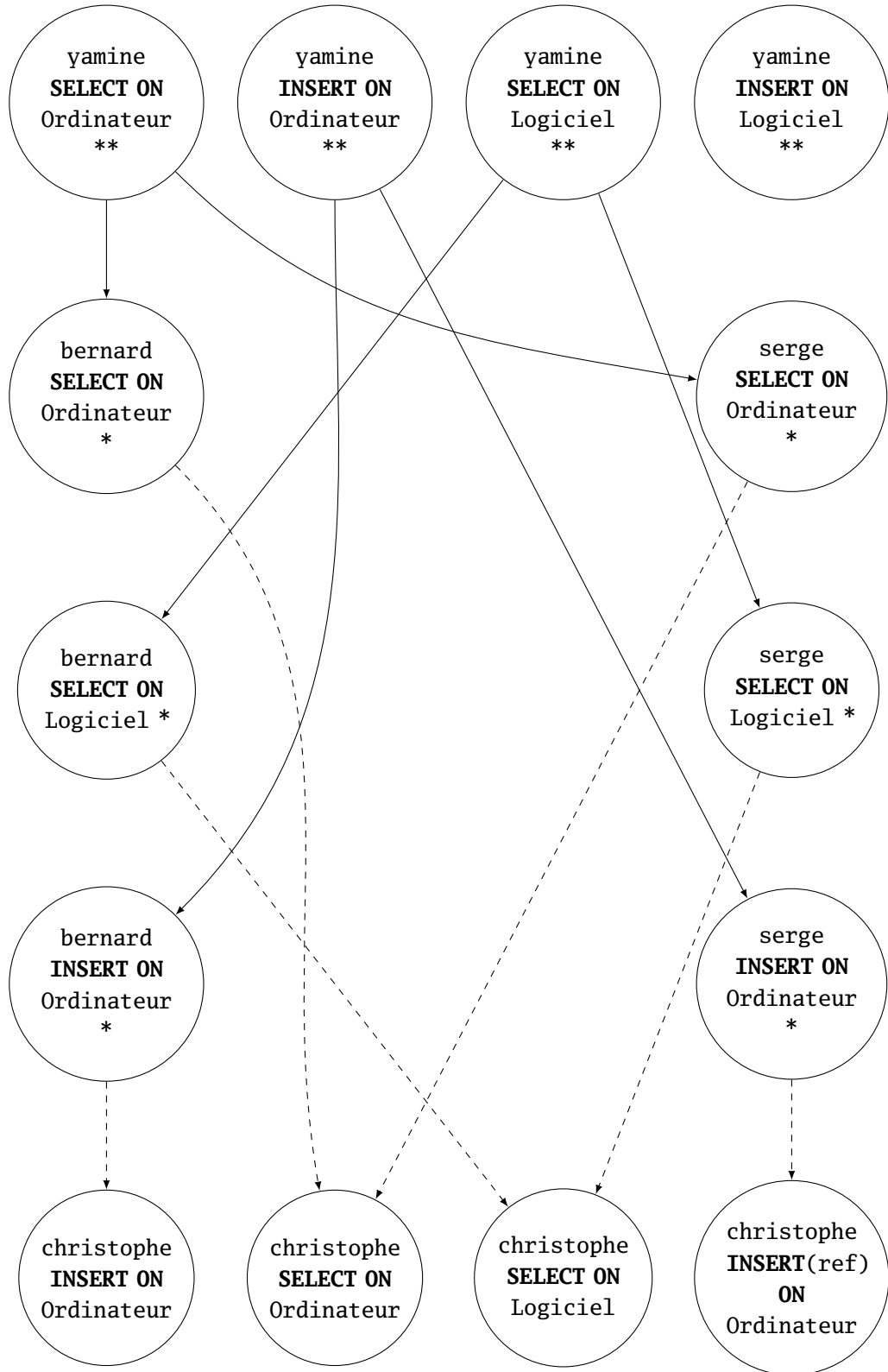


FIGURE 8.1. – Diagramme représentant les délégations de privilèges

```

REVOKE Privileges
ON Relation
TO liste utilisateurs
[CASCADE|RESTRICT]

```

□

On doit ajouter un des mots-clés suivants :

- **CASCADE**, et dans ce cas on retire également les privilèges cités qui n'avaient été fournis que par les utilisateurs listés dans **TO** à d'autres utilisateurs ;
- **RESTRICT** qui permet d'empêcher le retrait d'un privilège si ce privilège avait été donné à d'autres utilisateurs.

On peut également utiliser **REVOKE GRANT OPTION FOR** pour retirer le pouvoir d'octroyer un privilège.

Exemple 8.4.2 : Reprenons l'exemple 8.4.1. Supposons que **yamine** retire les privilèges de **serge** de la façon suivante :

```

REVOKE SELECT, INSERT ON Ordinateur TO serge
REVOKE SELECT ON Logiciel TO serge;

```

Dans ce cas, le diagramme de privilèges est représenté sur la figure 8.2.

8.4.5. Utilisation des vues

Supposons que l'on veuille imposer une restriction d'accès sur la relation **Ordinateur**, mais seulement sur les tuples concernant les ordinateurs de type **Mac II**. L'utilisation simple de **GRANT** ou **REVOKE** sur **Ordinateur** concernera l'ensemble des tuples de la relation.

Il suffit ici de construire une vue sur **Ordinateur** ne concernant que les tuples considérés et d'appliquer le **GRANT** ou le **REVOKE** sur cette vue (les utilisateurs n'auront donc pas accès à la relation **Ordinateur** dans son ensemble!) :

```

CREATE VIEW VueOrdinateur
AS (SELECT * FROM Ordinateur WHERE (type = 'Mac II'));

GRANT ...
ON VueOrdinateur
...

```

Les vues sont donc également un élément important des mécanismes de sécurité du langage SQL.

8. SQL (Structured Query Language)

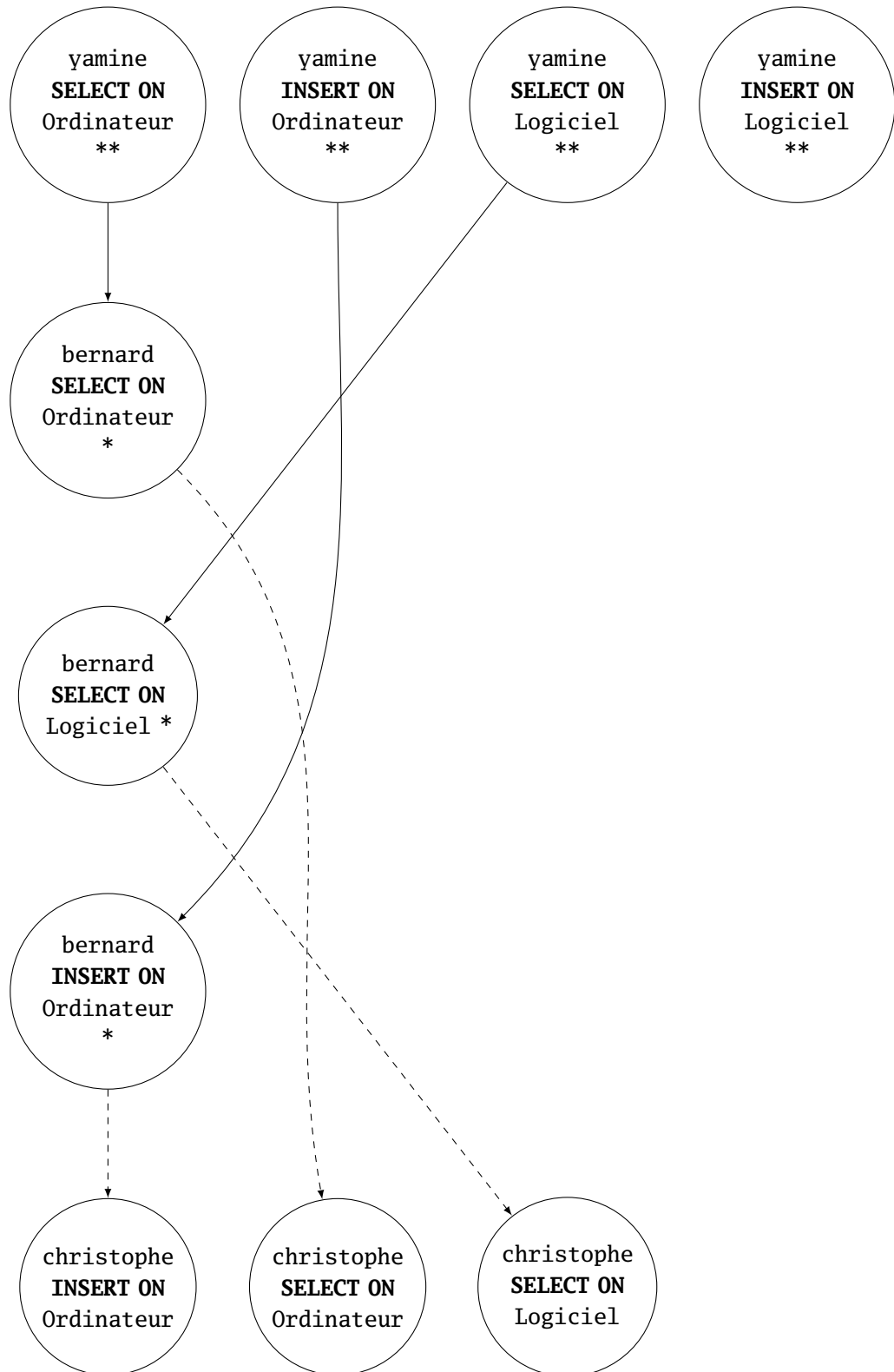


FIGURE 8.2. – Diagramme de privilèges après **REVOKE**

Relation(s)	Ajout d'un tuple t	Suppression d'un tuple t	Conditions
R1 UNION R2	si t respecte P1 il est inséré dans R1 si t respecte P2 il est inséré dans R2	si t respecte P1 il est supprimé dans R1 si t respecte P2 il est supprimé dans R2	t respecte P1 ou P2
R1 INTERSECT R2	si t respecte P1 il est inséré dans R1 si t respecte P2 il est inséré dans R2	si t respecte P1 il est supprimé de R1 si t respecte P2 il est supprimé de R2	t respecte P1 et P2
R1 MINUS R2	t est inséré dans R1	t est supprimé de R1	t respecte P1 et pas P2
SELECT * FROM R1 WHERE cond	t est inséré dans R1	t est supprimé de R1	t vérifie P1 et cond
SELECT X FROM R1	si $Y = R1 - X$ et YDEF est la val. par défaut de Y alors (t, Y) est inséré dans R1	tous les tuples de R1 ayant les mêmes valeurs en X que t sont effacés	t vérifie P1
R1 JOIN R2	t [R1] est inséré dans R1 et t [R2] est inséré dans R2	t [R1] est effacé de R1 et t [R2] est effacé de R2	t [R1] vérifie P1 et t [R2] vérifie P2

TABLE 8.1. – Conséquence des modifications d'une vue sur les relations sous-jacentes

9. Contraintes et triggers

Les informations qui sont stockées dans une base de données doivent la plupart du temps respecter des *contraintes d'intégrité*. Par exemple, le prix d'un ordinateur doit toujours être positif. Ces contraintes peuvent être plus compliquées : un attribut d'une relation doit apparaître également dans une autre relation, un certain nombre d'assertions doivent être vérifiées lors d'une mise à jour d'une relation etc.

Dans ce chapitre, nous allons étudier différents types de contraintes d'intégrité : les contraintes de clé tout d'abord, puis les assertions en enfin les *triggers*.

Les possibilités offertes par les contraintes et les *triggers* ne sont parfois pas complètement implantées dans les SGBD. Vous pouvez vous référer à l'annexe [A](#) pour plus de détails sur le SGBD que nous allons utiliser, PostgreSQL.

9.1. Clés primaires et clés étrangères

Une des contraintes les plus importantes dans une base de données est la déclaration de clés. Rappelons qu'une clé d'une relation R est un ensemble d'attributs S tel que deux tuples différents de R doivent au moins ne pas s'accorder sur la valeur d'un des attributs contenu dans S . La clé permet d'identifier de façon unique un tuple d'une relation.

SQL propose plusieurs types de clés : clés primaires, clés uniques et clés étrangères. Ces différentes contraintes sur des clés se déclarent toutes à l'intérieur de la clause **CREATE TABLE**.

9.1.1. Clés primaires

Une relation ne peut avoir qu'une seule clé primaire. On peut définir une clé primaire de deux façons différentes.

Si la clé primaire n'est composée que d'un seul attribut, on peut la définir de la façon suivante :

Syntaxe (clé primaire sur un attribut) :

```
CREATE TABLE Table (  
  ATT1 Type1,  
  ...  
  ATTi Typei PRIMARY KEY,  
  ...  
);
```

□

9. Contraintes et triggers

Si la clé primaire est composée d'un ensemble d'attributs, on sépare la clause **PRIMARY KEY** dans **CREATE TABLE**.

Syntaxe (clé primaire) :

```
CREATE TABLE Table (  
    ATT1  Type1,  
    ...  
    ATIn  Typen,  
    PRIMARY KEY (ATTi, ..., ATTj)  
);
```

□

Cette syntaxe est à privilégier : on sépare bien les contraintes de la déclaration des attributs.

Lorsque l'on déclare un ensemble d'attributs S comme clé primaire de la relation R , deux contraintes s'imposent :

- aucun attribut de S ne peut avoir la valeur **NULL** ;
- deux tuples de R ne peuvent avoir les mêmes valeurs pour tous les attributs de S .

Dans la plupart des SGBD, les contraintes de clés primaires induisent la construction automatique d'un index sur les attributs de la clé.

Exemple 9.1.1 : Reprenons l'exemple 8.3.1. La relation *Configuration* possède une clé primaire, qui est composée des attributs *nom*, *nom_sys* et *type*. On le déclare donc de la façon suivante :

```
CREATE TABLE Configuration (  
    nom      VARCHAR(20),  
    nom_sys  VARCHAR(20),  
    type     VARCHAR(20),  
    mem_min  INTEGER DEFAULT 4,  
    disk_min INTEGER DEFAULT 1,  
    PRIMARY KEY (nom, nom_sys, type)  
);
```

□

9.1.2. Contrainte UNIQUE

La contrainte **UNIQUE** permet de déclarer un attribut comme étant unique. Sa syntaxe est exactement la même que la contrainte **PRIMARY KEY**. Cependant, deux distinctions sont à faire :

- on peut avoir plusieurs déclarations **UNIQUE** pour une table, mais une seule déclaration **PRIMARY KEY** ;
- un élément **UNIQUE** peut avoir la valeur **NULL**.

9.1.3. Clés étrangères

Dans une relation, certains attributs ne sont pas « indépendants ». Ils doivent en effet correspondre à des attributs de tuples existants d'une autre relation. Par exemple, dans la relation *Configuration*, l'attribut *nom* doit correspondre effectivement à une valeur de l'attribut *nom* de la relation *Logiciel* présente dans un tuple de *Logiciel*. On parle alors de clé étrangère.

Pour qu'un attribut d'une relation R puisse apparaître comme clé étrangère d'un attribut d'une relation T , certaines hypothèses sont à respecter.

Hypothèse 9.1.1.

Pour que l'attribut a_R de la relation R puisse apparaître comme clé étrangère de l'attribut a_T de la relation T , il faut et il suffit que :

- a_R soit déclaré **UNIQUE** ou apparaissent dans la clé primaire de R ;
- les valeurs de a_T apparaissant dans des tuples de T doivent apparaître dans des tuples de R . □

Comme pour **PRIMARY KEY**, il y a deux façons de déclarer une clé étrangère.

Syntaxe (clé étrangère) :

- soit l'attribut est unique et dans ce cas :

```
CREATE TABLE Table1 (
    ATT1 Type1,
    ...
    ATTi Typei REFERENCES Table2(ATT2j),
    ...
);
```

- soit l'attribut n'est pas unique :

```
CREATE TABLE T1 (
    ATT1 Type1,
    ...
    ATTn Typen,
    FOREIGN KEY (ATTi,...,ATTj) REFERENCES T2(ATT2l,..., ATT2k),
    ...
);
```

□

Exemple 9.1.2 : Reprenons la table *Configuration* de l'exemple 9.1.1. On peut introduire des contraintes de clés étrangères de la façon suivante :

```
CREATE TABLE Configuration (
    nom          VARCHAR(20),
    nom_sys     VARCHAR(20),
    type        VARCHAR(20),
```


9. Contraintes et triggers

```
mem_min  INTEGER DEFAULT 4,  
disk_min INTEGER DEFAULT 1,  
PRIMARY KEY (nom, nom_sys, type),  
FOREIGN KEY (nom) REFERENCES Logiciel(nom),  
FOREIGN KEY (nom_sys) REFERENCES Systeme(nom_sys),  
FOREIGN KEY (type) REFERENCES TypeOrdinateur(type)  
);
```

□

9.1.4. Politique de gestion des clés étrangères

Nous avons vu comment créer des clés étrangères. Que se passe-t-il maintenant si on met à jour une table contenant une clé étrangère et que l'on donne une valeur non permise à cette clé? Nous allons voir que trois politiques différentes sont disponibles.

Politique par défaut : rejet

La politique par défaut est de ne pas autoriser de modifications illicites. Par exemple, en reprenant l'exemple 9.1.2, les modifications suivantes seront rejetées :

1. ajout d'un tuple de `Configuration` dont l'attribut `nom` est `NULL` ou n'apparaît pas dans un tuple de `Logiciel`;
2. modification d'un tuple de `Configuration` dont l'attribut `nom` devient `NULL` ou n'apparaît pas dans un tuple de `Logiciel`;
3. effacement d'un tuple de `Logiciel` dont l'attribut `nom` apparaît dans un tuple de `Configuration`;
4. modification d'un tuple de `Logiciel` dont l'attribut `nom` est modifié et apparaît dans un tuple de `Configuration`.

Politique en cascade

Une autre politique de gestion des clés étrangères est de répercuter tous les changements nécessaires pour garder l'intégrité de la base. C'est une politique par *cascade*. Par exemple, la modification d'un tuple de `Logiciel` dont l'attribut `nom` est modifié et apparaît dans un tuple de `Configuration` sera répercutée sur le tuple correspondant de `Configuration`. Cette politique ne concerne que les cas 3 et 4 présentés ci-dessus.

Politique NULL

Une autre politique est de mettre les valeurs des champs de `Configuration` à `NULL`. Cette politique est appelée *set-NULL*.

Dans ces deux cas, la syntaxe suivante est utilisée :

Syntaxe (politique de gestion des clés étrangères) :

```

CREATE TABLE T1 (
  ...
  FOREIGN KEY ATT1 REFERENCES T2(...)
    ON DELETE [SET NULL|CASCADE]
    ON UPDATE [SET NULL|CASCADE]
);

```

□

Il se peut que l'on ait un système de dépendances circulaires entre clés étrangères. Dans ce cas, comment faire pour pouvoir mettre à jour un tuple d'une des relations en question ? Il y aura toujours une violation de contrainte de clé dans ce cas. Une solution est disponible :

- grouper les mises à jour dans une *transaction* (cf. chapitre 12) ;
- retarder la vérification de cohérence des contraintes de clés étrangères.

Toute contrainte peut être déclaré **DEFERRABLE** ou **NOT DEFERRABLE** (c'est cette dernière option qui est prise par défaut). Une contrainte **NOT DEFERRABLE** sera vérifiée à chaque changement de la base la concernant. Une contrainte **DEFERRABLE** ne sera vérifiée qu'à la fin de la transaction courante. Dans le cas d'une contrainte **DEFERRABLE**, on peut choisir de la rendre **INITIALLY DEFERRED** et elle ne sera vérifiée effectivement qu'à la fin de la transaction, ou **INITIALLY IMMEDIATE** et elle sera vérifiée avant modification, mais on garde l'option de retarder la vérification.

Exemple 9.1.3 : Reprenons la table *Configuration* de l'exemple 9.1.1. On peut introduire des « retards » de vérification de la façon suivante :

```

CREATE TABLE Configuration (
  nom          VARCHAR(20),
  nom_sys     VARCHAR(20),
  type        VARCHAR(20),
  mem_min     INTEGER DEFAULT 4,
  disk_min    INTEGER DEFAULT 1,
  PRIMARY KEY (nom, nom_sys, type),
  FOREIGN KEY (nom) REFERENCES Logiciel(nom)
  DEFERRABLE INITIALLY DEFERRED,
  FOREIGN KEY (nom_sys) REFERENCES Systeme(nom_sys)
  DEFERRABLE INITIALLY IMMEDIATE,
  FOREIGN KEY (type) REFERENCES TypeOrdinateur(type)
);

```

□

9.2. Contraintes sur les attributs et les tuples

Nous avons vu précédemment comment imposer des contraintes de clés sur une relation. Il existe d'autres types de contraintes que nous allons présenter à présent. Elles sont de deux types :

9. Contraintes et triggers

- des contraintes sur les attributs d'une relation ;
- des contraintes sur les tuples d'une relation.

9.2.1. Contrainte NOT NULL

La contrainte **NOT NULL** imposée à un attribut n'autorise pas cet attribut à apparaître avec la valeur **NULL** dans un tuple.

Syntaxe (contrainte NOT NULL) :

```
CREATE TABLE Table (  
    ...  
    ATT1 Type1 NOT NULL,  
    ...  
);
```

□

Remarque : un attribut déclaré **UNIQUE** et **NOT NULL** est une clé primaire. □

9.2.2. Contrainte CHECK sur un attribut

On peut attacher à un attribut des contraintes plus complexes que l'on va exprimer sous la forme d'une expression logique dans une clause **CHECK**. Cette expression logique a la forme d'une expression que l'on trouve dans une clause **WHERE**.

Syntaxe (contrainte CHECK sur un attribut) :

```
CREATE TABLE Table (  
    ...  
    ATTi Typei ...  
    CHECK F,  
    ...  
);
```

F est une expression logique dans laquelle *ATTi* apparaît. Si un autre attribut apparaît dans *F*, celui-ci doit provenir d'une clause **SELECT** incluse dans *F*. □

La contrainte exprimée par **CHECK** sera vérifiée à chaque fois que l'attribut concerné sera modifié par un ajout ou une mise à jour. Attention, elle ne sera pas vérifiée si on ne le modifie pas (ce qui peut être problématique si la contrainte concerne un attribut d'une autre relation qui est lui mis à jour).

Exemple 9.2.1 : Considérons la relation *Configuration*. Supposons que l'on dispose d'une relation *Mémoire* qui nous fournisse les différentes configurations mémoire valides via un attribut *mem*. Dans ce cas, on peut écrire :

```

CREATE TABLE Configuration(
  ...
  mem_min  NUMBER(4)
           CHECK (mem_min IN (SELECT mem FROM Memoire)),
  ...
);

```

□

Remarque : la base de données utilisée en BE, PostgreSQL, ne permet pas d'utiliser des requêtes dans une clause **CHECK**, ceci pour éviter d'avoir des contraintes qui ne seront pas vérifiées lors d'une modification d'une autre relation (cas d'une clause **SELECT** dans l'expression logique). On peut utiliser dans ce cas un *trigger* (cf. 9.3.2) □

9.2.3. Contrainte CHECK sur un tuple

On peut également créer des contraintes qui s'appliquent sur les tuples d'une relation.

Syntaxe (contrainte **CHECK** sur un tuple) :

```

CREATE TABLE Table (
  ...
  ATTi Typei ...,
  ...

  CHECK F
);

```

F est une expression logique dans laquelle un ou plusieurs attributs de **Table** apparaissent. Si un attribut d'une autre relation apparaît dans F , celui-ci doit provenir d'une clause **SELECT** incluse dans F . □

Les contraintes **CHECK** exprimées sur un tuple sont vérifiées à l'insertion d'un tuple de la relation ou à la mise à jour d'un tuple de la relation.

Exemple 9.2.2 : Reprenons la table *Configuration* :

```

CREATE TABLE Configuration (
  nom      VARCHAR(20),
  nom_sys  VARCHAR(20),
  type     VARCHAR(20),
  mem_min  INTEGER DEFAULT 4,
  disk_min INTEGER DEFAULT 1,
  PRIMARY KEY (nom, nom_sys, type),
  FOREIGN KEY (nom) REFERENCES Logiciel(nom)
  DEFERRABLE INITIALLY DEFERRED,
  FOREIGN KEY (nom_sys) REFERENCES Systeme(nom_sys)
  DEFERRABLE INITIALLY IMMEDIATE,
);

```

9. Contraintes et triggers

```
FOREIGN KEY (type) REFERENCES TypeOrdinateur(type),
CHECK ((mem_min >= 0) AND (disk_min >= 0))
);
```

□

9.2.4. Modification de contraintes

On peut ajouter, modifier ou effacer des contraintes à n'importe quel moment. SQL propose des primitives permettant de réaliser ces opérations.

Nommage d'une contrainte

Syntaxe (nommage d'une contrainte) :

```
CONSTRAINT NomContrainte contrainte
```

□

Cette syntaxe permet de donner un nom à une contrainte. Il devient plus lisible en cas de message d'erreur de repérer quelle contrainte est mise en cause. De plus, on pourra utiliser ce nom pour modifier la contrainte ou l'effacer.

Exemple 9.2.3 :

```
CREATE TABLE Configuration (
  nom          VARCHAR(20),
  nom_sys     VARCHAR(20),
  type        VARCHAR(20),
  mem_min     INTEGER DEFAULT 4,
  disk_min    INTEGER DEFAULT 1,
  CONSTRAINT CleConf PRIMARY KEY (nom, nom_sys, type),
  CONSTRAINT CleEt1Conf FOREIGN KEY (nom)
                      REFERENCES Logiciel(nom)
                      DEFERRABLE INITIALLY DEFERRED,
  CONSTRAINT CleEt2Conf FOREIGN KEY (nom_sys)
                      REFERENCES Systeme(nom_sys)
                      DEFERRABLE INITIALLY IMMEDIATE,
  CONSTRAINT CleEt2Conf FOREIGN KEY (type)
                      REFERENCES TypeOrdinateur(type),
  CONSTRAINT IntegriteConf CHECK ((mem_min >= 0) AND (disk_min >= 0))
);
```

□

Modification d'une contrainte

On peut tout d'abord modifier une contrainte **DEFERRABLE** en la rendant soit **DEFERRED**, soit **IMMEDIATE** :

Syntaxe (modification d'une contrainte) :

```
SET CONSTRAINT NomContrainte [DEFERRED|IMMEDIATE];
```

□

On peut également ajouter ou enlever une contrainte en utilisant la clause **ALTER TABLE** vue dans la section 8.3.4 :

Syntaxe (modification d'une contrainte) :

```
ALTER TABLE Table DROP CONSTRAINT NomContrainte;
ALTER TABLE Table ADD CONSTRAINT NomContrainte
    definition de la contrainte;
```

□

9.3. Assertions et triggers

Il se peut que les contraintes imposées via une clause **CHECK** sur un attribut ou sur un tuple ne soient pas suffisantes pour pouvoir garantir l'intégrité de la base. Par exemple, la clause **CHECK (mem_min IN (SELECT mem FROM Memoire))** ne sera pas vérifiée si l'on effectue une mise à jour sur l'attribut `mem` de `Memoire` et la contrainte peut être alors violée. SQL met à notre disposition des contraintes plus fortes :

- les assertions, qui sont des expressions logiques qui doivent être vérifiées à *tout moment* ;
- les *triggers* qui sont une série d'actions qui sont associés à un événement particulier.

Ces contraintes sont associés au schéma de la base de données. Les assertions sont très coûteuses pour le SGBD. Elles doivent donc être utilisées avec parcimonie.

9.3.1. Assertions

Une assertion est une contrainte générale qui se déclare comme suit.

Syntaxe (création d'assertion) :

```
CREATE ASSERTION NomAssertion CHECK (F);
```

F est une expression logique.

□

La condition exprimée par une assertion doit être vraie à la création de l'assertion et sera vérifiée à chaque modification de la base (cela peut donc être très coûteux, surtout si la modification ne « concerne » par l'assertion). Notons également que tous les attributs qui apparaissent dans F doivent être sélectionnés au moyen d'une clause **SELECT**.

Remarque : Le SGBD PostgreSQL n'implante pas encore les assertions.

□

9.3.2. Triggers

Les *triggers* sont des règles d'actions qui sont déclenchées par un événement particulier, le plus souvent l'insertion, la mise à jour ou l'effacement d'un tuple. Lors d'un événement, les *triggers* correspondants sont réveillés. Si une condition associée à un *trigger* est vérifiée, celui-ci exécutera un certain nombre d'actions. Cette approche est très souple : on peut spécifier finement un comportement pour tout événement qui pourrait être problématique (mise à jour, effacement etc.).

Syntaxe (*trigger*) :

```
CREATE TRIGGER NomTrigger
[AFTER|BEFORE] [UPDATE [OF att]|INSERT|DELETE] ON Table
REFERENCING
  [OLD ROW AS NomAncienTuple],
  [NEW ROW AS NomNouveauTuple],
  [OLD TABLE AS NomAncienneTable],
  [NEW TABLE AS NomNouvelleTable]
[FOR EACH STATEMENT|FOR EACH ROW]
WHEN (C)
  BEGIN [ATOMIC]
    Action1;
    ...
    ActionP;
  END;
```

□

Détaillons la syntaxe de cette clause :

- on crée un *trigger* de nom **NomTrigger** ;
- on choisit le moment d'exécution des actions précisées par le *trigger*. En utilisant **AFTER**, la condition **C** sera testée après l'exécution de l'événement déclencheur, en utilisant **BEFORE**, la condition **C** sera testée avant l'exécution de l'événement déclencheur. Si **C** est vérifiée, les actions **Action1**, ..., **ActionP** sont exécutées ;
- les événements déclencheurs possibles sont **UPDATE** (avec possibilité de préciser sur quel attribut), **INSERT** et **DELETE** ;
- les clauses **FOR EACH STATEMENT** (par défaut) et **FOR EACH ROW** permettent de préciser respectivement si les actions du *trigger* s'appliquent une seule fois ou à chaque tuple concerné ;
- les éléments de la clause **REFERENCING** permettent de disposer de noms de variables pour les anciens et nouveaux tuples et les anciennes et nouvelles tables. Dans le cas d'un *trigger* **FOR EACH STATEMENT**, on utilise **OLD TABLE** et **NEW TABLE**, sinon on utilise **OLD ROW** et **NEW ROW**. Suivi l'événement déclencheur, on a différentes possibilités :
 - dans le cas d'un **INSERT**, on ne peut utiliser que **NEW [ROW|TABLE]** ;
 - dans le cas d'un **DELETE**, on ne peut utiliser que **OLD [ROW|TABLE]** ;

- dans le cas d'un **UPDATE**, on peut utiliser les deux.
Ces références vont permettre par exemple de comparer les valeurs avant et après modifications.
- la clause **WHEN** précise la condition du *trigger*. Elle est facultative ;
- les actions du *trigger* sont encadrées par les mot-clés **BEGIN** et **END** (facultatifs dans le cas d'une seule action). **ATOMIC** permet de préciser si les actions doivent être considérées comme étant des transactions.

Exemple 9.3.1 : Supposons que l'on veuille que la moyenne des prix des logiciels (cf. relation *Logiciel*) soit toujours inférieure à 10000. Si l'on fait une mise à jour sur l'ensemble des tuples de la relation, il se peut que la moyenne dépasse 10000 après un certain nombre de modifications, puis redevienne inférieure à 10000. Il faut donc créer un *trigger* de type **FOR EACH STATEMENT** (une modification d'un tuple peut violer la contrainte, alors que le résultat de plusieurs modifications ne la violeront pas).

```

CREATE TRIGGER TriggerMoyenneLogiciel
AFTER UPDATE OF prix ON Logiciel
REFERENCING
  OLD TABLE AS AncienneTable,
  NEW TABLE AS NouvelleTable
FOR EACH STATEMENT
WHEN (10000 > (SELECT AVG(prix) FROM Logiciel))
BEGIN
  DELETE FROM Logiciel
  WHERE (nom, classe, concep, provenance, revendeur, prix)
    IN NouvelleTable;
  INSERT INTO Logiciel
    (SELECT * FROM AncienneTable);
END;

```

□

En ce qui concerne des concepts avancés comme les *triggers*, il vaut toujours mieux se référer au manuel d'utilisation du SGBD utilisé. Pour PostgreSQL, on pourra se référer à la section [A.1](#).

10. Intégration de SQL dans un langage de programmation

10.1. Introduction

Nous n'avons manipulé SQL jusqu'à présent qu'en utilisant des expressions déclaratives et le bloc **SELECT ... FROM ... WHERE**. Or on a souvent besoin en programmation de définir des procédures ou des fonctions complexes faisant appel par exemple à du contrôle de flot : conditionnelles, boucles etc.

Un standard récent de SQL, PSM (*Persistent Storage Modules*), offre la possibilité de stocker des procédures et des fonctions dans la base de données côté SGBD. PSM permet d'écrire des fonctions complexes et possède les habituelles structures de contrôle (boucle, branchement conditionnel etc.). Cela permet d'avoir une fonction ou une procédure qui sera partagée par tous les utilisateurs du SGBD sans duplication de code inutile, de grouper des traitements complexes sans avoir à faire des « allers-retours » entre le client et le serveur et d'avoir des fonctions que le compilateur de requêtes n'a pas besoin d'optimiser à chaque fois. PostgreSQL permet ainsi d'écrire des fonctions en utilisant PL/pgSQL, un langage impératif enrichissant la syntaxe de SQL (cf. chapitre A pour un exemple de création de fonction avec PL/pgSQL).

La plupart des utilisations de SQL se font à l'aide d'un langage de programmation « classique » qui permettent eux de développer d'autres aspects des applications : logique métier, interfaces graphiques etc. Des interfaces à SQL ont été définies pour COBOL, C, C++, Ada etc. Ce problème d'intégration de SQL n'est pas trivial : les structures de données manipulées par SQL (basiquement des bases de données relationnelles) ne sont pas définies directement dans les langages de programmation, qui eux utilisent des structures propres (par exemple des pointeurs en C). Deux problèmes se posent alors :

- le problème de l'*impedance mismatch* (par analogie avec l'électronique) : il se peut que les types de données, même les plus élémentaires, ne soient pas codés de la même façon entre le langage hôte et SQL. Il faut alors gérer cette différence, par exemple lorsque l'on récupère le résultat d'une requête.
- le langage hôte doit avoir une représentation (typée si possible) de ce qu'est une base de données et de ses composants : table, ligne, colonne, clé etc.

En ce qui concerne le deuxième point, une représentation très sommaire (il manque des choses !) d'un méta-modèle d'une base de données dans un langage objet (UML en l'occurrence) pourrait être celle présentée sur la figure 10.1¹.

1. On remarquera en particulier que la classe **Attribut** est paramétrée, car chaque attribut a un type particulier. Le type est également contraint pour n'autoriser que des types « compatibles » avec SQL

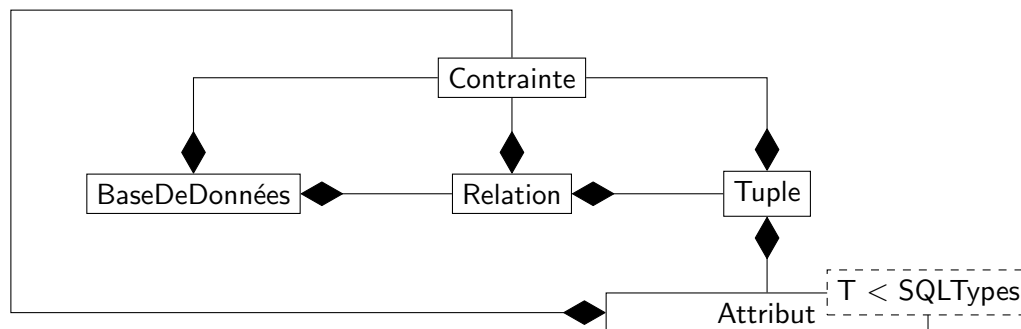


FIGURE 10.1. – Un méta-modèle objet sommaire d'une base de données

L'intégration de code SQL dans un langage hôte peut se faire de deux façons différentes suivant deux axes :

- *statiquement* ou *dynamiquement* : en SQL statique, les instructions SQL sont transformées à la compilation (par un compilateur ou un préprocesseur particulier) et exécutées lors de l'exécution du programme. En SQL dynamique, les instructions sont construites et exécutées lors de l'exécution du programme hôte.
- *intégré* ou *appelé* : en SQL intégré, les instructions SQL apparaissent dans le code source hôte comme des instructions « normales ». En SQL appelé, les instructions SQL sont encapsulées dans des appels à des fonctions/méthodes.

Évidemment, le SQL statique est souvent intégré et le SQL appelé est toujours dynamique. On remarquera enfin que du code SQL intégré statique est souvent transformé en code SQL appelé lors de la phase de compilation.

Nous avons choisi ici de présenter :

- l'interface entre SQL et C au travers d'appels statiques intégrés. Nous présenterons l'utilisation de l'utilitaire `ecpg` fourni par le SGBD PostgreSQL.
- l'interface entre SQL et Java au travers de JDBC (*Java DataBase Connectivity*). JDBC est une API (*Application Programming Interface*) Java d'accès aux SGBD. Elle est composée de plusieurs classes et interfaces dans le package `java.sql`. Ce type d'utilisation fournit un accès homogène aux SGBD et une abstraction des SGBD cibles. Il permet aussi d'écrire des requêtes SQL dynamiques et appelées depuis le langage Java.

10.2. SQL statique intégré avec C

Nous allons examiner l'intégration statique de SQL dans du code C. Nous allons donc écrire directement du code SQL. Un préprocesseur se chargera de transformer ce code SQL en appel à des fonctions d'une bibliothèque particulière qui permettent d'interagir à distance avec une base de données selon le modèle client/serveur (cf. section 10.3).

Le code SQL est embarqué dans le code C en utilisant la syntaxe suivante² :

2. Ne pas oublier le « ; » à la fin de la ligne.

Syntaxe (appel à du code SQL depuis C) :

```
EXEC SQL instructions ;
```

□

10.2.1. Établissement d'une connexion

Dans un premier temps, il faut ouvrir une connexion vers la base de données. Celle-ci est la plupart du temps repérée par une URL (*Uniform Resource Locator*) de la forme `nom_base@machine_serveur`. Il ne faut pas oublier ensuite de fermer la connexion à la base pour libérer les ressources.

Syntaxe (connexion et déconnexion à une base) :

```
EXEC SQL CONNECT TO bdbname@server USER username;
...
EXEC SQL DISCONNECT [DEFAULT|CURRENT|ALL];
```

□

On choisit de déconnecter la connexion par défaut, courante ou toutes les connexions en cours. On peut également donner un nom à une connexion pour ensuite l'utiliser lors de la déconnexion.

10.2.2. Production d'un exécutable

Prenons un exemple présenté sur le listing 10.1. Dans ce qui suit, nous utiliserons le SGBD PostgreSQL (cf. annexe A et l'utilitaire `ecpg` qui est fourni par PostgreSQL. Dans ce qui suit, je suppose que la variable d'environnement `$POSTGRES_HOME` contient le chemin vers le répertoire d'installation de PostgreSQL³. Le code source du listing 10.1 est enregistré dans le fichier `connexionC.pgc` (l'extension `pgc` est une convention pour l'utilisation de `ecpg`).

Listing 10.1 – Un exemple simple de connexion

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Trying to connect to database\n");
5     EXEC SQL CONNECT TO 'bdpolyIN306@serv-sun1.isae.fr' USER garion;
6     printf("Connected...\n");
7
8     printf("Trying to disconnect\n");
9     EXEC SQL DISCONNECT ALL;
10    printf("Disconnected...\n");
11
```

3. Cela peut changer suivant l'installation, voir la documentation locale de PostgreSQL.

```

12     return 0;
13 }

```

On remarque qu'il s'agit tout simplement d'un programme minimal écrit en C dans lequel on trouve des instructions SQL.

Dans un premier temps, nous allons utiliser l'utilitaire `ecpg` pour compiler notre code source en un code écrit complètement en C (% représente l'invite de commande du shell) :

```
% ecpg connexionC.pgc
```

On obtient alors un fichier `connexionC.c` qui est un code source purement écrit en C (cf. listing 10.2). On remarquera que ce code C fait appel à des fonctions d'une bibliothèque fournie par PostgreSQL, `libpg`, qui permet d'utiliser SQL de façon *dynamique* en C. Le compilateur `ecpg` transforme donc du code SQL statique en code SQL dynamique. Nous reviendrons sur ce point dans la conclusion.

Listing 10.2 – Le code C obtenu après compilation avec `ecpg`

```

1  /* Processed by ecpg (4.2.1) */
2  /* These include files are added by the preprocessor */
3  #include <ecpgtype.h>
4  #include <ecpglib.h>
5  #include <ecpgerrno.h>
6  #include <sqlca.h>
7  /* End of automatic include section */
8
9  #line 1 "connexionC.pgc"
10 #include <stdio.h>
11
12 int main() {
13     printf("Trying to connect to database\n");
14     { ECPGconnect(__LINE__, 0, "bdpolyIN306@serv-sun1.isae.fr" , "garion" , NULL , N
15 #line 5 "connexionC.pgc"
16
17     printf("Connected...\n");
18
19     printf("Trying to disconnect\n");
20     { ECPGdisconnect(__LINE__, "ALL");}
21 #line 9 "connexionC.pgc"
22
23     printf("Disconnected...\n");
24
25     return 0;
26 }

```

On compile ensuite classiquement le code C en n'oubliant pas de préciser où trouver les fichiers *header* des bibliothèques nécessaires :

```
% gcc -c connexionC.c -I$POSTGRESQL_HOME/include
```

Puis on lie le code objet ainsi obtenu pour obtenir un exécutable (il faut là encore préciser les bibliothèques utilisées pour le lien ainsi que le chemin où les trouver) :

```
% gcc -o connexionC connexionC.o -L$POSTGRESQL_HOME/lib -lecpg
```

On peut ensuite exécuter le programme ainsi produit.

10.2.3. Partager des variables avec le langage hôte

Lorsque l'on veut interagir avec une base de données depuis un langage hôte, il faut pouvoir partager des variables entre SQL et le langage hôte. Ceci permet par exemple de paramétrer des requêtes ou de traiter le résultat renvoyé par une requête.

Dans notre cas où le langage hôte est C, on peut utiliser des variables C dans les requêtes SQL en les préfixant par « : » dans les requêtes SQL. Il faut également déclarer ces variables à l'intérieur d'une construction spéciale, dont la syntaxe est définie ci-après.

Syntaxe (déclaration de variables partagées) :

```
EXEC SQL BEGIN DECLARE SECTION;
type var [= default value];
...
EXEC SQL END DECLARE SECTION;
```

□

Par exemple, une insertion dans la base de données en utilisant une variable peut se faire avec le bloc de code suivant :

```
EXEC SQL BEGIN DECLARE SECTION;
  char param[20];
EXEC SQL END DECLARE SECTION;
```

```
param = "new value";
```

```
EXEC SQL INSERT INTO Relation (attribut) VALUES (:param);
```

10.2.4. Requêtes sur la base de données

Pour effectuer des requêtes sur la base de données, il faut pouvoir récupérer le résultat de la requête dans une variable C. Lorsque la requête ne renvoie qu'une seule ligne, la syntaxe est simple. On utilise une version modifiée de **SELECT** qui indique la ou les variables dans lesquelles stocker le résultat.

Syntaxe (clause **SELECT avec variables partagées) :**

```
EXEC SQL SELECT att1, ..., attn INTO :var1, ..., :varn FROM Relation;
```

□

10. SQL et autres langages

Par exemple, la requête présentée sur le listing 10.3 permet de trouver le prix de l'application Oracle (on est alors sûr que l'on va obtenir un seul résultat).

Listing 10.3 – Une requête simple sur une base de données

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Trying to connect to database\n");
5     EXEC SQL CONNECT TO 'bdpolyIN306@serv-sun1.isae.fr' USER garion;
6     printf("Connected...\n");
7
8     EXEC SQL BEGIN DECLARE SECTION;
9         int prix;
10    EXEC SQL END DECLARE SECTION;
11
12    EXEC SQL SELECT prix INTO :prix FROM logiciel WHERE nom = 'Oracle';
13
14    printf("Result: %d\n", prix);
15
16    printf("Trying to disconnect\n");
17    EXEC SQL DISCONNECT ALL;
18    printf("Disconnected...\n");
19
20    return 0;
21 }
```

Si la requête renvoie plus d'un attribut en réponse, il suffit de stocker les différentes valeurs des attributs dans des variables différentes. Comment se passe maintenant le traitement d'une requête qui renvoie plus d'une ligne comme réponse ? Il faut alors utiliser un *curseur* qui permet de parcourir les lignes de la réponse.

Syntaxe (déclaration d'un curseur sur une requête) :

```
EXEC SQL DECLARE nomCurseur CURSOR FOR SELECT... ;
```

□

Il faut ouvrir le curseur lorsque l'on en a besoin et le fermer ensuite.

Syntaxe (ouverture/fermeture d'un curseur) :

```
EXEC SQL OPEN nomCurseur;
...
EXEC SQL CLOSE nomCurseur;
```

□

On parcourt ensuite les lignes de la réponse grâce à l'instruction `FETCH`.

Syntaxe (parcours d'une réponse) :

```
EXEC SQL FETCH NEXT FROM nomCurseur INTO :var1, ..., :varn;
```

□

Par exemple, le listing 10.4 propose une requête embarquée permettant de trouver les logiciels et le prix associé dans la base de données exemple que nous utilisons. La variable `sqlca.sqlcode` permet de savoir quand il n'y a plus de ligne à parcourir dans la réponse⁴.

Listing 10.4 – Une requête plus complexe sur une base de données

```

1 #include <stdio.h>
2
3 int main() {
4     printf("Trying to connect to database\n");
5     EXEC SQL CONNECT TO 'bdpolyIN306@serv-sun1.isae.fr' USER garion;
6     printf("Connected...\n");
7
8     EXEC SQL BEGIN DECLARE SECTION;
9         char nom[20];
10        int prix;
11    EXEC SQL END DECLARE SECTION;
12
13    EXEC SQL DECLARE curseur CURSOR FOR SELECT nom, prix FROM logiciel;
14
15    EXEC SQL OPEN curseur;
16
17    while (sqlca.sqlcode == 0) {
18        EXEC SQL FETCH NEXT FROM curseur INTO :nom, :prix;
19
20        if (sqlca.sqlcode == 0) {
21            printf("Software: %s, price: %d\n", nom, prix);
22        }
23    }
24
25    EXEC SQL CLOSE curseur;
26
27    printf("Trying to disconnect\n");
28    EXEC SQL DISCONNECT ALL;
29    printf("Disconnected...\n");
30
31    return 0;
32 }
```

4. Elle a également d'autres utilités que nous n'aborderons pas ici.

10.3. JDBC et les architectures client/serveur

En mode client/serveur, un programme client s'adresse à un programme qui s'exécute sur une machine distante (le serveur) pour échanger des informations et des services. Les exemples les plus classiques sont les architectures client/serveur X (« affichage graphique » sur les stations Unix), les serveurs Web (un serveur HTTP et un client qui est un navigateur) etc.

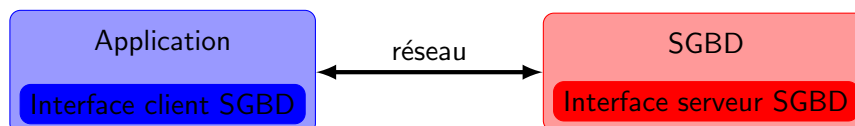


FIGURE 10.2. – Schéma général de connexion entre SGBD et programme utilisateur

Dans le cas d'un accès à une base de données, le programme ne communique pas directement avec le moteur du SGBD. Il communique avec l'interface client du SGBD qui gère le protocole de communication habituellement spécifique au SGBD. Jusqu'à récemment, les applications client/serveur étaient réalisées par le constructeur du système de gestion de bases de données. On ne pouvait donc pas changer le client ou le serveur indépendamment l'un de l'autre. Des protocoles comme ODBC ou JDBC sont nés pour pallier ce problème. ODBC a été développé pour fournir une API standard à SQL sur les plates formes Microsoft. ODBC fournit une couche d'abstraction indépendante de la base de données, mais dépendante de la plateforme (Microsoft uniquement). JDBC fournit également une couche d'abstraction indépendante de la base de données, mais également de la plateforme.

JDBC permet de développer des programmes clients (applications autonomes Java ou applets) qui accèdent au SGBD. En ce qui concerne l'accès aux bases de données, un programme Java qui utilise les services de JDBC est toujours structuré en deux couches (cf. figure 10.3) :

- la première couche est orientée vers le programme Java. Elle est composée du gestionnaire de drivers (*driver manager*) JDBC. C'est un objet Java auquel s'adressent les autres objets de l'application cliente Java pour obtenir des connexions vers les bases de données.
- la seconde couche est orientée vers le SGBD. Elle nécessite des drivers JDBC qui sont spécifiques aux bases de données auxquelles l'application cliente doit accéder. Il existe des drivers JDBC pour Oracle, PostgreSQL, MySQL, Sybase, Informix, etc. Les drivers JDBC peuvent être classés en quatre catégories (cf. figure 10.4).
- drivers de type 1 (JDBC-ODBC *bridge*). Ils définissent une interface entre l'API ODBC et l'API JDBC. Le driver transforme les appels à l'API JDBC en appels ODBC. Le driver ODBC se charge ensuite des requêtes natives en SQL. Cette solution est très coûteuse en temps, car on a trois couches d'abstraction (JDBC, ODBC et la couche native de base de données). De plus, les fonctionnalités de l'API JDBC sont limitées par celles d'ODBC.
- drivers de type 2 (moitié Java, moitié natif). Ce type de driver est une interface

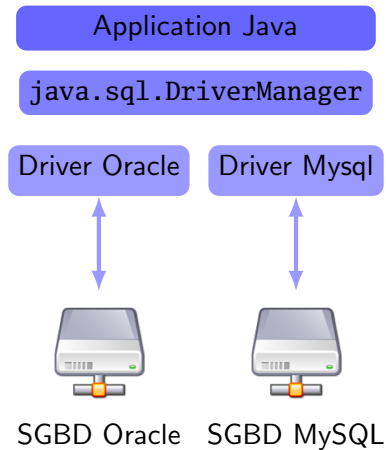


FIGURE 10.3. – Connexion de JDBC vers deux serveurs de bases de données

entre le driver manager JDBC et l'interface cliente du SGBD. Il est dédié à un SGBD particulier et est donc moins ouvert que le précédent. Par contre, il y a une couche de moins que dans le cas précédent, donc ce type de driver est beaucoup plus rapide.

- drivers de type 3. Il s'agit d'une interface entre le driver de JDBC et un service d'accès aux données que nous appelons middleware. Il encapsule un protocole de communication entre le driver du JDBC et le service middleware. Celui-ci peut être utilisé par une applet Java. Ce type de drivers permet d'avoir un fort taux d'abstraction, le serveur intermédiaire gérant par exemple les connexions aux différents serveurs.
- drivers de type 4. Il s'agit d'une interface entre le driver manager de JDBC et l'interface du serveur de SGBD. Il encapsule complètement l'interface cliente du SGBD. Notons que les drivers 3 et 4 autorisent l'utilisation d'applets contrairement aux drivers 1 et 2 car ces derniers nécessitent le chargement de codes natifs non autorisés par le protocole de sécurité défini pour les applets Java.

Actuellement, il existe des drivers de type 4 pour pratiquement tous les SGBD.

10.4. Fonctionnement

JDBC interagit avec le SGBD grâce à un driver (comme ceux qui sont décrits ci-dessus). L'accès aux données de fait en plusieurs étapes :

1. enregistrement du pilote JDBC ;
2. connexion avec la base de données ;
3. création d'une zone de description de requête ;
4. exécution de la requête ;
5. traitement des données retournées.

Nous allons les détailler dans ce qui suit.

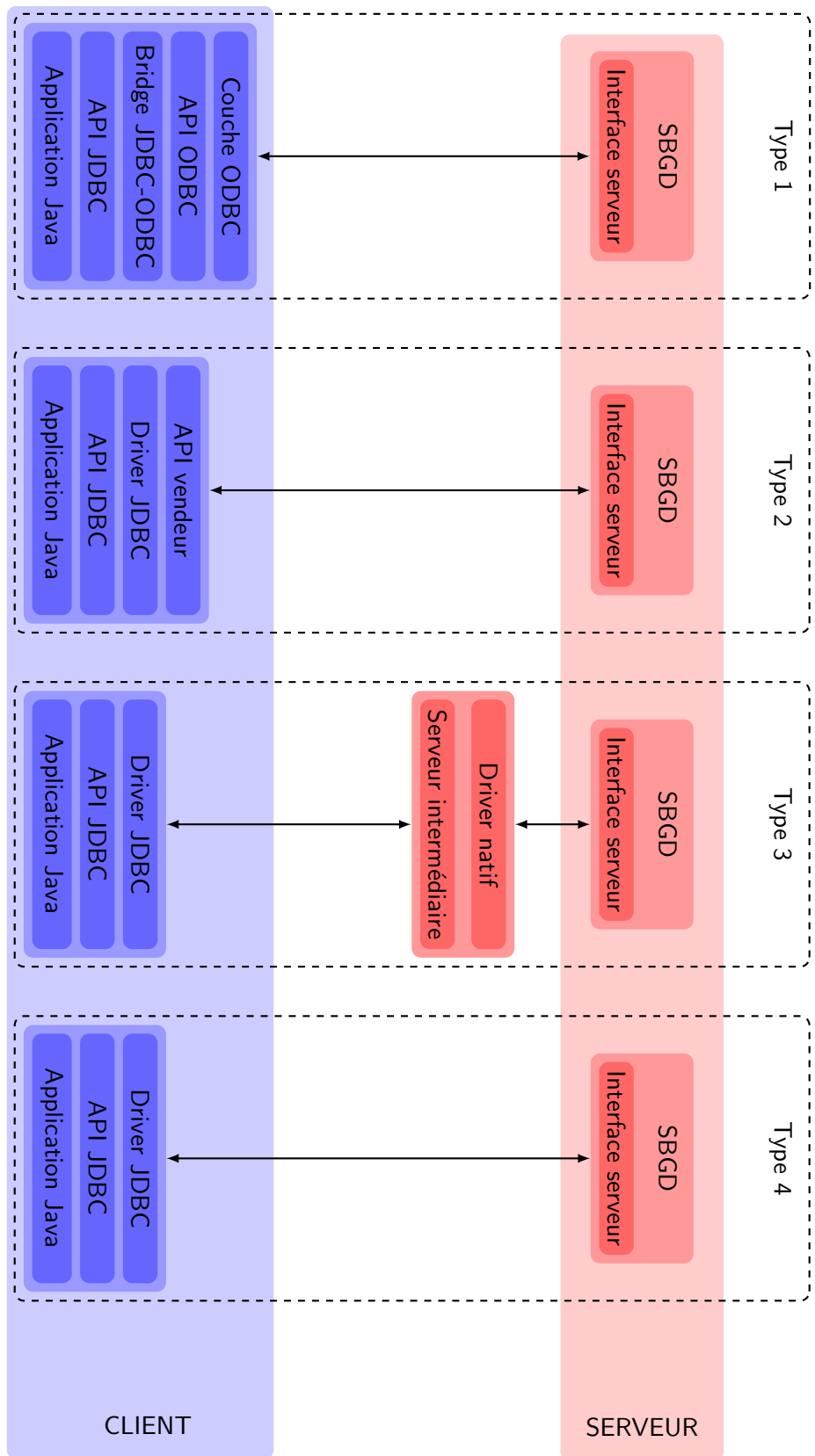


FIGURE 10.4. – Les différents types de drivers JDBC

10.4.1. Enregistrement du driver JDBC

Les connexions à une base de données sont représentées par l'interface `java.sql.Connection`. C'est une interface car l'« implantation » de cette connexion dépend de la base de données, du protocole d'accès utilisé etc. Pour obtenir une connexion vers une base de données, on utilise un objet de type `java.sql.Driver`. Les objets réalisant cette interface possèdent une méthode `connect()` qui établit la connexion vers une base de données.

Plutôt que de manipuler directement des objets de type `java.sql.Driver`, on utilise un objet de type `java.sql.DriverManager` qui permet d'enregistrer un ou plusieurs drivers depuis la même application et d'utiliser cet objet pour obtenir une ou plusieurs connexions vers les bases de données concernées. Cette classe possède les méthodes statiques nécessaires à l'obtention de la connexion.

Pour pouvoir enregistrer un driver, il suffit de demander le chargement de la classe correspondante via le *class loader* courant.

Syntaxe (chargement d'un driver JDBC) :

```
Class.forName("nomClasseDriver");
```

□

Attention, cet appel peut lever une `ClassNotFoundException` si le driver n'est pas accessible. Le chargement d'un driver JDBC enregistre automatiquement le driver auprès du gestionnaire de drivers.

Exemple 10.4.1 : Supposons que l'on veuille enregistrer un driver pour le SGBD PostgreSQL. On écrira alors :

```
try {
    Class.forName("org.postgresql.Driver");
}
catch (ClassNotFoundException e) {
    ...
}
```

10.4.2. Établissement d'une connexion

Pour référencer une base de données avec JDBC, on utilise un système d'URL (*Uniform Resource Locator*) très similaires aux URL utilisées pour accéder à un serveur Web. L'utilisation d'URL permet d'identifier de manière unique une base de données.

Syntaxe (URL d'identification d'une base de données) :

```
jdbc:<subprotocol>:<subname>
```

où `<subprotocol>` représente le driver de la base de donnée et `<subname>` un nom spécifique au driver considéré. □

Exemple 10.4.2 : Nous considérerons ici que nous voulons nous connecter via le SGBD PostgreSQL à une base de donnée `bdpolyIN306` située sur la machine `serv-sun1`. La syntaxe de l'URL est alors `jdbc:postgresql://serv-sun1/bdpolyIN306`. □

Pour obtenir une connexion, on utilise la méthode statique `getConnection` de `java.sql.DriverManager` dont la signature est :

Syntaxe (signature de `getConnection`) :

```
public static java.sql.Connection getConnection(String url,
                                               String user,
                                               String password)
        throws java.sql.SQLException
```

□

Exemple 10.4.3 : Supposons que l'on veuille maintenant se connecter à la base `bdpolyIN306` située sur `serv-sun1` en tant qu'utilisateur `garion`. On écrira alors (en supposant que l'on a importé les paquetages nécessaires) :

```
try {
    Connection connection;
    connection = DriverManager.getConnection("jdbc:postgresql://serv-sun1/bdpolyIN306",
                                           "garion");
}
catch (SQLException e) {
    ...
}
```

□

Les classes réalisant `Connection` possèdent plusieurs méthodes utiles : `getMetaData` pour obtenir des informations sur la base, méthodes de gestion des transactions, méthode `close` pour fermer la connexion etc.

10.4.3. Créer une requête SQL

Une fois la connexion établie, on dispose de différentes méthodes pour pouvoir créer une requête SQL. Ces requêtes sont encapsulées au travers d'objets de `java.sql` qui peuvent être de trois types :

- `java.sql.Statement` qui est une requête classique ;
- `java.sql.PreparedStatement` qui permettent de créer des requêtes paramétrées ou dont les paramètres ne peuvent pas être représentés sous forme d'objet de type `String` (non abordé ici). Ces requêtes sont précompilées dans le SGBD et permettent donc de pouvoir les appeler plusieurs fois de façon efficace (le compilateur de requêtes du SGBD n'est pas appelé à chaque fois). Le passage de paramètre se fait d'une façon particulière, cf. javadoc.

- `java.sql.CallableStatement` qui fait appel à une procédure ou une fonction stockée dans la base via *PMS* (cf. la section 10.1).

Nous n'allons aborder ici que les requêtes utilisant un objet de type `java.sql.Statement` (pour les autres requêtes, on pourra se référer à la javadoc correspondante). Pour créer une requête, on utilise la méthode suivante de `java.sql.Connection` :

Syntaxe (méthode `createStatement`) :

```
public java.sql.Statement createStatement() throws java.sql.SQLException
```

□

10.4.4. Exécution de la requête

Une fois l'objet de type `Statement` créé, on peut effectuer une requête SQL sur la base via une méthode sur l'objet `Statement`. La méthode utilisée dépend de la requête considérée :

Syntaxe (méthode d'exécution de requête) :

- `public java.sql.ResultSet executeQuery(String query)` pour une requête « classique » ;
 - `public int executeUpdate(String update)` pour une mise à jour de la base ;
 - `public boolean execute(String query)` pour une requête quelconque. Le booléen renvoyé permet de savoir si on a obtenu un résultat ou pas (cf. section 10.4.5).
- Toutes ces méthodes peuvent lever une `java.sql.SQLException`. □

On peut également utiliser des *batch updates* qui permettent de regrouper plusieurs mises à jour dans un seul objet de type `Statement`. Ceci permet d'améliorer les performances des requêtes.

10.4.5. Traitement de la requête

Dans le cas d'une requête de type `executeQuery`, la méthode renvoie un objet de type `java.sql.ResultSet`. Cet objet contient les résultats demandés par la requête. Dans le cas d'une requête de type `execute`, si le booléen renvoyé est vrai, on utilise la méthode `getResultSet` de `Statement` pour récupérer la réponse.

Les objets de type `java.sql.ResultSet` encapsulent la réponse à la requête. Ils « contiennent » les lignes correspondant à la requête demandée. Pour parcourir ces lignes, on utilise la méthode `next()` de `ResultSet`. C'est le même principe qu'un itérateur sur une collection. On peut alors accéder aux colonnes de la réponse en utilisant les méthodes appropriées parmi lesquelles :

- `public String getString(String nomColonne)` ;
- `public float getFloat(String nomColonne)` ;
- `public float getFloat(int numColonne)`.

Attention, les types renvoyés par ces méthodes sont des types de Java qui ne correspondent pas forcément aux types SQL.

Remarque : on peut également utiliser des objets de « type » `ResultSet` plus complexes comme qui permettent une navigation plus aisée (retours possibles dans la liste des lignes par exemple) ou de modifier via le `ResultSet` la ou les tables concernées. □

Exemple 10.4.4 : Supposons que l'on exécute la requête suivante :

```
ResultSet res = statement.executeQuery("SELECT nom, prix FROM Logiciel);
```

On peut alors parcourir les résultats avec :

```
while (rs.next()) {
    System.out.println(rs.getString("nom") +
        rs.getFloat("prix"));
}
```

□

Il se peut que l'on ne connaisse pas la structure du résultat renvoyé (cas d'une requête paramétrée par l'utilisateur par exemple). Pour obtenir ces informations, on utilise un objet de type `java.sql.ResultSetMetaData` obtenu via la méthode de `java.sql.ResultSet` suivante :

Syntaxe (méthode `getMetaData`) :

```
public java.sql.ResultSetMetaData getMetaData() throws java.sql.SQLException
```

□

Les objets de type `java.sql.ResultSetMetaData` possèdent plusieurs méthodes utiles :

- `getColumnCount()` pour avoir le nombre de colonnes ;
- `getColumnName(int i)` pour avoir le nom de la colonne `i` ;
- `getColumnType(int i)` pour avoir le type de la colonne `i` (représenté sous la forme d'un entier. Voir `java.sql.Types` pour la correspondance).

10.5. Un exemple complet

Un exemple complet est présenté sur le listing 10.5.

Listing 10.5 – Un exemple de requête effectué depuis un programme Java

```
1 import java.sql.*;
2
3 /**
4  * Une connexion sur une base PostgreSQL.
5  *
6  * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
7  * @version 1.0
8  */
9 public class Connexion {
10
```

```

11 public static void main(String[] args) {
12     try {
13         System.out.println("Chargement du driver et " +
14                             "ouverture de la connexion");
15         Class.forName("org.postgresql.Driver");
16
17         String url = "jdbc:postgresql://serv-sun1.isae.fr/bdpolyIN306";
18         String user = "garion";
19         Connection connection = DriverManager.getConnection(url,
20                                                         user,
21                                                         "");
22
23         // on demande une requete
24         Statement statement = connection.createStatement();
25         ResultSet res = statement.executeQuery("SELECT * " +
26                                             "FROM Ordinateur");
27
28         // renseignements sur le ResultSet
29         ResultSetMetaData m = res.getMetaData();
30         for (int i = 1; i <= m.getColumnCount(); i++) {
31             System.out.println(m.getColumnName(i) + " de type " +
32                               m.getColumnType(i));
33         } // end of for (int i = 1; i <= m.getColumnCount(); i++)
34         System.out.println();
35
36         // on parcourt le ResultSet
37         while (res.next()) {
38             System.out.println(res.getString("type") + " " +
39                               res.getFloat("prix"));
40         } // end of while (res.next())
41
42         // on ferme proprement
43         res.close();
44         statement.close();
45         connection.close();
46     } catch (ClassNotFoundException e) {
47         e.printStackTrace();
48     } // end of catch
49     catch (SQLException e) {
50         e.printStackTrace();
51     } // end of try-catch
52
53 } // end of main()
54 }

```


10.6. Conclusion

Nous venons de présenter ici deux façons d'intégrer SQL dans un langage de programmation hôte : une intégration statique dans le langage C et une intégration dynamique dans le langage Java.

L'intégration statique de SQL dans C possède plusieurs avantages : la gestion des variables partagées entre les deux langages est facilitée, le code SQL est vérifié par le précompilateur `epgc` que nous avons utilisé et surtout le code SQL reste aussi proche que celui que l'on peut écrire dans l'interpréteur. On remarquera qu'il existe également une extension standardisée (cf. [30]) du langage Java pour y intégrer statiquement SQL : SQLJ. SQLJ est fourni par un certain nombre de vendeurs, Oracle ou IBM par exemple[15]. Malheureusement, le SGBD PostgreSQL que nous utilisons ne fournit pas cette extension. On remarquera également que la présentation qui a été faite reste sommaire : nous ne parlons pas de la gestion des valeurs NULL, des résultats de requêtes vides et de la gestion des erreurs par exemple.

L'intégration dynamique de SQL, comme nous l'avons vue à travers JDBC, permet quant à elle d'éviter de passer par une phase de précompilation. On remarquera également que les précompilateurs SQL statique transforme en fait le code statique en appels dynamiques à des fonctions ou des méthodes du langage hôte permettant d'exécuter du code SQL sur un serveur. Par contre, l'écriture de programme utilisant des appels SQL dynamiques est plus lourde, la syntaxe SQL étant la plupart du temps « noyée » dans des chaînes de caractères du langage hôte. On remarquera également que l'on peut maintenant utiliser des annuaires comme JNDI (*Java Naming and Directory Interface*) pour localiser des bases de données via l'interface `javax.sql.DataSource`. On peut alors profiter de plus de services que dans le cas d'une connexion obtenue « classiquement » par le `DriverManager` : *pool* de connexions, transactions etc.

Enfin, on remarquera qu'il vaut mieux utiliser des requêtes préparées (*prepared statement*) dans les deux cas si l'on veut les répéter un grand nombre de fois : la requête sera optimisée par le SGBD et on pourra faire varier les paramètres de la requête sans que le SGBD n'ait forcément à la recompiler. Par exemple, 10000 exécutions de la même requête simple fait gagner 25% de temps en utilisant une requête préparée (cf. listings 10.6 et 10.7).

Listing 10.6 – Un exemple de requête simple exécutée 10000 fois

```

1 import java.sql.*;
2
3 /**
4  * Une requete simple repetee 10000 fois
5  *
6  * Created: Tue Jul 14 14:35:11 2009
7  *
8  * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
9  * @version 1.0
10 */
```

```

11 public class RequeteSimple {
12
13     public static void main(String[] args) {
14         try {
15             Class.forName("org.postgresql.Driver");
16
17             String url = "jdbc:postgresql://serv-sun1.isae.fr/bdpolyIN306";
18             String user = "garion";
19             Connection connection = DriverManager.getConnection(url,
20                                                                 user,
21                                                                 "");
22
23             // on demande une requete
24             Statement statement = connection.createStatement();
25
26             for (int i = 0; i < 10000; i++) {
27                 statement.executeQuery("SELECT * " +
28                                     "FROM Ordinateur");
29             }
30         } catch (ClassNotFoundException e) {
31             e.printStackTrace();
32         } // end of catch
33         catch (SQLException e) {
34             e.printStackTrace();
35         } // end of try-catch
36     }
37 }

```

Listing 10.7 – Un exemple de requête préparée exécutée 10000 fois

```

1 import java.sql.*;
2
3 /**
4  * Une requete preparee repetee 10000 fois
5  *
6  * Created: Tue Jul 14 14:35:11 2009
7  *
8  * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
9  * @version 1.0
10 */
11 public class RequetePrepree {
12
13     public static void main(String[] args) {
14         try {
15             Class.forName("org.postgresql.Driver");

```

10. SQL et autres langages

```
16
17     String url = "jdbc:postgresql://serv-sun1.isae.fr/bdpolyIN306";
18     String user = "garion";
19     Connection connection = DriverManager.getConnection(url,
20                                                         user,
21                                                         "");
22
23     // on demande une requete
24     PreparedStatement statement;
25     statement = connection.prepareStatement("SELECT * " +
26                                         "FROM Ordinateur");
27
28     for (int i = 0; i < 10000; i++) {
29         statement.execute();
30     }
31 } catch (ClassNotFoundException e) {
32     e.printStackTrace();
33 } // end of catch
34 catch (SQLException e) {
35     e.printStackTrace();
36 } // end of try-catch
37     }
38 }
```

11. Les dépendances fonctionnelles et multivaluées. Décomposition de relations et normalisation

11.1. Objectifs

On souhaite définir une démarche rigoureuse pour la conception d'un schéma relationnel optimal. La démarche que nous définissons dans ce chapitre est dite « démarche directe », elle ne fait pas appel au modèle entité-association¹. On dispose donc d'un modèle relationnel représentant un ensemble d'informations structuré.

Pour préciser ce que nous entendons par schéma relationnel optimal, reprenons la relation *Type Ordinateur* que nous avons définie dans le chapitre 6 :

Type Ordinateur {*type*, *cons*, *pays*, *rev*, *nb_postes*, *mem_max*, *uc*, *capa_lec*, *capa_disk*}

Supposons que nous ayons les tuples suivants pour la relation :

<i>type</i>	<i>cons</i>	<i>pays</i>	<i>rev</i>	<i>nb_postes</i>	<i>mem_max</i>	<i>uc</i>	<i>capa_lec</i>	<i>capa_disk</i>
Micral 75	Bull	France	Camif	1	512	IN80486	1044	40
Micral 60	Bull	France	Camif	1	256	IN80486	1044	20
Mac II	Apple	USA	Apple	1	256	Mo68020	1044	60

On s'aperçoit que plusieurs problèmes apparaissent dans ce schéma :

- l'information nous indiquant que le constructeur Bull a pour pays la France apparaît deux fois dans l'instance de la relation. Elle apparaîtra à chaque fois que l'on rajoutera un type d'ordinateur construit par Bull ;
- si l'on change par exemple le pays d'origine de Bull dans le premier tuple, il n'y a aucune garantie que le pays d'origine soit également changé dans le deuxième tuple ;
- si l'on efface le troisième tuple de l'instance, on perd l'information qui nous indiquait que Apple était un constructeur dont le pays d'origine est les USA.

Nous allons donc chercher à utiliser une méthode pour définir « correctement » un schéma relationnel. Les objectifs d'une telle approche sont de :

- minimiser les redondances d'informations,
- éviter les « anomalies de mises à jour »,
- privilégier les dépendances entre attributs.

Nous illustrerons ces objectifs à chaque fois qu'il sera possible dans ce qui suit.

1. Les techniques de normalisation peuvent évidemment s'appliquer aux relations obtenues en traduisant un modèle entité-association dans un modèle relationnel.

11.2. Dépendances fonctionnelles

11.2.1. Définition

Nous allons dans cette section définir la notion de *dépendance fonctionnelle* entre attributs (nous l'avons déjà aperçue avec la définition de la notion de clé dans le chapitre 6).

Définition 11.2.1.

Soient R une relation, $i \in \mathbb{N}$ tel que $i \geq 1$, A_1, \dots, A_n n attributs de R et B un attribut de R . Soient t_1 et t_2 deux tuples de R . Il existe une dépendance fonctionnelle (notée DF) entre A_1, \dots, A_n et B ssi pour tous tuples t_1 et t_2 de R , si t_1 et t_2 s'accordent sur les valeurs des attributs A_1, \dots, A_n , alors ils ont la même valeur pour l'attribut B .

On le note $R \models A_1 \dots A_n \longrightarrow B$.

On dit également que $A_1 \dots A_n$ détermine fonctionnellement B dans R . □

On peut remarquer que l'appellation « dépendance fonctionnelle » est justifiée : dans une relation R telle que celle proposée dans la définition, pour chaque valeur des attributs A_1, \dots, A_n , on ne peut trouver qu'une seule valeur de B . On remarquera également que les attributs d'une relation dépendent fonctionnellement de ses clés.

On se permettra parfois par raccourci d'écriture de ne pas mettre le nom de la relation et le symbole \models .

Les dépendances fonctionnelles permettent de restreindre les instances possibles d'un schéma relationnel. Elles sont donc très importantes pour la représentation des données dans une base de données. Elles doivent normalement être spécifiées en même temps que le schéma de la relation.

Attention, les dépendances fonctionnelles sont des propriétés qui s'appliquent à toutes les instances possibles de la relation et pas simplement à une instance particulière.

Exemple 11.2.1 : Reprenons la relation *Type Ordinateur*. On voit aisément que

$type \longrightarrow cons$

car pour chaque type d'ordinateur, on ne peut (*a priori* !) avoir qu'un constructeur.

Si l'on considère maintenant la relation *Logiciel* (cf. chapitre 6), on voit que

$nom_log \text{ revendeur} \longrightarrow prix$

Connaissant un logiciel et son revendeur, on peut déterminer son prix. □

On utilisera le raccourci d'écriture suivant :

$A_1 \dots A_n \longrightarrow B_1 \dots B_m$ représente les m dépendances fonctionnelles suivantes :

$$A_1 \dots A_n \longrightarrow B_1$$

...

$$A_1 \dots A_n \longrightarrow B_m$$

11.2.2. Notion de conséquence

On peut donner une définition de la déduction de dépendances fonctionnelles à partir d'un autre ensemble de dépendances.

Définition 11.2.2.

Soit S un ensemble de dépendances fonctionnelles sur une relation R .

Une dépendance fonctionnelle f sur R est dit conséquence de S si et seulement si $R \models S$ implique $R \models f$. On le note $S \models f$.

Un ensemble de dépendances fonctionnelles T sur R est conséquence de S si et seulement si pour toute dépendance $f \in T$, $S \models f$. On le note $S \models T$. \square

On peut par exemple montrer facilement que $\{X \rightarrow Y\} \models XZ \rightarrow YZ$ (c'est ce que l'on appelle la règle d'augmentation).

11.3. Règles et propriétés des dépendances fonctionnelles

Nous présentons dans cette section quelques règles permettant de travailler avec les dépendances fonctionnelles. Ces règles vont nous permettre de déduire à partir d'un ensemble de dépendances fonctionnelles d'autres dépendances fonctionnelles.

11.3.1. Règle de séparation/combinaison

Cette règle permet simplement d'appliquer le raccourci d'écriture que nous avons introduit dans la section précédente.

Proposition 11.3.1.

Soient R une relation et $A_1, \dots, A_n, B_1, \dots, B_m$ des attributs de R . Alors :

- $A_1 \dots A_n \rightarrow B_1 \dots B_m$ peut être remplacée par les m dépendances fonctionnelles $A_1 \dots A_n \rightarrow B_j$ pour $j \in \{1, \dots, m\}$;
- les m dépendances fonctionnelles $A_1 \dots A_n \rightarrow B_j$ pour $j \in \{1, \dots, m\}$ peuvent être remplacées par la dépendance fonctionnelle $A_1 \dots A_n \rightarrow B_1 \dots B_m$. \square

11.3.2. Dépendances fonctionnelles triviales

Certaines dépendances fonctionnelles sont triviales : celles dont la partie droite est un sous ensemble d'attributs de la partie gauche par exemple.

Définition 11.3.1.

Soit $A_1 \dots A_n \rightarrow B_1 \dots B_m$ une dépendance fonctionnelle. On dit qu'elle est :

- triviale si $\{B_1, \dots, B_m\} \subseteq \{A_1, \dots, A_n\}$;
- non triviale si $\exists j \in \{1, \dots, m\}$ tel que $B_j \notin \{A_1, \dots, A_n\}$;
- complètement non triviale si pour tout $j \in \{1, \dots, m\}$ $B_j \notin \{A_1, \dots, A_n\}$. \square

11.3.3. Dérivation syntaxique et règles d'inférence d'Armstrong

Armstrong a défini dans [1] un axiome et deux règles permettant de raisonner sur les dépendances fonctionnelles.

Définition 11.3.2 (système formel d'Armstrong).

Soient R une relation et $A_1, \dots, A_n, B_1, \dots, B_m, C_1, \dots, C_k$ des attributs de R .

L'axiome d'Armstrong, appelé axiome de réflexivité, est le suivant :

$$\text{pour tout } \{B_1, \dots, B_m\} \subseteq \{A_1, \dots, A_n\} \quad \vdash A_1 \dots A_n \longrightarrow B_1 \dots B_m$$

La règle d'inférence d'augmentation est la suivante : si $A_1 \dots A_n \longrightarrow B_1 \dots B_m$ alors $A_1 \dots A_n C_1 \longrightarrow B_1 \dots B_m C_1$. On la note :

$$A_1 \dots A_n \longrightarrow B_1 \dots B_m \quad \vdash \quad A_1 \dots A_n C_1 \longrightarrow B_1 \dots B_m C_1$$

La règle d'inférence de transitivité est la suivante : si $A_1 \dots A_n \longrightarrow B_1 \dots B_m$ et $B_1 \dots B_m \longrightarrow C_1 \dots C_k$ sont deux dépendances fonctionnelles de R , alors $A_1 \dots A_n \longrightarrow C_1 \dots C_k$. On la note :

$$\{A_1 \dots A_n \longrightarrow B_1 \dots B_m, B_1 \dots B_m \longrightarrow C_1 \dots C_k\} \quad \vdash \quad A_1 \dots A_n \longrightarrow C_1 \dots C_k \quad \square$$

À partir d'un ensemble de dépendances fonctionnelles S sur une relation R , l'utilisation de cet axiome et l'application de ces règles permet de déduire l'ensemble des dépendances fonctionnelles de cette relation déductibles de S . On définit donc la notion de dérivation de dépendance fonctionnelle.

Définition 11.3.3 (dérivation de dépendance fonctionnelle).

Soient S un ensemble de dépendance fonctionnelle et f une dépendance fonctionnelle. On dit que S dérive f ou que f est déduite de S , noté $S \vdash f$ si et seulement si une suite de dépendances fonctionnelles f_1, \dots, f_n telle que :

- $f_n = f$
- $\forall i \in \{1, \dots, n\}$ $f_i \in S$ ou f_i est déduite de $\{f_1, \dots, f_{i-1}\}$ en utilisant le système d'Armstrong.

f_1, \dots, f_n est appelée une dérivation de f à partir de S . □

Ce système formel propose une axiomatique complète et valide.

Théorème 11.3.1.

Soit S un ensemble de dépendances fonctionnelles sur une relation R .

- si f peut être dérivée de S en utilisant les règles d'Armstrong, alors f est une dépendance fonctionnelle sur R conséquence de S (validité des règles).
- si f est une dépendance fonctionnelle sur R conséquence de S , alors f peut être dérivée de S en utilisant les règles d'Armstrong (complétude des règles).

On peut donc écrire :

$$S \models f \Leftrightarrow S \vdash f \quad \square$$

Preuve :

On considère dans toute la suite de la preuve un ensemble de dépendances fonctionnelles S sur une relation $R(\Delta)$.

- la démonstration de la validité se fait par induction sur le nombre de règles utilisées :
 - au rang 0, cela signifie que soit $f \in S$, soit on a utilisé l'axiome de réflexivité pour trouver f . D'après la définition d'une dépendance fonctionnelle, si f est issue de l'axiome de réflexivité, alors f est bien une dépendance fonctionnelle issue de S .
 - supposons la relation d'induction vraie au rang $n \geq 0$ (donc si on a utilisé n règles pour dériver f , alors f est bien déduite de S). Supposons que l'on ait utilisé $n + 1$ règles pour dériver f . Deux cas se présentent :
 - soit la dernière règle utilisée est la règle d'augmentation. f est donc de la forme $A_1 \dots A_n C_1 \longrightarrow B_1 \dots B_m C_1$ et est issue d'une règle de la forme $A_1 \dots A_n \longrightarrow B_1 \dots B_m$. $A_1 \dots A_n \longrightarrow B_1 \dots B_m$ a été produite en utilisant n règles d'Armstrong donc d'après l'hypothèse d'induction, $A_1 \dots A_n \longrightarrow B_1 \dots B_m$ est bien une dépendance fonctionnelle sur R conséquence de S . D'après la définition d'une dépendance fonctionnelle, si $A_1 \dots A_n \longrightarrow B_1 \dots B_m$ est une dépendance fonctionnelle sur R conséquence de S , alors $A_1 \dots A_n C_1 \longrightarrow B_1 \dots B_m C_1$ est également une (C_1 est bien un attribut de R et on l'« ajoute » de chaque côté). Donc $A_1 \dots A_n C_1 \longrightarrow B_1 \dots B_m C_1$ est bien une dépendance sur R conséquence de S .
 - soit la dernière règle utilisée est la règle de transitivité, donc f est de la forme $A_1 \dots A_n \longrightarrow C_1 \dots C_k$ et est conséquence de $A_1 \dots A_n \longrightarrow B_1 \dots B_m$ et $B_1 \dots B_m \longrightarrow C_1 \dots C_k$.
 $A_1 \dots A_n \longrightarrow B_1 \dots B_m$ et $B_1 \dots B_m \longrightarrow C_1 \dots C_k$ ont été produites en utilisant au plus n règles d'Armstrong, donc d'après l'hypothèse d'induction ce sont bien des dépendances fonctionnelles sur R conséquences de S .
 Soient t_1 et t_2 deux tuples de R s'accordant sur les valeurs des attributs $A_1 \dots A_n$. Comme $A_1 \dots A_n \longrightarrow B_1 \dots B_m$ est une dépendance fonctionnelle, les valeurs de B_1, \dots, B_m pour t_1 et t_2 vont être identiques. Or $B_1 \dots B_m \longrightarrow C_1 \dots C_k$ est une dépendance fonctionnelle sur R donc les valeurs de C_1, \dots, C_k sont les mêmes pour t_1 et t_2 . Donc $A_1 \dots A_n \longrightarrow C_1 \dots C_k$ est bien une dépendance fonctionnelle sur R conséquence de S . ■
- la démonstration de la complétude se fait par l'absurde. Supposons que $f = X \longrightarrow Y$ soit une dépendance fonctionnelle sur R conséquence de S et que f ne puisse pas être dérivée de S en utilisant les règles d'Armstrong.
 Appelons X^+ l'ensemble des attributs Z tels que $X \longrightarrow Z$ soit déductible de S par les axiomes d'Armstrong. On sait avec l'axiome de réflexivité que $X \subseteq X^+$.
 Considérons l'instanciation de R suivante² :

2. J'ai utilisé des entiers comme valeurs pour faire plus simple mais on peut choisir des éléments plus généraux

11. Dépendances fonctionnelles

X^+					$\Delta - X^+$			
1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1

Soit $V \rightarrow W$ une dépendance fonctionnelle de S telle que $V \subseteq X$. L'instanciation précédente de R vérifie $V \rightarrow W$. Supposons en effet que $V \rightarrow W$ ne soit pas vérifiée par l'instanciation de R . Comme $V \subseteq X$, $V \subseteq X^+$ donc $X \rightarrow V$. Soit A un attribut contenu dans W , alors $W \rightarrow A$. Donc en appliquant la règle de transitivité sur $X \rightarrow V$, $V \rightarrow W$ et $W \rightarrow A$, on obtient $X \rightarrow A$. Donc $A \in X^+$. Donc $V \rightarrow W$ est bien vérifiée par l'instanciation de R .

$X \rightarrow Y$ est conséquence de S , d'après ce qui précède, l'instanciation de R vérifie $X \rightarrow Y$, donc $Y \subseteq X^+$. Donc chaque attribut de Y est contenu dans X^+ , donc chaque attribut de Y peut être déduite de X par les règles d'Armstrong.

11.3.4. Calcul de la fermeture d'un ensemble d'attributs

Les dépendances fonctionnelles jouent un rôle important dans la normalisation des relations. Il est primordial de pouvoir calculer l'ensemble des dépendances fonctionnelles pouvant être déduites d'un ensemble de départ (ce que l'on appelle la *fermeture* d'un ensemble de dépendances fonctionnelles). Le théorème d'Armstrong présenté précédemment nous assure que l'on peut calculer l'ensemble des dépendances fonctionnelles déductibles d'un ensemble à partir de trois règles d'Armstrong. Ce calcul est souvent très fastidieux.

Pour pouvoir calculer efficacement les dépendances fonctionnelles déduites d'un ensemble, on peut s'appuyer sur un algorithme permettant de calculer la fermeture d'un ensemble d'attributs. La fermeture d'un ensemble d'attributs est l'ensemble des attributs déterminés fonctionnellement par l'ensemble de départ relativement à un ensemble de dépendances fonctionnelles.

Définition 11.3.4.

Soient $\{A_1, \dots, A_n\}$ un ensemble d'attributs et S un ensemble de dépendances fonctionnelles. $\{A_1, \dots, A_n\}^+$, appelé fermeture de $\{A_1, \dots, A_n\}$ est l'ensemble des attributs B tels que toute relation satisfaisant les dépendances fonctionnelles de S satisfait également $A_1 \dots A_n \rightarrow B$. □

Pour pouvoir vérifier si $X \rightarrow Y$ est une dépendance fonctionnelle déduite d'un ensemble de DF S , il suffit donc de vérifier que $Y \in \{X\}^+$.

L'algorithme 11.3.1 permet de calculer la fermeture d'un ensemble d'attributs.

Comme pour le système formel d'Armstrong, il existe un théorème de validité/complétude pour cet algorithme.

Théorème 11.3.2.

L'algorithme de calcul de fermeture est valide et complet. Plus précisément, soient S un ensemble de dépendances fonctionnelles sur une relation R et $\{A_1, \dots, A_n\}$ des attributs de R . Alors $B \in \{A_1, \dots, A_n\}^+$ si et seulement si $A_1 \dots A_n \rightarrow B$ peut être déduite de S . □

Algorithme 11.3.1 : Calcul de la fermeture d'un ensemble d'attributs X relativement à un ensemble de dépendances fonctionnelles S

entrées : un ensemble d'attributs X et un ensemble de dépendances fonctionnelles S

sortie : la fermeture de X par rapport à S

```

1 closure ← X ;
2 répéter
3   card ← |closure| ;
4   pour chaque  $Y \rightarrow Z \in S$  faire
5     si  $Y \subseteq \text{closure}$  alors
6       closure ← closure  $\cup$  {Z} ;
7     fin
8   fin
9 jusqu'à card = |closure| ;
10 retourner closure ;
```

Preuve :

La preuve de ce théorème se fait classiquement en deux parties. On considère dans toute la preuve que S est un ensemble de dépendances fonctionnelles sur une relation R et $\{A_1, \dots, A_n\}$ sont des attributs de R .

1. montrons que si $B \in \{A_1, \dots, A_n\}^+$, alors $A_1 \dots A_n \rightarrow B$ peut être déduite de S .
Pour cela, la preuve s'effectue par récurrence sur le nombre d'« itérations » de l'algorithme 11.3.1 nécessaires à l'obtention de B .
 - au rang 0, la preuve est évidente : on n'a pas appliqué le cœur de l'algorithme et B est un des attributs de départ (on a une dépendance triviale).
 - supposons la relation vraie au rang $n \geq 0$: tous les attributs ajoutés en utilisant n fois le cœur de l'algorithme conduisent bien à une dépendance fonctionnelle. Supposons que B ait été ajouté grâce à une règle $C_1 \dots C_m \rightarrow B$. On sait par hypothèse de récurrence que R satisfait $A_1 \dots A_n \rightarrow C_1 \dots C_m$. Par transitivité on en déduit bien que $A_1 \dots A_n \rightarrow B$.
2. supposons que $A_1 \dots A_n \rightarrow B$ soit une dépendance fonctionnelle déduite de S , mais que $B \notin \{A_1, \dots, A_n\}^+$. Nous allons construire une instance de relation qui satisfait toutes les dépendances de S sauf $A_1 \dots A_n \rightarrow B$ et ainsi montrer que ce n'est pas possible.

L'instance I de relation que nous construisons contient exactement deux tuples qui s'accordent sur les valeurs de $\{A_1, \dots, A_n\}^+$, mais qui ne s'accordent pas sur les autres attributs de R . Dans ce cas, $A_1 \dots A_n \rightarrow B$ n'est pas satisfaite par I (car $B \notin \{A_1, \dots, A_n\}^+$).

Soit $C_1 \dots C_m \rightarrow C$ une dépendance fonctionnelle de S . Supposons que I ne vérifie pas $C_1 \dots C_m \rightarrow C$. Dans ce cas, comme I ne possède que deux tuples, cela signifie que $\{C_1, \dots, C_m\} \subseteq \{A_1, \dots, A_n\}^+$ et que $C \notin \{A_1, \dots, A_n\}^+$. Or l'algorithme nous assure que $C_1 \dots C_m \rightarrow C$ sera utilisé et que C sera donc ajouté

11. Dépendances fonctionnelles

à $\{A_1, \dots, A_n\}^+$. Ce n'est donc pas possible. ■

On remarquera que la complexité de ce cet algorithme est $|S| \times |R|$ (où $|R|$ est le nombre d'attributs de R). Cet algorithme est donc assez coûteux. . .

11.3.5. Couverture irredondante ou minimale

Lorsque l'on travaille avec des dépendances fonctionnelles sur une relation donnée, on cherche à avoir un nombre minimal de dépendances permettant de générer toutes les autres dépendances de l'ensemble de départ.

Dans un premier temps, nous allons introduire une notion de minimalité pour les dépendances fonctionnelles via la notion de *dépendance fonctionnelle élémentaire*.

Définition 11.3.5.

Soient R une relation et A_1, \dots, A_n, B des attributs de R tel que l'on ait $A_1 \dots A_n \longrightarrow B$. $A_1 \dots A_n \longrightarrow B$ est une dépendance fonctionnelle élémentaire si et seulement si il n'existe pas $\{A_i, \dots, A_j\} \subset \{A_1, \dots, A_n\}$ tel que l'on ait $A_i \dots A_j \longrightarrow B$ pour R . □

Attention, une dépendance fonctionnelle élémentaire n'a qu'un seul attribut en « partie droite » de la DF.

On remarquera que si les attributs d'une relation dépendent tous fonctionnellement des clés et des superclés de la relation, la dépendance existant avec les clés est une dépendance fonctionnelle élémentaire contrairement aux superclés. On pourrait donc donner une nouvelle définition d'une clé pour une relation R : il s'agit d'un sous-ensemble d'attributs de R tel que tous les attributs de R dépendent fonctionnellement élémentairement de ce sous-ensemble.

Nous allons maintenant considérer les dépendances qui peuvent se déduire par transitivité des autres dépendances. On dit alors qu'elles sont *redondantes*. Pour vérifier que $X \longrightarrow A$ est redondante à partir d'un ensemble de dépendances fonctionnelles S , il suffit de calculer $\{X\}^+$ à partir de $S - \{X \longrightarrow A\}$ et de vérifier que $A \in \{X\}^+$. Si oui, alors $X \longrightarrow A$ est redondante.

Définition 11.3.6.

Soit S un ensemble de dépendances fonctionnelles sur une relation R . On appelle *couverture irredondante* de S l'ensemble de dépendances fonctionnelles élémentaires S' tel que :

- toute dépendance fonctionnelle élémentaire de S est dans la fermeture de S' ;
- aucune dépendance de S' n'est redondante. □

Nous allons maintenant chercher à obtenir une couverture minimale (on dit aussi irredondante) d'un ensemble de DF S . Il suffit pour cela de suivre la définition et de simplifier les DF pour n'obtenir dans un premier temps que des DF élémentaires, puis d'éliminer les DF redondantes (cf. algorithme 11.3.2).

Exemple 11.3.1 : Prenons l'ensemble de DF suivant pour une relation R :

- $A \longrightarrow BC$

Algorithme 11.3.2 : Calcul de la couverture irredondante d'un ensemble de dépendances fonctionnelles S

entrées : un ensemble de dépendances fonctionnelles S

sortie : la couverture irredondante de S

- 1 $S_1 \leftarrow$ décomposer les DF de S pour n'avoir qu'un seul attribut à droite ;
 - 2 $S_2 \leftarrow$ éliminer les attributs en surnombre à gauche dans les DF de S_1 ;
 - 3 $S_3 \leftarrow$ éliminer les DF redondantes dans S_2 ;
 - 4 **retourner** S_3 ;
-

— $AB \rightarrow D$

— $D \rightarrow C$

$A \rightarrow BC$ est en fait un raccourci d'écriture pour $A \rightarrow B$ et $A \rightarrow C$. En utilisant $A \rightarrow B$ et la règle d'augmentation avec A , on obtient $A \rightarrow AB$. Avec la règle de transitivité, on obtient $A \rightarrow D$ et donc $AB \rightarrow D$ n'est pas une dépendance fonctionnelle élémentaire. On obtient pour dépendances fonctionnelles élémentaires :

— $A \rightarrow B$

— $A \rightarrow C$

— $A \rightarrow D$

— $D \rightarrow C$

Parmi ces DF, $A \rightarrow C$ est redondante : on peut l'obtenir en appliquant la règle de transitivité sur $A \rightarrow D$ et $D \rightarrow C$. La couverture irredondante de notre ensemble de départ est donc :

— $A \rightarrow B$

— $A \rightarrow D$

— $D \rightarrow C$

□

11.4. Normalisation de relations

Nous avons vu dans la section 11.1 qu'un schéma relationnel « mal » construit pouvait entraîner des problèmes de cohérence sur la base de données (anomalies de mise à jour, de suppression par exemple). La normalisation des relations propose de résoudre ces problèmes en décomposant la relation problématique en plusieurs relations.

11.4.1. Décomposition de relations

L'objectif de la décomposition d'une relation R est de construire des sous-relations R_1, \dots, R_n telles que $R = R_1 \bowtie \dots \bowtie R_n$. Il reste deux propriétés à vérifier :

- $R_1 \bowtie \dots \bowtie R_n$ doit permettre de calculer exactement la relation initiale : en particulier, on doit retrouver tous les tuples de R (et pas plus !). On dit alors que la jointure est *conservatrice* ou *sans perte d'informations* (SPI) ;
- les dépendances fonctionnelles de R doivent être préservées par la jointure des sous-relations. On dit que la relation est *sans perte de dépendance* (SPD).

11. Dépendances fonctionnelles

Supposons que $R_1 \bowtie \dots \bowtie R_n$ soit une décomposition de R . Comment vérifier que cette décomposition est SPI conserve les dépendances fonctionnelles de R ?

Décomposition SPI

Pour vérifier que la décomposition est SPI, une technique simple, dite *méthode de poursuite*, est disponible. Il suffit de vérifier que « $R_1 \bowtie \dots \bowtie R_n \subseteq R$ »³ de la façon suivante (on suppose que les attributs de R sont A_1, \dots, A_m) :

- considérer un tuple (a_1, \dots, a_m) de $R_1 \bowtie \dots \bowtie R_n$
- construire un tableau avec pour colonnes les attributs de R et lignes chaque R_i
- remplir le tableau : pour chaque ligne, on utilise les a_j lorsque la relation R_i contient l'attribut en question et on remplace par une variable sinon
- ensuite, pour chaque DF $X \rightarrow Y$, on considère deux lignes du tableau. Si ces deux lignes s'entendent sur les valeurs de X , alors pour chaque attribut A_j de Y , si une des lignes a a_j pour valeur de A_j , on remplace l'éventuelle variable utilisée dans l'autre ligne par A_j
- si une des lignes est (a_1, \dots, a_m) , alors la décomposition est SPI

Cette méthode est détaillée sur l'algorithme 11.4.1

Exemple 11.4.1 : Par exemple, considérons la relation $R(A, B, C, D)$ et deux dépendances sur cette relation $A \rightarrow B$ et $B \rightarrow CD$.

Si l'on considère la décomposition de R en (A, B) et (B, C, D) , à la construction du tableau, on a :

	A	B	C	D
(A, B)	a_1	a_2	$x_{1,3}$	$x_{1,4}$
(B, C, D)	$x_{2,1}$	a_2	a_3	a_4

A la première itération, on obtient (seule $B \rightarrow CD$ est utilisée) :

	A	B	C	D
(A, B)	a_1	a_2	a_3	a_4
(B, C, D)	$x_{2,1}$	a_2	a_3	a_4

On s'aperçoit que si l'on respecte les DF on obtient bien le tuple de R , donc la décomposition est SPI.

Par contre, si l'on considère la décomposition (A) et (B, C, D) , le tableau suivant ne peut plus évoluer :

	A	B	C	D
(A, B)	a_1	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$
(B, C, D)	$x_{2,1}$	a_2	a_3	a_4

La décomposition n'est pas SPI. Il suffit de considérer comme exemple les valeurs suivantes pour les tables :

3. L'autre inclusion est triviale

Algorithme 11.4.1 : Algorithme appliquant la méthode de poursuite

entrées : une décomposition (R_1, \dots, R_n) d'une relation $R(A_1, \dots, A_m)$ et un ensemble de dépendances fonctionnelles S sur R

sortie : vrai si la décomposition est SPI

```

1 construire une matrice  $T$  de dimensions  $m \times n$  où  $T_{i,j}$  représente la valeur de  $A_i$  pour
   $R_j$  ;
2 pour  $i \leftarrow 1$  to  $m$  faire
3   pour  $j \leftarrow 1$  to  $n$  faire
4     si  $A_i \in R_j$  alors  $T_{i,j} \leftarrow a_i$  ;
5     ;
6     sinon  $T_{i,j} \leftarrow x_{i,j}$  ;
7     ;
8   fin
9 fin
10 répéter
11   pour chaque  $X \rightarrow Y \in S$  faire
12     pour  $i \leftarrow 1$  to  $m$  faire
13       pour  $j \leftarrow (i + 1)$  to  $m$  faire
14         si  $T_i$  et  $T_j$  sont identiques sur les attributs de  $X$  alors
15           pour chaque  $A_k \in Y$  faire
16             si  $T_{i,k}$  ou  $T_{j,k}$  est une constante alors
17               remplacer la variable par la constante  $a_k$  ;
18             sinon
19               choisir  $x_{i,k}$  comme variable commune dans les deux lignes ;
20             fin
21           fin
22         fin
23       fin
24     fin
25   fin
26 jusqu'à  $(a_1, \dots, a_n) \in T$  ou que le tableau n'évolue plus ;
27 si  $(a_1, \dots, a_n) \in T$  alors retourner vrai ;
28 ;
29 sinon retourner faux ;
30 ;

```

11. Dépendances fonctionnelles

A	B	C	D
1	2	3	4
5	6	7	8

A
1
5

B	C	D
2	3	4
6	7	8

□

Il existe un théorème de décomposition, appelé théorème de Heath, qui propose une décomposition à jointure conservatrice, mais sans garantie de conservation des dépendances fonctionnelles⁴.

Théorème 11.4.1.

Soit $R(\Delta)$ une relation telle que $X \rightarrow Y$ soit vraie. Alors $R(\Delta) = R(X, Y) \bowtie_X R(\Delta - \{Y\})$. □

Preuve :

La démonstration du théorème est simple et laissée comme exercice. ■

Un autre théorème intéressant est le suivant :

Théorème 11.4.2.

Soit R une relation et $\{R_1, R_2\}$ une décomposition de R . Si $R_1 \cap R_2 \rightarrow R_1 - R_2$ ou $R_1 \cap R_2 \rightarrow R_2 - R_1$ est une dépendance fonctionnelle sur R , alors $\{R_1, R_2\}$ est SPI. □

Preuve :

La démonstration du théorème est simple et laissée comme exercice. ■

Enfin, on peut énoncer le théorème de « propagation » des décompositions SPI :

Théorème 11.4.3.

Soit R une relation et $\{R_1, \dots, R_n\}$ une décomposition SPI de R par rapport à un ensemble de DF S . Si $\{R_{1,1}, R_{1,2}\}$ est une décomposition SPI de R_1 par rapport à F , alors $\{R_{1,1}, R_{1,2}, R_2, \dots, R_n\}$ est une décomposition SPI de R par rapport à F . □

Preuve :

La démonstration du théorème est simple et laissée comme exercice. ■

Décomposition SPD

Pour vérifier que la décomposition vérifie les dépendances fonctionnelles initiales, il suffirait de construire la jointure des relations de la décomposition et de vérifier que les dépendances sont vérifiées sur la jointure. La construction d'une jointure étant très coûteuse, ne pourrait-on pas vérifier la cohérence locale (i.e. au niveau de chaque relation) des dépendances pour conclure à la vérification des dépendances fonctionnelles ?

Définition 11.4.1 (décomposition SPD).

Une décomposition R est sans perte de dépendances par rapport à un ensemble de dépendances fonctionnelles S s'il existe $F \subseteq S^+$ tel que :

— pour toute $X \rightarrow Y \in F$ il existe $R_i \in R$ telle que $XY \subseteq R_i$

4. On note dans toute la suite du chapitre $R(X, Y) = \pi_{X, Y} R$.

— $F^+ = S^+$ □

Soit S un ensemble de dépendances fonctionnelles sur une relation R . Soit $R_i \subseteq R$, on note $S_{R_i} = \{X \rightarrow Y \in S^+ : XY \subseteq R_i\}$ (c'est l'ensemble des dépendances fonctionnelles de S projetables sur R_i).

Théorème 11.4.4.

Soit $\{R_1, \dots, R_n\}$ une décomposition de R par rapport à S ensemble de dépendances fonctionnelles sur R , alors $\{R_1, \dots, R_n\}$ est sans perte de dépendances par rapport à S ssi $(\bigcup_{i=1}^n S_{R_i})^+ = F^+$. □

Preuve :

La démonstration est aisée en remplaçant dans la définition d'une décomposition SPD F par $\bigcup_{i=1}^n S_{R_i}$. ■

Si l'on veut appliquer ce théorème, on doit calculer S^+ , ce qui peut être très coûteux (exponentiel par rapport au nombre d'attribut dans S). On peut par contre utiliser l'algorithme 11.4.2 pour vérifier si une décomposition est SPD.

Algorithme 11.4.2 : Algorithme SPD

entrées : une décomposition $\{R_1, \dots, R_n\}$ d'une relation $R(A_1, \dots, A_m)$ et un ensemble de dépendances fonctionnelles S sur R

sortie : vrai si la décomposition est SPD

```

1 spd ← vrai ;
2 tant que spd et il existe  $X \rightarrow Y \in S$  non traitée faire
3   |  $Z \leftarrow X$  ;
4   | répéter
5   |   | pour  $i \leftarrow 1$  to  $n$  faire  $Z \leftarrow Z \cup ((Z \cap R_i)^+ \cap R_i)$ ;
6   |   | ;
7   | jusqu'à  $Y \subseteq Z$  ou  $Z$  ne change pas ;
8   | si  $Y \not\subseteq Z$  alors spd ← faux ;
9   | ;
10 fin
11 retourner spd ;
```

Théorème 11.4.5.

L'algorithme 11.4.2 renvoie vrai si la décomposition donnée en entrée est SPD, faux sinon. □

Nous allons maintenant voir que la décomposition sous forme normale permet de garantir les propriétés demandées. La plupart de ces formes normales ont été formalisées par Codd.

11.4.2. Première et seconde forme normale

La première forme normale est très simple : elle stipule simplement que les domaines des attributs doivent être atomiques (on ne peut pas les décomposer en sous-relations comme un attribut « adresse » par exemple).

Définition 11.4.2.

Soit R une relation d'attributs A_1, \dots, A_n . R est sous première forme normale (ou 1NF) ssi pour tout $i \in \{1, \dots, n\}$ les A_i a un domaine atomique. \square

La seconde forme normale permet de stipuler qu'une relation est telle que les attributs de la relation n'appartenant pas à une clé dépendent fonctionnellement des clés de la relation.

Définition 11.4.3.

Soit R une relation. R est sous seconde forme normale (ou 2NF) ssi R est sous première forme normale et si tout attribut B n'appartenant pas à une clé de la relation est en dépendance élémentaire avec toutes les clés de la relation. \square

La seconde forme normale impose donc la non existence de dépendances *partielles* vers les clés de la relation. On peut aussi dire qu'un attribut n'appartenant pas à une clé de la relation ne peut pas être en dépendance élémentaire avec une partie d'une clé de la relation.

L'exemple typique de relation qui n'est pas sous deuxième forme normale est une relation gérant les DVD empruntés par un client. La relation *Location* comporte les attributs *NumClient*, *NomClient*, *AdresseClient*, *NumDVD*, *DateEmprunt*. On voit que l'on a les dépendances suivantes : $NumClient \rightarrow NomClient AdresseClient$, $NumClient NumDVD \rightarrow DateEmprunt$. ($NumClient, NumDVD$) est la clé de la relation (on suppose donc qu'un client n'emprunte qu'une seule fois un DVD...). Or $NumClient \rightarrow NomClient AdresseClient$, donc la relation n'est pas en 2NF, car on n'a pas $NumClient NumDVD \rightarrow NomClient AdresseClient$. On peut remarquer en effet que l'adresse et le nom du client seront dupliqués à chaque location.

11.4.3. Troisième forme normale

Définition 11.4.4.

Soit R une relation. R est sous troisième forme normale (ou 3NF) ssi R est sous seconde forme normale et si tout attribut B n'appartenant pas à une clé de la relation ne dépend pas d'un attribut non clé. \square

Par exemple, la relation *Type Ordinateur* n'est pas 3NF : sa seule clé est *type*, mais la dépendance $cons \rightarrow pays$ existe.

Théorème 11.4.6.

Toute relation admet une décomposition en 3NF avec jointure conservatrice et préservation des dépendances fonctionnelles. \square

L'algorithme 11.4.3 dit de *normalisation*, appelé également algorithme de *synthèse* ou algorithme d'*Armstrong*, permet de décomposer un schéma relationnel en sous-schéma 3NF⁵.

Algorithme 11.4.3 : Synthèse d'une relation R avec un ensemble S de dépendances fonctionnelles associé.

entrées : une relation R et un ensemble S de DF sur R

sortie : un ensemble R_i de relations décomposant R sous forme 3NF avec jointure conservatrice et préservation des DF

```

1  $S' \leftarrow \text{CouvIrr}(S)$  ;
2 Partitionner  $S'$  en sous-ensembles  $S'_1, \dots, S'_n$  tels que pour tout  $i \in \{1, \dots, n\}$ , les
   dépendances fonctionnelles de  $S'_i$  ait la même partie gauche ;
3 pour chaque  $i \in \{1, \dots, n\}$  faire
4   | Construire la restriction  $R_i$  de  $R$  aux attributs contenus dans  $S'_i$  ;
5 fin
6 si il existe  $i \in \{1, \dots, n\}$   $R_i$  tq  $R_i$  contient une clé de  $R$  alors
7   | retourner  $\{R_1, \dots, R_n\}$  ;
8 sinon
9   | construire une relation  $R_c$  avec une des clés de  $R$  ;
10  | retourner  $\{R_1, \dots, R_n, R_c\}$  ;
11 fin

```

Exemple 11.4.2 : Reprenons la relation *Ordinateur* dont la clé est l'attribut *type*. On voit clairement que l'on a une dépendance fonctionnelle $cons \rightarrow pays$ en plus des dépendances « classiques » concernant la clé *type*.

Dans ce cas, on peut séparer la relation en deux sous-relations via l'algorithme précédent :

- une relation *Constructeur* $\{cons, pays\}$
- une relation *Type Ordi* $\{type, cons, rev, nb_postes, mem_max, uc, capa_lec, capa_disk\}$

□

11.4.4. Troisième forme normale de Boyd-Codd

La normalisation en 3NF permet de construire une relation à jointure conservatrice et préservant les dépendances fonctionnelles initiales. La plupart des anomalies sont éliminées, mais certaines peuvent encore persister. En effet, rien n'empêche dans un schéma sous 3NF qu'il existe une dépendance d'un attribut non clé vers un attribut clé. Dans ce cas, des problèmes de mise à jour peuvent encore survenir.

Par exemple, supposons que dans la relation *Type Ordi* nous ayons les dépendances fonctionnelles suivantes :

5. On suppose que *CouvIrr* est une fonction permettant de calculer la couverture irredondante d'un ensemble de DF.

11. Dépendances fonctionnelles

- *type* est la clé qui détermine les autres attributs ;
- $rev \longrightarrow type$ qui signifie qu'un revendeur ne peut vendre qu'un seul type d'ordinateur (on remarquera que l'instance proposée dans la section 11.1 viole cette dépendance).

Dans ce cas, la relation est bien 3NF : il n'y a pas d'attribut non clé dépendant d'attribut non clé. Par contre, la deuxième dépendance entre un attribut non clé et un attribut clé peut induire les anomalies suivantes :

- il est impossible d'enregistrer un revendeur sans modèle associé ;
- la suppression d'un tuple peut entraîner la suppression d'un revendeur.

Définition 11.4.5.

Soit R une relation. R est en troisième forme normale de Boyce-Codd (ou BCNF) si et seulement si R est 3NF et que les seules dépendances élémentaires sont de la forme $A_1 \dots A_n \longrightarrow B$ où $\{A_1, \dots, A_n\}$ est une superclé. \square

Théorème 11.4.7.

Toute relation R admet une décomposition en BCNF. La préservation des dépendances fonctionnelles initiales n'est pas garantie. \square

Toute relation en BCNF est également en 3NF, mais on ne garantit pas que les dépendances fonctionnelles initiales puissent être vérifiées par la jointure des sous-relations construites par décomposition.

Nous présentons maintenant un algorithme de décomposition en BCNF.

Algorithme 11.4.4 : Décomposition BCNF d'une relation R avec un ensemble S de dépendances fonctionnelles associé.

entrées : une relation R et un ensemble S de DF sur R

sortie : un ensemble R_i de relations décomposant R sous forme BCNF

```
1 resultat  $\leftarrow$   $\{R\}$  ;
2 tant que il existe un schéma  $R_i \in$  resultat qui n'est pas BCNF faire
3   | chercher une dépendance non triviale  $X \longrightarrow Y$  dans  $R_i$  tel que  $X$  ne soit pas une
4   | clé ;
5   | resultat  $\leftarrow$  resultat  $- \{R_i\} \cup \{R_i - \{Y\}\} \cup \{X, Y\}$  ;
6   | fin
7   | pour chaque  $R_i(Att_i)$  et  $R_j(Att_j)$  tq  $Att_i \subseteq Att_j$  faire
8   |   | resultat  $\leftarrow$  resultat  $- \{R_i\}$  ;
9   |   | fin
10  retourner resultat ;
```

Attention, cet algorithme ne garantit pas la préservation des dépendances fonctionnelles initiales. De plus, le résultat obtenu dépend de l'ordre dans lequel on a effectué les décompositions.

Exemple 11.4.3 : Reprenons la relation *Type Ordinateur* de clé *type* et possédant la dépendance fonctionnelle supplémentaire $rev \longrightarrow type$. Dans ce cas, l'algorithme précédent va décomposer la relation en deux sous-relations :

- *Revendeur* {*rev, type*}
 - une relation *Type Ordi* {*rev, cons, pays, nb_postes, mem_max, uc, capa_lec, capa_disk*}
-

11.4.5. Mise en œuvre

Pour mettre en œuvre ces algorithmes de normalisation, on part souvent d'une seule relation contenant tous les attributs que l'on veut manipuler et des dépendances fonctionnelles que l'on aura trouvées par analyse du cahier des charges. On appelle cette relation *relation universelle*.

La plupart du temps, si l'on veut normaliser « proprement » une relation universelle, on va la mettre sous forme 3NF et ainsi obtenir un certain nombre de relations. On va ensuite essayer de mettre chacune de ces relations en BCNF, en vérifiant bien après mise sous forme BCNF que les dépendances fonctionnelles initiales sont encore respectées dans les nouvelles relations.

11.5. Dépendances multivaluées et normalisation associée

11.5.1. Définition

Les anomalies de mise à jour et de redondance dans une base de données ne proviennent pas toujours de dépendances fonctionnelles. Prenons un exemple : supposons que l'on ait une relation *sport-déplace*: {*nom, sport, moyen*} qui représente les liens entre une personne, les sports qu'elle pratique et ses moyens de transport habituellement utilisés. On voit qu'une personne peut utiliser plusieurs moyens de transport et pratiquer plusieurs sports. Il n'y a aucune dépendance fonctionnelle dans cette relation, l'ensemble des attributs formant la clé de *sport-déplace*. Or, si une personne pratique plusieurs sports et utilise plusieurs moyens de transport, des redondances apparaissent :

<i>nom</i>	<i>sport</i>	<i>moyen</i>
Dupont	football	bus
Dupont	football	vélo
Dupont	judo	bus
Dupont	judo	vélo

La dépendance ici n'est plus *fonctionnelle* : c'est un ensemble de sports par exemple qui dépend du nom. On parle alors de dépendance multivaluée.

Définition 11.5.1.

Soit $R(\Delta)$ une relation d'ensemble d'attributs Δ et X, Y, Z une partition de Δ . On dit qu'il existe une dépendance multivaluée entre X et Y ssi toute valeur de X détermine un ensemble unique de valeurs de Y indépendamment des valeurs de Z .

On le note $X \twoheadrightarrow Y$.

□

Dans l'exemple précédent, on a par exemple $nom \twoheadrightarrow sport$: un individu peut pratiquer plusieurs sports et ces sports ne dépendent pas des moyens de transport qu'il utilise.

11. Dépendances fonctionnelles

On peut également donner une autre définition des dépendances multivaluées.

Définition 11.5.2.

Soit $R(\Delta)$ une relation d'ensemble d'attributs Δ et X, Y, Z une partition de Δ . On dit qu'il existe une dépendance multivaluée entre X et Y ssi si (x, y, z) et (x, y', z') sont des tuples de R , alors (x, y', z) et (x, y, z') le sont aussi. \square

11.5.2. Propriétés des dépendances multivaluées

Proposition 11.5.1.

Les dépendances fonctionnelles sont des cas particuliers de dépendances multivaluées : si $A \rightarrow B$ alors $A \twoheadrightarrow B$. \square

Proposition 11.5.2.

Les dépendances multivaluées triviales sont des dépendances multivaluées de la forme suivante :

$$X \twoheadrightarrow Y \text{ où } Y \subseteq X. \quad \square$$

On retrouve la même définition que pour les dépendances fonctionnelles.

Proposition 11.5.3.

Les dépendances multivaluées sont complémentaires : si $R(\Delta)$ est partitionnée en $R(X, Y, Z)$, alors si $X \twoheadrightarrow Y$, alors $X \twoheadrightarrow Z$. \square

11.5.3. Normalisation des dépendances multivaluées

Nous allons présenter dans cette section une nouvelle forme normale, la quatrième forme normale, qui permet de normaliser les relations possédant des dépendances multivaluées. Dans un premier temps, nous énonçons comme pour les dépendances fonctionnelles un théorème de décomposition, appelé théorème de Fagin.

Théorème 11.5.1.

Soit $R(X, Y, Z)$ une relation telle que $X \twoheadrightarrow Y$. Alors $R(X, Y, Z) = R(X, Y) \bowtie_X R(X, Z)$. Cette décomposition est sans perte d'information. \square

La quatrième forme normale est une forme normale découlant de la BCNF :

Définition 11.5.3.

Soit R une relation. R est dite en quatrième forme normale ou 4NF si et si seulement si toutes les dépendances multivaluées non triviales de R de la forme $X \twoheadrightarrow Y$ sont telles que X est un superclé de R . \square

On voit tout de suite qu'une relation en 4NF est également en BCNF (car toute dépendance fonctionnelle est multivaluée).

L'algorithme présenté dans la section 11.4.4 permettant de mettre sous BCNF une relation s'applique encore ici : on peut l'utiliser pour mettre une relation en 4NF.

Exemple 11.5.1 : Reprenons la relation *sport-déplace*. On a ici une première dépendance $nom \twoheadrightarrow sport$ qui l'on va utiliser dans l'algorithme. On obtient alors deux relations :

- *sport* : {*nom*, *sport*}
- *deplace* : {*nom*, *moyen*}

Ces deux relations sont correctement normalisées. □

11.6. Conclusion

Nous avons introduit dans ce chapitre les notions de dépendances fonctionnelles et multivaluées et montrer que ces dépendances pouvaient entraîner des anomalies de mises à jour des relations concernées. Nous avons ensuite introduit différentes formes normales permettant de résoudre ces problèmes. Ce chapitre n'est qu'une introduction à la normalisation de relations. Il existe en effet d'autres formes normales (5NF, ONF, DKNF), d'autres dépendances (hiérarchiques, produit). On pourra se référer à [27, 18, 14].

On pourra toutefois remarquer la normalisation des relations n'est pas appliquée systématiquement. En effet, la normalisation repose sur la décomposition de relations en sous-relations. Les requêtes sur les relations vont donc induire des jointures qui sont toujours pénalisantes en termes de performances. On trouvera donc souvent des relations qui sont intentionnellement non normalisées pour garantir un certain niveau de performance. Dans ce cas, il faut bien être conscient des conséquences que cela implique sur la redondance des informations et des éventuelles anomalies de mises à jour.

12. Gestion des transactions

Nous allons nous intéresser dans ce chapitre à deux aspects importants des SGBDs :

- la gestion des pannes du système. Ces pannes peuvent provenir par exemple d'un problème réseau, d'un disque qui ne fonctionne plus, d'une panne de courant etc. Le but d'un SGBD est de garantir la *résilience* du système, i.e. de garantir que l'on pourra retrouver une base de données dans un état cohérent après redémarrage.
- la gestion de la concurrence dans un environnement multi-utilisateurs. Les données de la base ne doivent pas être corrompues (au sens des contraintes d'intégrité de la base) parce que des opérations légales sont effectuées en même temps.

Tous ces concepts sont indépendants du modèle de données utilisé dans le SGBD. Nous allons dans un premier temps présenter la notion de transaction et les propriétés attendues d'une transaction. Nous nous intéresserons ensuite au problème de la reprise d'un SGBD après une panne. Nous aborderons ensuite le problème de la gestion de la concurrence dans un environnement multi-utilisateurs. Nous présenterons des notions importantes : *lock* à deux phases, sérialisabilité etc. Enfin, nous présenterons les solutions apportées par SQL pour la gestion des transactions.

12.1. Notion de transaction

12.1.1. Définition

Pour l'instant, nous n'avons utilisé que l'interpréteur SQL. Dans ce cas, une opération sur la base de données est vue comme atomique : on ne peut pas effectuer d'autre opération avant que celle-ci ne soit complétée. Supposons que nous ayons une relation de schéma `Compte(no_compte, no_client, solde)` qui représente un compte bancaire par son numéro, le numéro de son client et son solde. Considérons la séquence d'opérations suivantes qui réalise un transfert de 100€ du compte C1 vers le compte C2 :

```
UPDATE Compte SET solde = solde - 100 WHERE no_compte = 'C1';  
UPDATE Compte SET solde = solde + 100 WHERE no_compte = 'C2';
```

Dans cet exemple, ce qui est considéré comme une opération atomique (un transfert de fonds de compte à compte) est en fait constitué de deux opérations de mise à jour de la base de données. Dans ce cas, que se passe-t-il si pour une raison quelconque la seconde transaction n'est pas effectuée ? On se retrouve alors avec une base de données dans un état incohérent. Il faut considérer ces deux mises à jour élémentaires comme étant un *tout*. On dira alors que ces deux opérations forment une *transaction*.

Définition 12.1.1.

Une transaction est une unité logique de travail. C'est une séquence d'opérations élémentaires qui est vue comme atomique d'un point de vue externe au SGBD. □

Remarquons tout de suite que les opérations élémentaires de la base de données peuvent amener la base à un état incohérent (après la première mise à jour, les comptes ne sont pas équilibrés). C'est la transaction dans son ensemble qui, d'un point de vue externe, va amener la base dans un état cohérent.

Le but de la gestion des transactions est de garantir qu'une transaction s'exécute dans son ensemble ou ne s'exécute pas du tout (même si certaines de ses opérations élémentaires se sont déjà exécutées). Elle sera donc vue comme *indivisible* d'un point de vue externe.

Enfin, on peut voir les opérations élémentaires que nous avons manipulées jusqu'à présent comme des transactions élémentaires (séquence d'une seule opération). La propriété précédente s'applique bien à ce cas particulier : une opération élémentaire de mise à jour est soit effectuée dans sa totalité par le SGBD, soit n'est pas effectuée (même dans le cas de mises à jour complexes).

12.1.2. Propriétés des transactions : les propriétés ACID

Nous venons de voir une première propriété des transactions : elles doivent être considérées comme étant indivisibles. Il existe d'autres bonnes propriétés que l'on peut annoncer sur les transactions : par exemple, que doit-il se passer lorsque plusieurs transactions se déroulent dans un environnement multi-utilisateurs ? Ces propriétés sont appelées propriétés ACID.

Proposition 12.1.1.

Des transactions ayant un « bon » comportement respectent les propriétés suivantes, dites propriétés ACID :

- « *A* » pour atomicité : *la transaction doit s'effectuer dans sa totalité ou ne pas s'effectuer du tout ;*
- « *C* » pour cohérence : *une transaction préserve la cohérence de la base de données. Elle amène la base d'un état cohérent vers un autre état cohérent.*
- « *I* » pour isolation : *une transaction doit s'effectuer comme si aucune autre transaction ne s'effectuait en même temps.*
- « *D* » pour durabilité : *les effets d'une transaction sur la base de données ne doivent pas être perdus une fois que la transaction s'est effectuée.* □

Comment alors garantir ces différentes propriétés ? Nous allons voir que la propriété de durabilité va être garantie par la possibilité de reprise après panne, l'atomicité et l'isolation sont garanties par le *scheduler* qui gère la concurrence entre transactions. Reste le problème de la cohérence, qui est en fait vérifiée par l'établissement de contraintes sur la base de données (voir le chapitre 9 et en particulier le retardement de vérification de contraintes à la section 9.1.4).

12.1.3. Opérations de base du gestionnaire de transactions

Un gestionnaire de transactions possède deux opérations de base : le *commit* et le *rollback*. Nous reviendrons sur ces termes plus en détail dans ce qui suit. Nous présentons rapidement ici ces deux opérations car elles vont nous servir dans tout ce qui suit.

L'opération COMMIT signale une fin de transaction réussie. La base de données doit alors être dans un état cohérent. Toutes les mises à jour effectuées par la transaction peuvent être validées et effectuées de façon permanente.

L'opération ROLLBACK signale l'abandon d'une transaction. Toutes les mises à jour effectuées par la transaction doivent être annulées pour revenir au point de départ de la transaction.

Exemple 12.1.1 : Par exemple, la transaction présentée en section 12.1.1 pourrait être représentée de la façon suivante :

```
BEGIN TRANSACTION
  UPDATE Compte SET solde = solde - 100 WHERE no_compte = 'C1';
  IF erreur_quelconque THEN GOTO UNDO;
  UPDATE Compte SET solde = solde + 100 WHERE no_compte = 'C2';
  IF erreur_quelconque THEN GOTO UNDO;
  COMMIT;
  GOTO FINISH;

UNDO:
  ROLLBACK;

FINISH:
END TRANSACTION
```

12.2. Reprise après panne

Nous allons nous intéresser dans cette section à la reprise après panne d'un système. Une des propriétés ACID des transactions est la durabilité. Il faut s'assurer que les changements effectués par une transaction seront effectivement écrits durablement sur le support de stockage secondaire comme par exemple un disque dur. Un point essentiel pour la résolution de ce problème est le fait que les transactions travaillent sur une image de la base de données en mémoire centrale (en utilisant les *buffers* mémoire). Le *buffer manager* a pour rôle de gérer l'écriture des données en mémoire centrale sur le disque. Ces écritures ne peuvent normalement pas se faire à chaque opération élémentaire (en particulier pour des problèmes de performance). Le problème va se poser principalement si le système doit être redémarré : les données en mémoire centrale disparaissent. Il faut donc savoir quand les modifications ont été réellement inscrites sur le disque contenant la base de données pour pouvoir reconstruire la base de données dans un état cohérent. Le problème de la durabilité des informations contenues dans la base de données est donc

12. Gestion des transactions

intimement lié à celui de la reprise après panne du système. Cette possibilité de reprise après panne va être assurée par un journal qui va contenir l'ensemble des mises à jour effectuées sur le système.

Dans ce qui suit, pour étoffer les explications, on considérera que l'on a disposition quatre opérations qui représente les actions possibles du *transaction manager* :

- **INPUT(X)** : l'élément X de la base de donnée est amené du disque en mémoire principale ;
- **READ(X)** : lire la valeur de X en mémoire centrale (dans un *buffer*). Si besoin est, on utilise **INPUT** pour aller tout d'abord la chercher sur le disque ;
- **WRITE(X, t)** : on affecte t à la valeur de X en mémoire centrale. Là encore, si besoin est on utilise **INPUT** pour récupérer X en mémoire principale ;
- **OUTPUT(X)** : la valeur de X est inscrite sur le disque dur.

On peut résumer ces différentes opérations sur le schéma 12.1.

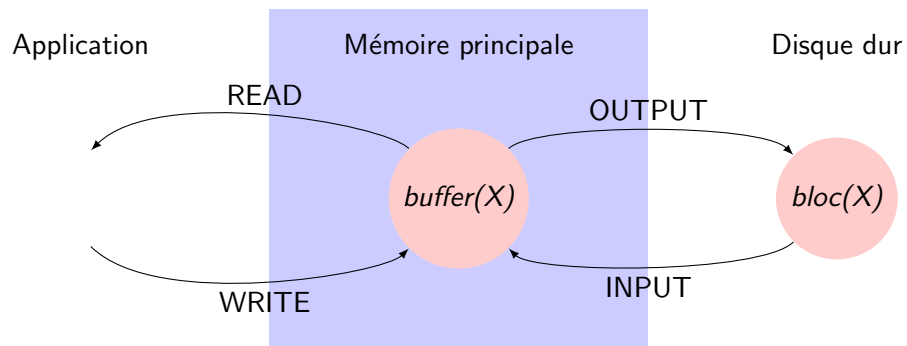


FIGURE 12.1. – Opérations de base disponibles

On remarquera que nous prenons ici des hypothèses très simplificatrices. Par exemple, lorsque l'on lit ou écrit des éléments du disque, on ne peut utiliser que des blocs du disque (cf. chapitre 3).

Par exemple, si l'on reprend la transaction présentée en section 12.1.3 (transfert de 100 euros d'un compte à un autre), on pourrait utiliser les opérations de lecture/écriture précédentes de la façon suivante (MemC1 représente la valeur du compte C1 en mémoire et DC1 la valeur du compte C1 sur le disque dur) :

Étape	Action	t	MemC1	MemC2	DC1	DC2
0		0	1000	500	1000	500
1	t := READ(C1)	1000	1000	500	1000	500
2	t := t - 100	900	1000	500	1000	500
3	WRITE(C1, t)	900	900	500	1000	500
4	t := READ(C2)	500	900	500	1000	500
5	t := t + 100	600	900	500	1000	500
6	WRITE(C2, t)	600	900	600	1000	500
7	OUTPUT(C1)	600	900	600	900	500
8	OUTPUT(C2)	600	900	600	900	600

On aurait très bien pu utiliser les opérations de la façon suivante :

Étape	Action	t	MemC1	MemC2	DC1	DC2
0		0	1000	500	1000	500
1	t := READ(C1)	1000	1000	500	1000	500
2	t := t - 100	900	1000	500	1000	500
3	WRITE(C1,t)	900	900	500	1000	500
4	OUTPUT(C1)	600	900	600	900	500
5	t := READ(C2)	500	900	500	1000	500
6	t := t + 100	600	900	500	1000	500
7	WRITE(C2,t)	600	900	600	1000	500
8	OUTPUT(C2)	600	900	600	900	600

Dans les deux cas, que se passe-t-il si le système s'arrête avant l'exécution de OUTPUT(C2) ? L'état de la base sur le disque est incohérent, même si dans le premier cas l'état de la base était cohérent en mémoire principale.

C'est le rôle du *transaction manager* et du *logging manager* de gérer correctement ces entrées/sorties.

12.2.1. Journal

Pour pouvoir conserver l'ensemble des modifications et des opérations importantes qui ont été effectuées sur la base de données, on utilise un journal (*log* en anglais). Ce journal peut contenir différentes entrées :

- <START T> : la transaction T a démarré ;
- <COMMIT T> : la transaction T a réussi. Tous les changements effectués par T doivent être inscrits sur le disque ¹ ;
- <ABORT T> : la transaction n'a pas réussi. Dans ce cas, aucun de ses changements ne doit apparaître sur le disque.
- des entrées du type <T, X, v> : la transaction T a changé la valeur de l'élément X et l'ancienne valeur ou la nouvelle valeur (suivant le type de journalisation que l'on choisit) de X était v. Ce changement concerne le changement en mémoire centrale et non pas sur le disque comme nous le verrons plus tard.

Le système dispose également d'une opération FLUSH LOG qui lui permet de forcer l'écriture du journal sur le disque.

12.2.2. Un exemple simple de reprise : l'undo-logging

Nous allons présenter les principes de reprise après panne à travers un premier exemple simple, l'*undo-logging*. L'idée intuitive de ce type de reprise est d'effacer les changements qui ont été faits par une transaction qui ne s'est pas déroulée correctement (interrompue par une panne ou par un ROLLBACK).

1. Ce point peut poser de gros problèmes. En effet, nous ne sommes pas sûr que les changements ont été effectivement inscrits sur le disque (c'est le rôle du *buffer manager* de s'occuper de cela. Il faut donc peut-être forcer l'écriture des mises à jour.)

12. Gestion des transactions

Les principes sur lesquels repose l'*undo-logging* sont les suivants :

Principe 12.2.1.

1. si une transaction T modifie un élément X , alors on doit écrire $\langle T, X, v \rangle$ dans le journal sur le disque avant que la nouvelle valeur de X ne soit écrite sur le disque. Cette entrée signifie : l'ancienne valeur de X était v et X a été modifié en mémoire principale.
2. si une transaction T est réussie, alors on écrit $\langle \text{COMMIT } T \rangle$ après que tous les changements effectués par la transaction ont été écrits sur le disque. □

Exemple 12.2.1 : Par exemple, un journal correspondant à la transaction présentée dans la section 12.1.3 sera construit de la façon suivante si cette transaction s'effectue correctement :

Étape	Action	t	MemC1	MemC2	DC1	DC2	Journal
0		0	1000	500	1000	500	$\langle \text{START } T \rangle$
1	$t := \text{READ}(C1)$	1000	1000	500	1000	500	
2	$t := t - 100$	900	1000	500	1000	500	
3	$\text{WRITE}(C1, t)$	900	900	500	1000	500	$\langle T, C1, 1000 \rangle$
4	$t := \text{READ}(C2)$	500	900	500	1000	500	
5	$t := t + 100$	600	900	500	1000	500	
6	$\text{WRITE}(C2, t)$	600	900	600	1000	500	$\langle T, C2, 500 \rangle$ □
7	FLUSH LOG	600	900	600	1000	500	
8	OUTPUT(C1)	600	900	600	900	500	
9	OUTPUT(C2)	600	900	600	900	600	
10		600	900	600	900	600	$\langle \text{COMMIT } T \rangle$
11	FLUSH LOG	600	900	600	900	600	

Remarquons que les écritures du journal sur le disque doivent être forcées par la *log manager* via l'instruction `FLUSH LOG` : on ne peut évidemment pas les laisser en mémoire centrale. De plus, si l'on suit les règles énoncées précédemment, on est obligé d'utiliser `FLUSH LOG` avant d'écrire effectivement les nouvelles valeurs de $C1$ et de $C2$ sur le disque.

12.2.3. Reprise avec undo-logging

Supposons maintenant que nous disposions d'un journal comme celui présenté précédemment. Supposons également qu'une panne du système se produise. Comment, au redémarrage du système, garantir que la base de données va être dans un état cohérent ?

Nous devons examiner le journal en entier. On part de la fin du journal et on lit les entrées en « remontant ». Notons tout de suite que si on trouve une entrée du type $\langle \text{COMMIT } T \rangle$, alors on est sûr que les données modifiées par la transaction T ont été inscrites sur le disque. T ne peut pas avoir laissé la base de données dans un état incohérent.

Supposons maintenant que nous rencontrions une entrée $\langle T, X, v \rangle$ et que nous n'avons pas rencontré d'entrée $\langle \text{COMMIT } T \rangle$. Dans ce cas, la transaction T est incomplète et on doit la déjouer. Malheureusement, on ne sait pas quelles sont les modifications qui ont

été écrites sur le disque. On utilise alors les entrées $\langle T, X, v \rangle$ pour remettre la base dans un état cohérent. Ce principe s'applique également si on a rencontré une entrée $\langle \text{ABORT } T \rangle$ dans le journal, car la transaction a été annulée et on peut retrouver grâce aux entrées $\langle T, X, v \rangle$ comment être sûr de l'annuler.

Exemple 12.2.2 : Sur l'exemple précédent, supposons que nous ayons le journal suivant :

```
<START T>
<T,C1,1000>
<T,C2,500>
```

Dans ce cas, en remontant le journal, on sait que la transaction T ne s'est pas terminée (par exemple parce que `OUTPUT(C2)` ne s'est pas effectuée). On peut donc annuler la transaction en utilisant les entrées du journal et en prévenant bien sûr l'utilisateur que T a été annulée. \square

12.2.4. Notion de point de contrôle

Le principe précédent nous oblige à parcourir le journal dans sa totalité, ce qui peut être extrêmement long. En effet, le fait de trouver une entrée $\langle \text{COMMIT } T \rangle$ ne garantit pas que toutes les transactions ont été complètement effectuées ou annulées (les transactions peuvent s'exécuter en parallèle). Il se peut qu'une transaction ait été démarrée précédemment et ne se soit pas terminée.

La solution la plus simple pour pallier ce problème est d'écrire un *point de contrôle* dans le journal : périodiquement, on arrête d'accepter les transactions, on attend que toutes les transactions en cours se finissent (et on écrit un `COMMIT` ou un `ABORT` dans le journal suivant le cas), on marque le journal avec un point de contrôle et on recommence à accepter les transactions. Lors d'une reprise, on n'est alors plus obligé de relire tout le journal, mais simplement la partie suivant le dernier point de contrôle.

Il est quand même gênant d'arrêter d'accepter des transactions pendant un certain temps. Les transactions en cours peuvent prendre du temps pour se finir et le système est bloqué pendant ce laps de temps. Une technique plus complexe de point de contrôle est donc souvent mise en place et se déroule de la façon suivante :

1. écrire une entrée $\langle \text{START CHKPT}(T_1, \dots, T_n) \rangle$ dans le journal. T_1, \dots, T_n sont toutes les transactions actives au moment de l'écriture.
2. attendre que T_1, \dots, T_n se finissent. Continuer à accepter des transactions.
3. quand T_1, \dots, T_n ont fini, écrire $\langle \text{END CHKPT} \rangle$ dans le journal.

Dans ce cas, lors de la reprise du système, pour reconstruire un état cohérent de la base, on parcourt le journal depuis la fin :

- soit on rencontre d'abord une entrée $\langle \text{END CHKPT} \rangle$ et dans ce cas, on sait que toutes les transactions incomplètes ont démarré après l'entrée $\langle \text{START CHKPT}(T_1, \dots, T_n) \rangle$ précédente. Les entrées du journal précédant $\langle \text{START CHKPT}(T_1, \dots, T_n) \rangle$ sont donc inutiles.

12. Gestion des transactions

- soit on rencontre d'abord une entrée $\langle \text{START CHKPT}(T_1, \dots, T_n) \rangle$ et dans ce cas, le système est tombé en panne durant le point de contrôle. Il suffit donc de remonter le journal jusqu'à la transaction T_i la plus ancienne.

12.2.5. Journalisation par undo/redo

Nous avons vu précédemment le principe de la journalisation par *undo*. Il existe le même principe par *redo* qui va permettre de rejouer les transactions qui se sont correctement déroulées (on écrit alors dans le journal des entrées du type $\langle T, X, v \rangle$ où v est la nouvelle valeur prise par X). Les principes régissant une journalisation par *redo* sont les suivants :

Principe 12.2.2.

1. si une transaction T modifie un élément X , alors on doit écrire $\langle T, X, v \rangle$ dans le journal sur le disque avant que la nouvelle valeur de X ne soit écrite sur le disque. Cette entrée signifie : X a été modifié en mémoire principale et la nouvelle valeur de X est v .
2. si une transaction T est réussie, alors on écrit $\langle \text{COMMIT } T \rangle$ avant de commencer à écrire les changements effectués par la transaction sur le disque. □

Exemple 12.2.3 : Sur l'exemple précédent, on obtiendrait le journal par *redo* suivant :

Etape	Action	t	MemC1	MemC2	DC1	DC2	Journal
0		0	1000	500	1000	500	$\langle \text{START } T \rangle$
1	t := READ(C1)	1000	1000	500	1000	500	
2	t := t - 100	900	1000	500	1000	500	
3	WRITE(C1, t)	900	900	500	1000	500	$\langle T, C1, 900 \rangle$
4	t := READ(C2)	500	900	500	1000	500	
5	t := t + 100	600	900	500	1000	500	
6	WRITE(C2, t)	600	900	600	1000	500	$\langle T, C2, 600 \rangle$
7		600	900	600	1000	500	$\langle \text{COMMIT } T \rangle$
8	FLUSH LOG	600	900	600	1000	500	
9	OUTPUT(C1)	600	900	600	900	500	
10	OUTPUT(C2)	600	900	600	900	600	

On vérifiera que l'on peut faire une reprise après panne en prenant le principe inverse de l'*undo-logging*, i.e. en partant du début du journal et en rejouant toutes les transactions qui se sont correctement déroulées. □

Ces deux systèmes de reprise possèdent pourtant des défauts :

- le système de reprise par *undo* oblige le système à écrire les changements effectués par une transaction immédiatement, ce qui peut augmenter le nombre d'entrées/sorties et pénaliser le système au niveau performance ;
- d'un autre côté, le système de reprise par *redo* oblige le système à conserver beaucoup d'informations en mémoire centrale puisqu'il faut attendre que la transaction soit finie pour écrire les données sur le disque.

La plupart du temps, on utilise donc un système de reprise après panne qui combine les deux techniques. On écrira donc dans le journal des entrées du type $\langle T, X, v, w \rangle$ pour signifier que la transaction T change la valeur de l'élément X , que l'ancienne valeur de X est v et que sa nouvelle valeur est w .

Dans ce cas, la règle imposée sur l'écriture du journal est la suivante.

Principe 12.2.3.

Dans une procédure de journalisation par *undo/redo*, avant de modifier un élément X de la base de données sur le disque, il est nécessaire d'écrire une entrée $\langle T, X, v, w \rangle$ sur le disque. \square

Remarquons tout de suite que le journal sera plus volumineux que dans le cas de l'*undo logging* ou du *redo logging*. De plus, l'entrée de *commit* $\langle \text{COMMIT } T \rangle$ peut être écrite indifféremment après ou avant l'écriture effective des modifications sur le disque.

Exemple 12.2.4 : Toujours sur l'exemple précédent, voici un exemple de journal produit par *undo/redo* :

Etape	Action	t	MemC1	MemC2	DC1	DC2	Journal
0		0	1000	500	1000	500	$\langle \text{START } T \rangle$
1	$t := \text{READ}(C1)$	1000	1000	500	1000	500	
2	$t := t - 100$	900	1000	500	1000	500	
3	$\text{WRITE}(C1, t)$	900	900	500	1000	500	$\langle T, C1, 1000, 900 \rangle$
4	$t := \text{READ}(C2)$	500	900	500	1000	500	
5	$t := t + 100$	600	900	500	1000	500	
6	$\text{WRITE}(C2, t)$	600	900	600	1000	500	$\langle T, C2, 500, 600 \rangle$
8	FLUSH LOG	600	900	600	1000	500	
9	OUTPUT(C1)	600	900	600	900	500	
10		600	900	600	900	500	$\langle \text{COMMIT } T \rangle$
11	OUTPUT(C2)	600	900	600	900	600	

La politique de reprise après panne est alors la suivante :

- refaire toutes les transactions qui ont réussi (on retrouve les *commit*) en commençant du début ;
- défaire toutes les transactions incomplètes en commençant par la plus récente.

On peut également utiliser le système de point de contrôle pour éviter de parcourir entièrement le journal. Il existe cependant un point de détail technique : après l'écriture de l'entrée $\langle \text{START CKPT}(T1, \dots, Tn) \rangle$ on doit écrire tous les *buffers* contenant des données modifiées par les transactions. Il se peut donc que l'on ait une entrée $\langle \text{END CKPT} \rangle$ avant une entrée $\langle \text{COMMIT } T \rangle$ où T est une transaction concernée par le point de contrôle.

Concrètement, voici un ensemble d'opérations à réaliser pour reprendre un système avec *undo/redo* et point de contrôle :

1. commencer avec deux listes de transactions, la liste UNDO et la liste REDO. Initialiser la liste UNDO avec la liste de toutes les transactions enregistrées dans le compte rendu du point de contrôle le plus récent. Initialiser la liste REDO avec l'ensemble vide.

12. Gestion des transactions

2. faire une recherche en avant dans le journal, en partant du point de contrôle.
3. si une entrée dans le journal correspondant à un `BEGIN TRANSACTION` est rencontrée pour la transaction `T`, ajouter `T` à la liste `UNDO`.
4. si une entrée dans le journal correspondant à un `COMMIT` est rencontrée pour la transaction `T`, déplacer `T` de la liste `UNDO` vers la liste `REDO`.

Le système remonte le journal pour annuler les transactions de la liste `UNDO` (reprise en arrière). Il repart ensuite en avant pour rejouer les transactions de la liste `REDO` (reprise en avant).

Exemple 12.2.5 : Supposons que nous ayons le journal suivant :

```
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
<START CHKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CHKPT>
<COMMIT T2>
<COMMIT T3>
```

Que se passe-t-il lors que le système tombe en panne après cette séquence ? Dans ce cas, `T2` et `T3` sont considérées comme étant terminées. `T1` est antérieure au démarrage du point de contrôle et comme on trouve `<END CHKPT>` comme entrée, on considère que les changements effectués par `T1` ont été effectivement écrits sur le disque (car on écrit tous les changements en mémoire principale sur le disque après `<START CKPT(T)>`). On va donc rejouer les transactions `T2` et `T3`. Par contre, la première mise à jour de `T2` étant effectuée avant le point de contrôle, on est certain qu'elle a été écrite sur le disque (car on vide les *buffers*).

Par contre, si la panne arrive juste avant que l'entrée `<COMMIT T3>` ne soit écrite sur le disque, alors `T3` est considérée comme étant incomplète. Dans ce cas on va défaire `T3` en remontant dans le journal. □

12.2.6. Reprise après panne des supports

Lors d'une panne des supports, il faut restaurer la base de données à partir d'une copie de sauvegarde (*dump*) et utiliser le journal pour rejouer toutes les transactions qui s'étaient terminées depuis la création de la copie de sauvegarde. Attention, le processus de sauvegarde d'une base peut être très long. On ne peut évidemment pas se contenter de sauvegarder le journal, car celui-ci est souvent beaucoup plus volumineux que la base elle-même.

12.3. Gestion de la concurrence

Les interactions entre plusieurs transactions peuvent amener la base de données à être incohérente. En effet, deux transactions peuvent travailler sur des données identiques, et l'entrelacement de leurs opérations de mises à jour élémentaires peuvent conduire à violer les contraintes d'intégrité de la base de données. L'ordre dans lequel doivent être exécutées les différentes opérations élémentaires contenues dans les transactions doit donc être géré par un composant spécial, le *transaction scheduler*.

12.3.1. Quelques problèmes de gestion de la concurrence

Nous allons dans un premier temps présenter quelques problèmes de gestion de la concurrence : le problème de la perte d'une mise à jour, le problème des dépendances non validées et le problème de l'analyse incohérente. On suppose ici que les opérations de mises à jour de la base écrivent effectivement les données sur le disque.

Problème de la perte d'une mise à jour

Le tableau 12.1 présente une mise à jour perdue² :

- T1 récupère la valeur de A à t_1 . Appelons cette valeur v ;
- T2 récupère la valeur de A à t_2 ;
- T1 modifie la valeur de A à t_3 . Appelons cette valeur v' ;
- T1 modifie la valeur de A à t_4 . Ces modifications vont être faites en supposant que la valeur de A est v , alors que sa valeur est actuellement v' .

La mise à jour effectuée par T1 est perdue à la date t_4 , car T2 écrit une nouvelle valeur de A sans tenir compte de v' .

Transaction T1	Temps	Transaction T2
-	-	-
RETRIEVE(A)	t_1	-
-	t_2	RETRIEVE(A)
UPDATE(A)	t_3	-
-	t_4	UPDATE(A)

TABLE 12.1. – Exemple de perte d'une mise à jour

Le problème des dépendances non validées

Le tableau 12.2 présente un exemple de dépendance non validée :

- T2 met à jour l'élément A à t_1 ;
- T1 récupère la valeur de A ou met à jour A à t_2 ;
- T2 est annulée à t_3

2. L'opération RETRIEVE représente la récupération de la valeur d'une ou plusieurs données dans la base.

12. Gestion des transactions

Si T1 fait un RETRIEVE de A, T1 s'exécute sur une hypothèse fautive : l'élément A n'a pas la valeur observée à l'instant t_2 , mais il conserve la valeur qu'il avait avant l'instant t_1 .

Si T1 fait un UPDATE de A, cette mise à jour est perdue car le ROLLBACK à l'instant t_3 a pour conséquence que l'élément A récupère la valeur qui était la sienne avant l'instant t_1 .

On parle également de *données fantômes*.

Transaction T1	Temps	Transaction T2
-	-	-
-	t_1	UPDATE(A)
RETRIEVE(A) ou UPDATE(A)	t_2	-
-	t_3	ROLLBACK

TABLE 12.2. – Exemple de dépendance non validée

Le problème de l'analyse incohérente

Le tableau 12.3 présente un exemple d'analyse incohérente : on considère trois comptes C1, C2 et C3 qui présentent respectivement des soldes de 40, 50 et 20 au départ de la transaction. La transaction T1 calcule la somme des trois comptes. La transaction T2 transfère 10 du compte C1 vers le compte C2. Or on trouve une somme de 120 au lieu de 110.

Transaction T1	Temps	Transaction T2
-	-	-
RETRIEVE(C1)	t_1	-
sum = 40	-	-
RETRIEVE(C2)	t_2	-
sum = 90	-	-
-	t_3	UPDATE(C3, 30)
-	t_4	UPDATE(C1, 30)
RETRIEVE(C3)	t_5	-
sum = 120	-	-

TABLE 12.3. – Exemple d'analyse incohérente

12.3.2. Séquentialité

Définitions

Dans cette section, nous nous intéressons de manière plus formelle aux conditions qui doivent être réunies pour qu'un ensemble de transactions s'exécutant de façon concurrente préserve la cohérence de la base de données. Nous partons pour cela d'une hypothèse forte :

Hypothèse 12.3.1.

Toute transaction exécutée isolément des autres transactions amène la base de données d'un état cohérent vers un état cohérent. \square

Nous nous intéressons maintenant à des ordonnancements d'actions élémentaires, actions provenant d'une ou plusieurs transactions. Ces actions sont des actions de lecture ou d'écriture d'éléments de la base.

On supposera dans ce qui suit que toutes les transactions respectent l'hypothèse 12.3.1.

Définition 12.3.1.

Soient T_1, \dots, T_n des transactions. On appelle ordonnancement une séquence d'actions élémentaires de lecture et d'écriture effectuées par les transactions T_1, \dots, T_n . Cette séquence est complète : pour $i \in \{1, \dots, n\}$ toutes les opérations effectuées par T_i se retrouvent dans la séquence.

Soit T_i une transaction. On représentera par $r_i(A)$ une lecture de l'élément A de la base de données par la transaction T_i . On représentera par $w_i(A)$ une écriture de l'élément A de la base de données par la transaction T_i . \square

Exemple 12.3.1 : Soit une transaction T_1 effectuant les opérations suivantes sur la base de données :

- lecture d'un élément A
- modification de A
- lecture d'un élément B
- modification de B

Soit une deuxième transaction T_2 effectuant les mêmes opérations. Un ordonnancement pour T_1 et T_2 est :

$$r_1(A); w_1(A); r_2(A); w_2(A); r_2(B); r_1(B); w_2(B); w_1(B) \quad \square$$

Définition 12.3.2.

Soient T_1, \dots, T_n des transactions. Un ordonnancement σ sur T_1, \dots, T_n est séquentiel ssi $\forall i \in \{1, \dots, n\} \forall j \in \{1, \dots, n\} i \neq j$, si une action de T_i précède une action de T_j dans σ , alors toutes les actions de T_i précèdent toutes les actions de T_j dans σ . \square

Un ordonnancement séquentiel est donc un ordonnancement qui assure que les transactions s'effectuent les unes après les autres.

Exemple 12.3.2 : Reprenons l'exemple précédent. Un ordonnancement séquentiel sur T_1 et T_2 est le suivant :

$$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B) \quad \square$$

Considérons maintenant l'ordonnancement suivant :

$$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$$

12. Gestion des transactions

Cet ordonnancement n'est pas séquentiel. Pourtant, il produira le même résultat qu'un ordonnancement séquentiel : quelques soient les contraintes sur la base et en particulier sur A et B , si T_1 et T_2 respectent l'hypothèse 12.3.1, alors ces contraintes seront respectées après l'exécution de cette séquence d'actions. On dit alors que l'ordonnancement est *sérialisable*.

Définition 12.3.3.

Un ordonnancement est sérialisable si son exécution produit le même résultat qu'un ordonnancement séquentiel. □

Sérialisabilité par conflit

Nous cherchons maintenant à obtenir une condition suffisante pour assurer qu'un ordonnancement est sérialisable. Cette condition est appelée « sérialisabilité par conflit ».

Définition 12.3.4.

Soient deux actions consécutives dans un ordonnancement, alors ces deux actions sont en conflit si lorsque l'on les permute dans l'ordonnancement, l'effet d'au moins une de ces actions est changé. □

Considérons deux transactions différentes T_i et T_j et deux éléments d'une base de données X et Y . Il existe des séquences de deux actions qui ne sont pas en conflit :

- $r_i(X); r_j(Y)$ même si $X = Y$ car il n'y a pas de modifications de la base de données ;
- $r_i(X); w_j(Y)$ si $X \neq Y$ car si on écrit Y avant de lire X , rien n'est perturbé. De la même façon, la lecture de X n'influe pas l'écriture de Y ;
- $w_i(X); r_j(Y)$ si $X \neq Y$;
- $w_i(X); w_j(Y)$ si $X \neq Y$;

Par contre, il existe des séquences d'actions qui sont en conflit :

- $r_i(X); w_i(Y)$ sont en conflit. En effet, l'ordre des opérations dans une même transaction ne peut pas être changé.
- $w_i(X); w_j(X)$;
- $r_i(X); w_j(X)$ et $w_i(X); r_j(X)$.

En résumé, il n'y aura pas conflit à moins que les deux actions concernent le même élément et qu'une d'elle est une action d'écriture.

Définition 12.3.5.

On dit que deux ordonnancements sont équivalents par conflit si on peut transformer l'un en l'autre par une série d'échanges d'actions non conflictuels. Un ordonnancement est sérialisable par conflit s'il est équivalent par conflit à un ordonnancement séquentiel. □

Le fait d'avoir un ordonnancement sérialisable par conflit nous garantit d'avoir un ordonnancement sérialisable. Attention, cette condition est suffisante et pas nécessaire. Mais c'est la solution qui est utilisée dans la plupart des SGBD actuels.

On peut facilement vérifier cette propriété en construisant un *graphe de précedence*.

Définition 12.3.6.

Soit σ un ordonnancement comportant les transactions T_1 et T_2 . On dit que T_1 précède T_2 , noté $T_1 <_{\sigma} T_2$ s'il existe une action A_1 de T_1 et une action A_2 de T_2 telles que :

- A_1 précède A_2 dans σ ;
- A_1 et A_2 concernent le même élément de la base ;
- au moins une des deux actions est une action d'écriture.

On construit un graphe de précédence de la façon suivante : chaque nœud du graphe est une transaction et il existe un arc du nœud T_i au nœud T_j si $T_i <_{\sigma} T_j$. \square

Exemple 12.3.3 : Considérons l'ordonnancement suivant :

$$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \quad \square$$

Un graphe de précédence pour cet ordonnancement est présenté sur la figure 12.2.

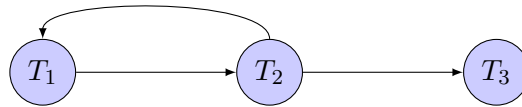


FIGURE 12.2. – Graphe de précédence pour $r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

Théorème 12.3.1.

Un ordonnancement ayant un graphe de précédence acyclique est sérialisable par conflit. Un ordonnancement sérialisable par conflit a un graphe de précédence acyclique. \square

Preuve :

La seconde partie du théorème se montre rapidement par contraposée. La première partie se montre par induction sur le nombre de transactions dans la base. \blacksquare

12.3.3. Verrouillage

Nous cherchons donc à avoir des ordonnancements de transactions concurrentes qui soient sérialisables. Une des façons de garantir cette propriété est d'utiliser un système de verrous. Le principe du verrouillage est que lorsqu'une transaction ne veut pas qu'un élément de la base de données ne soit modifié tant qu'elle l'utilise, elle *pose un verrou* sur cet objet. L'effet du verrou est d'empêcher tout autre transaction de modifier ou d'accéder à la valeur de l'élément sur lequel le verrou est posé.

Nous allons donc poser deux grands principes :

Principe 12.3.1.

Une transaction ne peut accéder ou modifier un élément que si elle a un verrou sur cet élément et qu'elle ne l'a pas relâché. Si une transaction a un verrou sur un élément, elle devra relâcher ce verrou. C'est le principe de cohérence des transactions. \square

Principe 12.3.2.

Deux transactions ne peuvent pas posséder en même temps un verrou sur un même élément. C'est le principe de légalité des ordonnancements. \square

12. Gestion des transactions

On peut construire des systèmes de verrouillage simple. Nous allons nous intéresser ici à un système de verrous un peu plus complexe composé de verrous exclusifs et partagés. On pourra se référer à [18] pour plus de détails sur les verrous, en particulier sur les systèmes de gestion des verrous.

Verrouillage à deux phases

Il existe une condition très simple pour garantir qu'un ordonnancement de transactions est sérialisable par conflit. Il s'agit du protocole de verrouillage à deux phases ou *two-phases locking* (2PL).

Définition 12.3.7.

Pour toute transaction, toutes les demandes de verrous précèdent les demandes de déverrouillage. □

Cela signifie deux choses :

- avant d'agir sur un élément (par exemple, un n-uplet d'une base de données), une transaction doit obtenir un verrou sur cet objet.
- après l'abandon d'un verrou, une transaction ne doit plus jamais pouvoir obtenir de verrous.

Théorème 12.3.2.

Le protocole du verrouillage à deux phases garantit la sérialisabilité par conflit. □

Preuve :

La démonstration se fait classiquement par induction sur le nombre de transactions. ■

Verrous exclusifs et partagés

Le système précédent, très simple, comporte un gros défaut : même si une transaction veut simplement lire un élément, elle doit prendre un verrou de même « importance » que si elle voulait l'écrire. Nous allons maintenant nous intéresser à un système de verrouillage particulier : les verrous exclusifs et partagés. Un verrou exclusif, noté X sur un élément de la base représente un verrou d'écriture. Il permet également d'accéder en lecture à l'élément. Il ne peut en avoir qu'un seul sur un élément. D'un autre côté, un verrou partagé, noté S représente un verrou de lecture seule sur un élément. Il peut en avoir plusieurs en même temps sur un même élément. On autorise donc la lecture simultanée par plusieurs transactions d'un élément. Par contre, on ne peut pas avoir à la fois un verrou exclusif et un verrou partagé sur un même élément.

Les possibilités de lecture ou d'écriture sur un élément suivant le type de verrou posé sont représentées sur le tableau 12.4.

Les possibilités d'obtention d'un verrou sur un élément par une transaction suivant le verrou déjà posé sur cet élément par une autre transaction sont représentées sur le tableau 12.5.

On pourra remarquer que les principes de cohérence des transactions, de légalité des ordonnancements et de 2PL sont facilement transposables avec ce système de verrous de deux types (cf. [18] pour plus de détails).

		Lecture	Écriture
Verrous	X	Oui	Oui
	S	Oui	Non

TABLE 12.4. – Possibilités de lecture ou d'écriture

		Demande de verrou	
		X	S
Verrou posé	X	Non	Non
	S	Non	Oui

TABLE 12.5. – Possibilités d'obtention d'un verrou

Le protocole d'accès aux données d'une base est le suivant :

Principe 12.3.3.

Une transaction qui souhaite accéder à la valeur d'un élément doit d'abord obtenir un verrou S sur cet élément.

Une transaction qui souhaite modifier un élément doit d'abord obtenir un verrou X sur cet élément.

Si une demande de verrou émise par la transaction T_2 est refusée car elle entre en conflit avec un verrou déjà détenu par la transaction T_1 , la transaction T_2 est mise en attente. T_2 devra attendre jusqu'à ce que T_1 abandonne le verrou.

Une transaction possédant un verrou de type S sur un élément peut demander un verrou de type X sans relâcher son verrou. \square

Remarque : Les verrous X sont conservés jusqu'à la fin de la transaction (COMMIT ou ROLLBACK). En général, les verrous S sont également conservés jusqu'à la fin de la transaction. \square

Nous allons maintenant revenir sur les trois problèmes que nous avons présentés en début de section. Nous allons pour cela introduire trois notations :

- $sl_i(X)$ signifie que la transaction T_i demande un verrou de type S sur l'élément X ;
- $xl_i(X)$ signifie que la transaction T_i demande un verrou de type X sur l'élément X ;
- $u_i(X)$ signifie que la transaction T_i relâche un verrou S ou X qu'elle possède sur l'élément X ;

Nous allons utiliser ces notations pour présenter les solutions. Il est bien évident que l'on n'écrira jamais explicitement ces opérations de gestion des verrous dans les transactions. C'est le rôle du gestionnaire de verrous.

La figure 12.6 présente la solution au problème de la mise à jour perdue utilisant le système de verrouillage.

L'opération UPDATE de la transaction T_1 à l'instant t_3 n'est pas acceptée, car c'est une demande implicite de verrou X sur A qui rentre en conflit avec le verrou S détenu par la transaction T_2 . T_1 est donc mise en attente. Pour des raisons identiques, T_2 est mise en attente à l'instant t_4 .

12. Gestion des transactions

Transaction T1	Temps	Transaction T2
-	-	-
RETRIEVE(A)	t_1	-
$sl_1(A)$	-	-
-	t_2	RETRIEVE(A)
-	-	$sl_2(A)$
UPDATE(A)	t_3	-
$xl_1(A)$	-	-
attente	-	-
attente	t_4	UPDATE(A)
attente	-	$xl_2(A)$
attente	-	attente

TABLE 12.6. – Exemple de perte d’une mise à jour « résolue »

Le verrouillage a résolu le problème de la perte d’une mise à jour en le remplaçant par un problème de *blocage* (voir section 12.3.4).

La figure 12.7 présente la solution au problème de dépendance non validée en utilisant le système de verrouillage.

Transaction T1	Temps	Transaction T2
-	-	-
-	t_1	UPDATE(A)
-	-	$xl_2(A)$
RETRIEVE(A) ou UPDATE(A)	t_2	-
$sl_1(A)$ ou $xl_1(A)$	-	-
attente	-	-
attente	t_3	ROLLBACK
attente	-	$u_2(A)$
obtention du verrou	-	-

TABLE 12.7. – Exemple de dépendance non validée résolue

L’opération réalisée à la date t_2 par la transaction T_1 (RETRIEVE ou UPDATE) n’est pas acceptée, car c’est une demande implicite de verrou sur A qui rentre en conflit avec le verrou X détenu par T_2 . T_1 est donc mise en attente.

Elle reste en attente tant que T_2 n’a pas atteint son point de terminaison (ici un ROLLBACK). Lorsque T_2 abandonne son verrou, T_1 peut poursuivre son exécution en observant une valeur *validée*.

La figure 12.8 présente la solution au problème d’analyse incohérente en utilisant le système de verrouillage.

L’opération UPDATE de la transaction T_2 n’est pas acceptée à l’instant t_4 , car c’est une demande implicite de verrou X sur $C1$ qui entre en conflit avec le verrou S déjà détenu par T_1 . La transaction T_2 est donc mise en attente.

Transaction T1	Temps	Transaction T2
-	-	-
RETRIEVE(C1)	t_1	-
$sl_1(C1)$	-	-
sum = 40	-	-
RETRIEVE(C2)	t_2	-
$sl_1(C2)$	-	-
sum = 90	-	-
-	t_3	UPDATE(C3,30)
-	-	$xl_2(C3)$
-	t_4	UPDATE(C1,30)
-	-	$xl_2(C1)$
-	-	attente
RETRIEVE(C3)	t_5	attente
$sl_1(C3)$	-	attente
attente	-	attente

TABLE 12.8. – Exemple d'analyse incohérente « résolue »

De façon similaire, le RETRIEVE de la transaction T_1 à la date t_5 n'est également pas accepté, car c'est une demande implicite d'un verrou S sur $C3$ qui entre en conflit avec le verrou X déjà détenu par T_2 . T_1 est donc également mise en attente.

On se retrouve donc avec un blocage.

12.3.4. Blocage

Comme nous venons de le voir, le verrouillage peut générer des problèmes de blocage, alors qu'il était censé résoudre les problèmes de sérialisabilité. Si un blocage se produit, il est souhaitable que le système le détecte et le supprime.

Nous présentons une solution simple pour gérer ce problème : détecter le blocage revient à détecter un cycle dans le *graphe d'attente* (*wait-for graph*). Il faut savoir que le système maintient une table de verrous qui lui permet de savoir quelles sont les transactions en attente d'un verrou et quelles sont les transactions possédant des verrous.

Définition 12.3.8.

Soit T_1, \dots, T_n des transactions. Un *graphe d'attente* est un graphe où :

- les nœuds du graphe sont les transactions ;
- il existe un arc du nœud T_i vers le nœud T_j ssi il existe un élément de la base A tel que T_j possède un verrou sur A , T_i attende un verrou sur A et que T_i ne peut obtenir son verrou que lorsque T_j aura relâché le sien. □

S'il n'y a pas de cycle dans le graphe, alors les transactions vont pouvoir se terminer. Dans le cas contraire, il y a un blocage. Il faut donc annuler une des transactions qui cause le cycle.

12. Gestion des transactions

On va choisir une des transactions bloquées comme *victime* et l'annuler, ce qui libère ses verrous et permet aux autres transactions de poursuivre leur exécution.

Remarque : La victime a « échoué » et a été annulée sans avoir commis de faute. Certains systèmes vont automatiquement relancer l'exécution d'une telle transaction depuis le début, en faisant l'hypothèse que les conditions qui ont causé le blocage ne vont probablement pas se reproduire. D'autres systèmes retournent simplement un message « victime d'un blocage » à l'application ayant initié la transaction. On laisse alors au programme le soin de gérer au mieux cette situation. □

Il existe d'autres méthodes pour résoudre les blocages. Par exemple, on peut utiliser un *timeout* pour les détecter, un ordre sur les éléments de la base etc. On peut se référer à [18] pour plus de détails.

12.3.5. Verrouillage intentionnel

Dans le modèle relationnel, on peut définir des verrous de *granularités* différentes : sur un tuple, sur une relation entière, sur l'ensemble de la base de données, ou sur une valeur d'attribut d'un tuple. On parle alors de *granularité de verrouillage*. Évidemment, il y a un compromis à trouver : plus la granularité est fine, plus la concurrence est élevée. Plus elle est grossière, moins on a besoin de positionner et de tester des verrous et donc moins la charge est élevée.

Par exemple, si une transaction a positionné un verrou X sur toute une relation, il n'y a pas besoin de positionner un ensemble de verrous X sur chaque tuple de la relation ; d'un autre côté, aucune transaction concurrente ne sera capable d'obtenir le moindre verrou sur cette relation, ou sur les tuples de la relation.

Le problème qui se pose est le suivant : comment savoir, lorsqu'une transaction T_1 demande un verrou X sur une relation R , si une autre transaction n'a pas déjà posé un verrou sur un tuple de R , sans avoir à examiner chaque tuple de R pour voir si l'un d'entre eux est verrouillé par une autre transaction, ou examiner chaque verrou existant pour voir si l'un d'entre eux est positionné sur un tuple de R ?

La solution qui existe est le *protocole de verrouillage intentionnel*. Ce protocole impose qu'aucune transaction n'a le droit d'obtenir un verrou sur un tuple sans avoir auparavant obtenu un verrou intentionnel sur la relation qui contient ce tuple.

Les niveaux de granularité considérés sont les suivants :

1. la relation est l'élément de plus haut niveau que l'on peut verrouiller ;
2. chaque relation est composée de *blocs* qui contiennent eux-même des tuples ;
3. enfin chaque tuple est verrouillable.

Dans le verrouillage intentionnel, il existe deux types de verrous. Tout d'abord, on peut manipuler des verrous X et S qui ont le sens vu précédemment et qui s'appliquent aux trois niveaux de granularité. Puis, on peut manipuler des *verrous intentionnels* : le verrou intentionnel partagé (IS), le verrou intentionnel exclusif (IX).

Principe 12.3.4.

Le principe du verrouillage intentionnel est le suivant :

1. si l'on veut placer un verrou S ou X sur un élément, on commence en haut de la hiérarchie ;
2. si on se situe sur l'élément que l'on veut verrouiller, on demande alors le verrou correspondant ;
3. si l'élément est « plus bas » dans la hiérarchie, alors on pose un verrou intentionnel correspondant sur ce nœud. □

Pour déterminer si une transaction est autorisée à avoir un verrou particulier, on utilise la table 12.9.

		Verrou demandé			
		IS	IX	S	X
Verrou Posé	IS	Oui	Oui	Oui	Non
	IX	Oui	Oui	Non	Non
	S	Oui	Non	Oui	Non
	X	Non	Non	Non	Non

TABLE 12.9. – Possibilité d'acquisition de verrous intentionnels

Détaillons les différentes possibilités :

- supposons qu'une transaction souhaite acquérir un verrou IS . Cela signifie que cette transaction souhaite lire un élément de granularité inférieure à l'élément courant. Le seul problème qui puisse arriver est qu'une transaction demande le droit d'écrire sur l'élément courant. Comme l'élément que l'on veut lire est de granularité plus fine, il pourrait y avoir problème.
- supposons qu'une transaction souhaite acquérir un verrou IX . Alors cette transaction souhaite mettre à jour un élément fils de l'élément courant. Dans ce cas, on doit interdire la lecture ou l'écriture de l'élément courant. Dans le cas de verrous IS ou IX déjà posé, les conflits éventuels se résolvent au niveau de granularité inférieur.
- supposons qu'une transaction souhaite acquérir un verrou S sur l'élément courant. La transaction veut alors lire l'élément courant dans sa totalité. Alors dans ce cas, on va refuser toute opération de modification de l'élément courant ou de ses descendants.
- finalement, supposons qu'une transaction souhaite acquérir un verrou X sur l'élément courant. La transaction veut modifier l'élément courant, donc on n'autorise aucune prise de verrou sur l'élément courant.

Attention, ce système de verrouillage intentionnel peut conduire à l'apparition de données *fantômes* (cf. section 12.5). On ne peut en effet verrouiller que les éléments existants. Supposons qu'une transaction T_1 pose un verrou IS sur une relation et un verrou S sur chacun des tuples de cette relation. T_1 travaille alors sur les tuples qu'elle a verrouillés. Supposons que pendant cette phase de traitement une autre transaction T_2 insère un nouveau tuple, dans ce cas ce tuple ne sera pas verrouillé par T_1 . Il se peut alors qu'il y ait un problème (i.e. un comportement non-sérialisable).

12.4. Bases de données distribuées

Nous considérons maintenant rapidement le problème des bases de données distribuées. On peut distribuer une base de données pour améliorer les performances du système, ou pour mieux pallier les éventuelles défaillances d'une machine. Nous n'allons ici nous intéresser qu'à un problème particulier. Supposons que nous ayons une transaction qui exécute des opérations sur des bases de données distribuées placées sur des sites différents. Comment être sûr que toutes les opérations effectuées par les transactions se sont effectivement déroulées dans les bases de données ? Comment pouvoir effectuer un *commit* sur la transaction ?

Pour résoudre ce problème, on utilise le principe du *commit* à 2 phases. Pour cela, on suppose que les hypothèses suivantes sont vérifiées :

- chaque base de données a son propre journal et il n'y a pas de journal global ;
- il existe un site, appelé *coordinateur*, qui joue un rôle spécial (il peut s'agir par exemple du site qui a initié la transaction). Son rôle est de garantir que les différents SGBD valident ou annulent les mises à jour dont ils sont responsables à l'unisson.

Le protocole est fondé sur un échange de message entre le coordinateur et les différents sites. Nous allons maintenant décrire la procédure de *commit* à deux phases sur une transaction T :

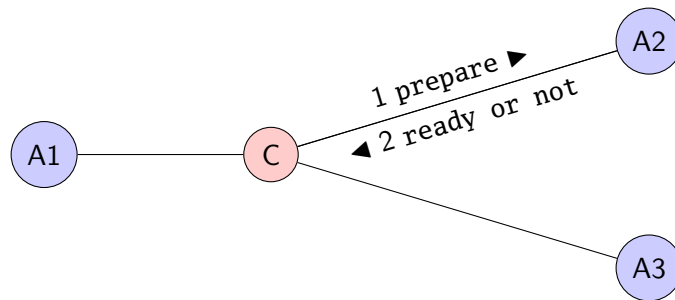
1. Phase I : préparation

- a) le coordinateur place une entrée `<Prepare T>` dans son journal ;
- b) le coordinateur envoie le message `prepare T` aux différents sites ;
- c) chaque site recevant le message `prepare T` décide de valider (*commit*) ou d'annuler (*rollback*) les composants de T le concernant ;
- d) si un site veut valider les composants de T le concernant, il entre dans un état dit *precommitted*. Dans cet état, le site ne peut plus annuler les composants de T sans un ordre du coordinateur. Le site s'assure localement que les composants de T ne devront pas être annulés en cas de panne du système, écrit l'entrée `<Ready T>` dans son journal et envoie le message `ready T` au coordinateur ;
- e) si le site veut annuler les composants de T, il écrit l'entrée `<Don't commit T>` dans son journal et envoie le message `don't commit T` au coordinateur. Il peut alors annuler les composants de T le concernant.

La phase I est résumée sur la figure 12.3.

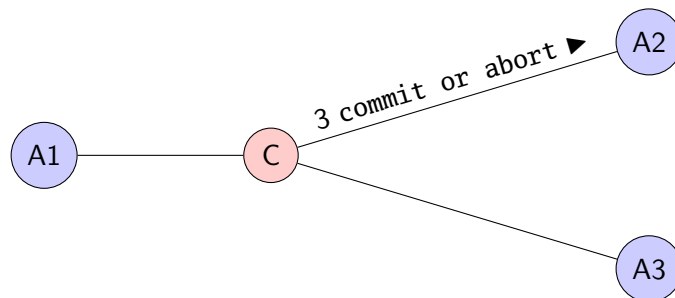
2. Phase II : validation

- a) si le coordinateur a reçu le message `ready T` de la part de tous les sites, alors il écrit l'entrée `<Commit T>` dans son journal et envoie le message `commit T` à tous les sites ;
- b) si le coordinateur a reçu le message `don't commit T` de la part d'un des sites, alors il écrit l'entrée `<Abort T>` dans son journal et envoie le message `abort T` à tous les sites ;

FIGURE 12.3. – Phase I du *commit* à deux phases

- c) si un site reçoit un message **commit** T, il valide les composants de T le concernant et écrit <Commit T> dans son journal ;
- d) si un site reçoit un message **abort** T, il annule les composants de T le concernant et écrit <Abort T> dans son journal ;

La phase II est résumée sur la figure 12.4.

FIGURE 12.4. – Phase II du *commit* à deux phases

Le coordinateur attend que tous les sites lui répondent dans la phase II. Si un site ne répond pas après un certain temps, il considère qu'il lui a envoyé un message **don't commit** T.

Cette procédure garantit que les transactions concernant des bases de données distribuées seront durables : même en cas de panne d'un des sites, on pourra récupérer toutes les bases dans un état cohérent.

12.5. Les fonctionnalités fournies par SQL

12.5.1. COMMIT et ROLLBACK

Par défaut, en utilisant l'interface SQL « classique » d'une base de données, chaque commande est elle-même une transaction. Par contre, lorsque du code SQL est embarqué dans un langage de programmation (cf chapitre 10), il faut pouvoir contrôler une transaction de manière explicite. Pour cela, on utilise la syntaxe suivante :

12. Gestion des transactions

Syntaxe (transaction) :

```
START TRANSACTION
...
[COMMIT|ROLLBACK]
```

□

Le mot-clé `START TRANSACTION` permet de démarrer la transaction. On peut valider la transaction par `COMMIT` ou l'annuler avec `ROLLBACK`. Attention, si on a utilisé des contraintes différées (cf. section 9.1.4).

12.5.2. Transactions en lecture seule

Les transactions qui n'effectuent que des opérations de lecture sur la base sont des cas particuliers de transactions. En particulier, il est beaucoup plus aisé d'exécuter une autre transaction en parallèle de telles transactions, car elles ne modifient pas la base. On peut préciser qu'une transaction n'effectue que des lectures, principalement pour des raisons de performances du SGBD, via la syntaxe suivante :

Syntaxe (transaction en lecture seule) :

```
SET TRANSACTION READ ONLY;
```

□

Une transaction qui peut lire et écrire des données est déclarée comme suit (c'est ce qui se passe par défaut) :

Syntaxe (transaction par défaut) :

```
SET TRANSACTION READ WRITE;
```

□

12.5.3. Niveau d'isolation

SQL ne propose pas de verrouillage explicite. On suppose par défaut que les transactions vont être toutes sérialisables. SQL propose toutefois de pouvoir atténuer le niveau d'isolation d'une transaction et d'avoir une propriété moins forte que la sérialisabilité, qui est le *seul* niveau d'isolation qui garantit une exécution correcte des transactions. Le fait de diminuer le niveau d'isolation va permettre d'améliorer les performances du SGBD, mais peut conduire à un état incohérent de la base.

On précise le degré d'isolation d'une transaction de la façon suivante :

Syntaxe (degré d'isolation) :

```
SET TRANSACTION ISOLATION LEVEL [SERIALIZABLE|REPEATABLE READ|
                                   READ COMMITTED|READ UNCOMMITTED]
```

□

Les niveaux d'isolation possibles sont présentés ici du plus strict (**SERIALIZABLE** qui assure un comportement correct des transactions) au moins strict (**READ UNCOMMITTED**).

Le fait de changer de niveau d'isolation peut amener des erreurs dans la gestion des transactions (violation de la sérialisabilité). Ces comportements anormaux sont au nombre de trois :

- la lecture salissante (*dirty read*) : supposons que la transaction T_1 effectue une mise à jour sur un certain tuple, que la transaction T_2 récupère ensuite ce tuple et que la transaction T_1 soit annulée par un **ROLLBACK**. Les valeurs des attributs du tuple observé par T_2 sont alors fausses.
- la lecture non renouvelable : supposons que la transaction T_1 récupère un tuple, que la transaction T_2 effectue ensuite une mise à jour de ce tuple et que la transaction T_1 récupère de nouveau le « même » tuple. La transaction T_1 a en fait récupéré le même tuple deux fois mais a observé des valeurs différentes.
- le fantôme : supposons que la transaction T_1 récupère un ensemble de tuples qui satisfont une certaine condition. Supposons que la transaction T_2 insère ensuite une ligne qui satisfait la même condition. Si la transaction T_1 répète maintenant la même demande, elle observera une ligne qui n'existait pas précédemment (que l'on appelle un fantôme).

Le tableau 12.10 présente maintenant quelles sont les violations de sérialisabilité autorisées par les différents niveaux d'isolation.

	lecture salissante	lecture non renouvelable	fantôme
READ UNCOMMITTED	Oui	Oui	Oui
READ COMMITTED	Non	Oui	Oui
REPEATABLE READ	Non	Non	Oui
SERIALIZABLE	Non	Non	Non

TABLE 12.10. – Lien entre niveaux d'isolation et violations de sérialisabilité

On pourra remarquer que l'apparition de fantômes peut être évitée grâce à un verrouillage du *chemin d'accès* utilisé pour obtenir les données considérées. Par exemple, on peut verrouiller les entrées d'un index sur l'élément.

12.6. Conclusion

La gestion des transactions a pour but de garantir les propriétés ACID pour les transactions gérées par un SGBD. Ces propriétés sont très importantes, car elles nous garantissent que les transactions vont laisser une base de données dans un état cohérent, même si elles s'exécutent de façon concurrente.

Les principes de la reprise après panne nous permettent d'assurer la durabilité des transactions. Nous n'avons vu ici que les procédures de journalisation *undo* et *undo/redo*. Le lecteur souhaitant plus de détails peut se référer à [18] et à [24].

En ce qui concerne la gestion de la concurrence parmi les transactions, il existe d'autres méthodes de verrouillage : verrous mis à jour, verrous incrémentaux etc. La gestion de

12. Gestion des transactions

la concurrence peut également être effectuée par points de contrôle temporisés et par validation. Là encore, on pourra se référer à [18] et à [22]. Un type de sérialisabilité, la sérialisabilité par vue, n'a pas été abordé ici. On pourra là encore trouver des détails dans [18].

Nous avons survolé le problème des bases de données distribuées. Nous avons présenté le principe du *commit* à deux phases, mais il en existe d'autres plus performants : *commit* à trois phases par exemple. Nous n'aborderons pas ces sujets par manque de place et de temps.

Finalement, la gestion des transactions est assez transparente pour un utilisateur d'un SGBD. Il existe toutefois des commandes SQL qui permettent de gérer plus finement les transactions au niveau utilisateur.

13. Bases de données objets

Le modèle relationnel est un modèle simple. Il est composé de tables plates, les attributs des relations sont des types simples etc. Malheureusement, ce modèle peut être parfois insuffisant pour obtenir une modélisation plus fine de la réalité. Par exemple, on pourrait vouloir avoir des attributs dans une relation qui ne soient plus des types simples, mais des types complexes. Les caractéristiques des langages de programmation orientés objets sont intéressantes pour la modélisation de bases de données : système de typage puissant (collections, références), notion de classe, héritage. Il peut donc être utile d'introduire ces notions dans un système de gestion de bases de données. De plus, les analyses et les modèles de systèmes complexes sont de plus en plus faits actuellement via des modèles objets supportant ces notions.

Un langage de programmation orienté objets comme C++, Java ou Smalltalk permet de créer, manipuler et détruire des objets en mémoire virtuelle. La durée de vie d'un objet n'excède pas la durée de vie du programme qui a servi à créer cet objet. Par ailleurs, comme dans tous les langages de programmation, il est possible de sauvegarder les objets dans des fichiers pour prolonger leur durée de vie. Ce type de sauvegarde fournit une première forme de *persistance* des objets qui assez primitive. En effet :

- la persistance n'est pas transparente, elle doit être gérée explicitement par le ou les programmes qui accèdent aux objets.
- la programmation de ces opérations de sauvegarde est délicate car les adresses des objets sont souvent des adresses virtuelles et il faut donc les convertir en adresses du support physique. De plus, lors de la définition de ce mécanisme d'adressage, il faudra gérer les relations inter-objets : la sauvegarde d'un objet peut amener à sauvegarder en fait tout un ensemble d'objets en relation avec ce dernier.
- l'accès aux objets en mémoire, contrôlé par le mécanisme de mémoire virtuelle, peut avoir de très mauvaises performances, et même au pire nécessiter un accès disque par objet.
- enfin, lorsque les objets sont stockés dans des fichiers physiques par des utilisateurs concurrents, l'accès à ces objets est difficile et nécessite la connaissance de la structure des fichiers. Cela engendre le problème de la dépendance physique entre programmes et données.

Nous retrouvons ici tous les problèmes engendrés par la gestion des données sur des fichiers classiques.

L'approche base de données objets apporte une solution récente à la gestion transparente de la persistance des objets. Deux approches pour la construction d'un système de gestion de base de données à objets (SGBDO) ont été dégagées. L'approche *relationnel-objet*, représentée par le standard SQL :1999 (ou SQL-99) et l'approche *orientée objet* représentée par le standard de l'ODMG (*Object Data Management Group*).

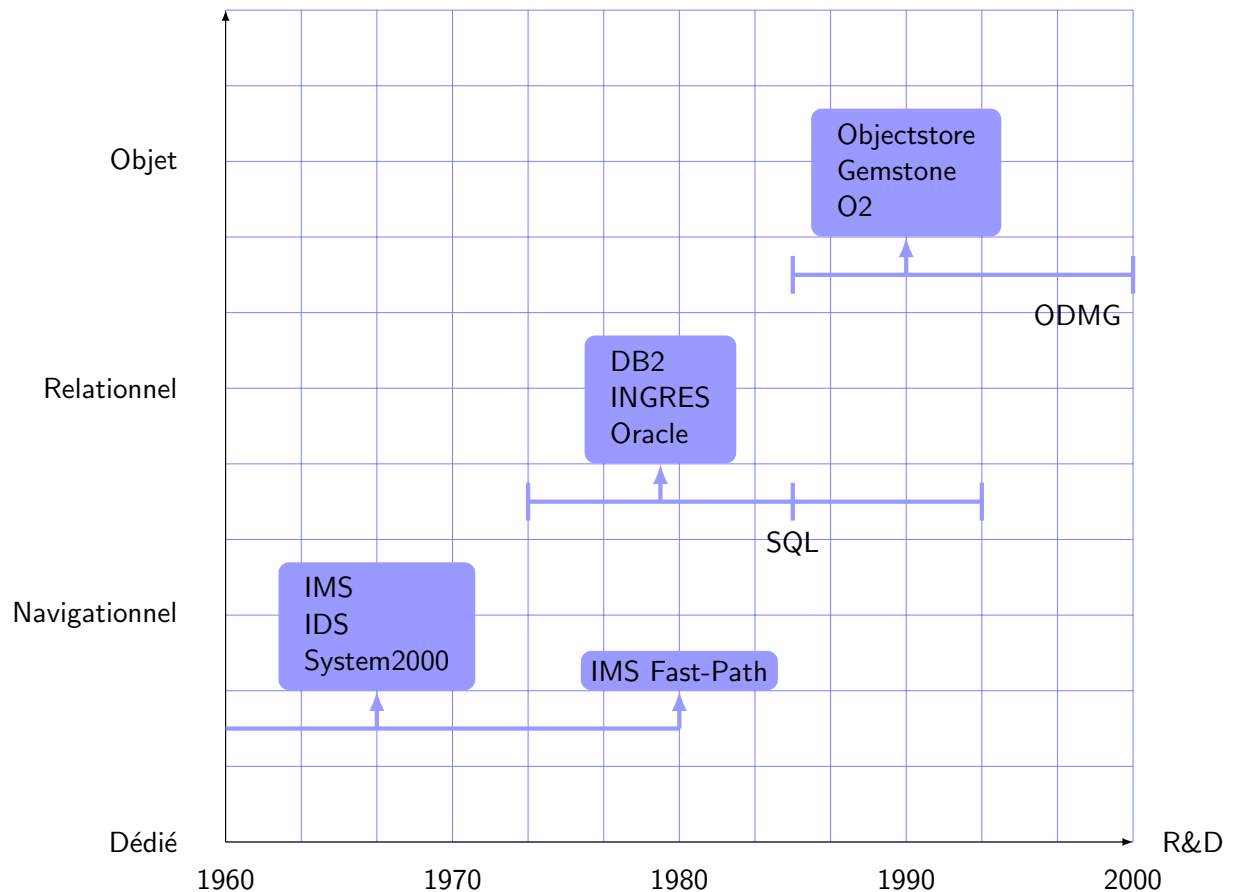


FIGURE 13.1. – Évolution des Systèmes de Gestion de Bases de Données

Ce chapitre présente les notions de base de données objets et de système de gestion de bases de données objets (SGBDO).

13.1. Historique des bases de données orientées objets

La figure 13.1 donne un historique de l'évolution des systèmes de gestion de bases de données.

L'approche système dédié désigne un SGBD spécialisé pour une application particulière comme la CAO. En général, un système dédié est un ensemble de d'applications qui utilise un système de gestion de fichiers. Cette approche s'oppose aux SGBD génériques qui visent des domaines d'application plutôt que des applications particulières. Pour les SGBD génériques, on observe une première étape de développement aboutissant à une première version du SGBD suivie d'une étape d'extension correspondant au réglage pour de meilleures performances et à l'ajout de fonctionnalités pour les versions suivantes.

Il aura fallu en moyenne deux décennies pour chacune des deux premières générations

de SGBD (navigationnels et relationnels) arrive à maturité. Un critère de maturité évident est la standardisation du langage (CODASYL et SQL). La période de maturité a été réduite pour les SGBDO qui ont profité des avancées des modèles relationnels et des langages de programmation orientés objets.

13.2. Pourquoi des bases de données objets ?

La simplicité du modèle relationnel a fait son succès : types simples, ensemble de tables plates, algèbre relationnelle permettant la construction de requêtes avec une base mathématique solide. Malheureusement, le modèle relationnel devient inadapté lorsqu'il est mis en œuvre dans des domaines d'application faisant intervenir des données fortement structurées. Nous présentons dans ce qui suit quelques-unes de ses insuffisances.

13.2.1. Nature des applications

Les applications de gestion classiques sont caractérisées par des types de données simples (entier, réel, date, ...), des articles plats, un besoin d'efficacité lors des accès aux données, une gestion de la concurrence, de la répartition et de la sécurité. Les SGBD relationnels satisfont cet ensemble de besoins. Les domaines d'application non traditionnels des bases de données comme l'ingénierie (CAO, FAO, génie logiciel, ...), la bureautique, l'administration de réseaux et actuellement les données sur le web engendrent des besoins nouveaux. Dans ce cas, le modèle de données doit être beaucoup plus riche avec de nouveaux types de base (graphique, texte, données géométriques, ...) et doit permettre l'utilisation de structures complexes (hiérarchie ou graphe). De plus, le langage doit permettre l'expression de calculs ou de mises à jour complexes sur des objets de base. Enfin, de nouvelles fonctionnalités sont attendues : objets multimédias, gestion de version ou de configuration, transactions évoluées etc.

13.2.2. Gestion des données fortement structurées

Le modèle relationnel est orienté tuple. Il permet la manipulation de relations plates dont le schéma est décrit par un ensemble d'attributs définis sur des domaines atomiques. Il n'offre pas de solutions satisfaisantes pour gérer des données dont la structure est complexe. En effet, un objet fortement structuré est décomposé en un ensemble de n-uplets répartis dans différentes relations plates et la notion de hiérarchie sous-jacente à la structure de tels objets se traduit par des références entre n-uplets. La reconstitution de tels objets à travers un modèle relationnel est coûteuse car elle impose la réalisation de jointures entre les différentes relations qui les décrivent.

13.2.3. Intégration dans un langage de programmation

Le langage normalisé SQL est incomplet (pas de structures de contrôle complexes par exemple) et exige de recourir à son intégration dans un langage de programmation. Le couplage de SQL avec un langage de programmation offre l'accès aux données relationnelles

13. Bases de données objets

depuis le langage de programmation grâce à un précompilateur ou à des appels dynamiques (cf chapitre 10). La correspondance entre les variables du programme et les requêtes SQL est assurée par la notion de curseur et ses opérations associées (déplacer, affecter, initialiser, ...) ¹. La programmation avec deux langages, l'un procédural et l'autre déclaratif, est assez délicate. De plus, ce couplage n'est pas homogène : systèmes de types différents, paradigmes de programmation différents (impératif d'un côté, assertionnel de l'autre), styles de programmation différents (élément par élément d'une part, ensembliste d'autre part). Des API comme ODBC ou JDBC permettent ce couplage de façon générique (cf. chapitre 10).

13.2.4. Conversion de types ou impedance mismatch

L'intégration de SQL dans un langage de programmation pose le problème de dysfonctionnement (*impedance mismatch* ²) : on encapsule des tables dont les attributs ont des types qui ne sont pas les mêmes que ceux du langage de programmation. La résolution de ce problème implique des conversions de types ainsi que des contrôles de types qui compliquent la programmation. On peut se référer à [44] et [3] pour plus de détails.

13.2.5. Les interfaces de manipulation

Les interfaces de manipulation des SGBD sont fondées sur des langages ensemblistes ou prédicatifs simples et puissants parmi lesquels SQL joue le rôle d'un standard. Elles peuvent également être fondées sur une approche « langage naturel » ou sur une approche graphique. Elles ne sont cependant pas suffisantes lorsque l'utilisateur manipule des applications complexes dans lesquelles le volume de données gérées est très important.

13.3. Bases de données objets

Les bases de données objets visent à résoudre les problèmes évoqués dans la section précédente par trois moyens complémentaires :

- réduire, voire éliminer les dysfonctionnements (en particulier l'*impedance mismatch*) entre langage de programmation et langage de bases de données ;
- supporter directement les objets arbitrairement complexes grâce à un modèle à objets ;
- partager le code réutilisable des applications au sein du SGBD.

Bien qu'il y ait consensus sur les *fonctions* que doit accomplir un SGBDO, il n'y a pas de *modèle* de base de données objets unique comme c'est le cas pour les bases de données relationnelles. Les modèles de bases de données objets sont issus des langages à objets et des modèles de données sémantiques (inspirés des réseaux sémantiques et des

1. Nous n'avons pas abordé cette notion dans ce cours, mais nous l'avons aperçu dans le chapitre 10 avec l'instruction FETCH par exemple

2. Ce terme provient bien de l'électronique. Le problème est le même que celui qui consiste à essayer de « connecter » deux appareils avec des impédances incompatibles.

Sémantique	Objets
entité	objet
identité d'entité	identité d'objet
type	classe
attribut	attribut
agrégation	classe dépendante
généralisation	héritage
associations	
	opération
	encapsulation

TABLE 13.1. – Correspondance entre concepts issus de la sémantique et issus des objets

modèles de connaissances issus de l'intelligence artificielle). Ils résultent de l'évolution de modèles comme le modèle relationnel ou le modèle entité/association.

L'objectif commun de tous ces modèles est de pouvoir capturer le plus de sémantique des données en termes de concepts tels que : entité, identité d'entité, association agrégation, généralisation, spécialisation, d'où leur nom de modèle conceptuel. Ces modèles sont ensuite traduits vers des modèles de base de données comme par exemple le modèle relationnel. Le tableau 13.1 propose une correspondance entre concepts sémantiques et concepts provenant du « monde » objet.

13.3.1. Base de données objets : définition

Définition 13.3.1 (base de données objets).

Une base de données objets est une organisation cohérente d'objets persistants et partagés par des utilisateurs concurrents. □

13.3.2. Système de gestion de base de données objets

Un système de gestion de bases de données objets est un SGBD qui fournit les fonctions de base d'un SGBD classique, c'est-à-dire :

- l'intégrité des données : les objets de la base doivent satisfaire les contraintes d'intégrité sémantiques indépendamment des pannes ;
- la persistance (ou permance) des données : les objets sont stockés indépendamment des programmes qui les créent et doivent être accédés rapidement grâce à des opérations d'accès ;
- la gestion des transactions : les programmes d'accès à la base de données sont des transactions qui vérifient les propriétés d'ACIDité (Atomicité, Cohérence, Isolation et Durabilité) grâce aux mécanismes de contrôle de concurrence et de fiabilité ;
- l'indépendance physique des données : le schéma conceptuel de la base de données n'impose pas d'implantation particulière des données, celle-ci est décrite par le schéma physique ;

13. Bases de données objets

- la possibilité de requêtes : les requêtes portant sur le schéma conceptuel ou externe doivent pouvoir être exprimées dans un langage déclaratif (comme SQL) qui se prête à l'optimisation.

Définition 13.3.2 (SGBD objet).

La définition minimale d'un SGBDO est donnée par :

$$SGBDO = SGBD + \text{objets} + \text{héritage} + \text{polymorphisme} \quad \square$$

13.3.3. Schéma conceptuel d'une base de données à objets

Le schéma conceptuel décrit un modèle de la base de données à objets indépendant de toute implantation. On doit pouvoir travailler avec une méthodologie qui décrit des règles de construction du schéma de la base de données à partir du schéma conceptuel.

Le schéma conceptuel d'une base de données objets est défini par l'ensemble des classes et des associations décrivant la réalité à modéliser.

Définition 13.3.3 (modèle conceptuel objet).

Modèle conceptuel = {Classes + Associations} □

Des notations semi-formelles peuvent être utilisées comme support de description de modèles conceptuels de bases de données objets (par exemple UML [4]).

13.3.4. Bases de données objets : deux approches

D'après les définitions précédentes, un SGBDO s'appuie sur deux technologies, les bases de données et les langages à objets. Plusieurs constructions de SGBDO sont donc possibles suivant la façon de combiner ces différentes technologies. En fonction du modèle supporté, les SGBDO sont classés en deux catégories : relationnel-objet et orientés objets selon que l'on parte du modèle relationnel ou d'un modèle à objets.

Approche relationnel-objet

Définition 13.3.4 (approche relationnel-objet).

L'approche relationnel-objet est fondée sur une extension du modèle relationnel et de son langage de référence SQL avec les concepts de la programmation orientée objets. □

Cette extension peut être vue comme une couche orientée objet traduite en relationnel. On y considère que les concepts des bases de données relationnelles sont suffisamment simples pour être étendus par des concepts orientés objets. Cette analyse a abouti au langage SQL-99, extension de SQL2. Une extension plus poussée consiste à supporter directement les concepts orientés objets au sein du SGBD relationnel ce qui implique d'étendre aussi l'algèbre relationnelle (i.e. décrire les opérations propre aux concepts orientés objets par des opérateurs dans l'algèbre relationnelle).

Approche orientée objet

Définition 13.3.5 (approche orientée objet).

L'approche orientée objet repose sur une extension d'un modèle à objets pour la définition et la manipulation d'objets persistants. □

L'approche base de données orientées objets (BDOO) est duale à l'approche relationnel-objet en ce sens qu'elle part d'un modèle à objets pour aboutir à un SGBDO. Il y a deux façons de réaliser un SGBDOO selon que l'on parte d'un langage de programmation orienté objets ou bien que l'on parte d'un modèle à objets.

Dans le premier cas, le langage doit être étendu pour permettre la définition de schémas de bases de données à objets et l'écriture de programmes d'application qui intègrent des appels au SGBDOO (db4o [16] avec Java et C#, ObjectStore avec C++ [31], Gemstone avec Smalltalk [20]). Ces SGBDOO sont souvent appelés systèmes à objets persistants.

La seconde solution est de définir un modèle à objets qui intègre naturellement les concepts des modèles de données sémantiques et des langages à objets. Le SGBDOO peut alors être indépendant de tout langage de programmation et offrir une interface multilingage (comme par exemple le SGBDOO O2 [3]).

13.4. Caractéristiques principales des bases de données objets

Les concepts de gestion de la persistance, de support de collections et d'évolution de schémas constituent les principales caractéristiques des bases de données à objets. Ces caractéristiques sont communes aux SGBDRO et aux SGBDOO.

13.4.1. Gestion de la persistance

La gestion de la persistance doit être telle que les objets persistants puissent être manipulés par les langages comme les objets temporaires (i.e. ne résidant qu'en mémoire virtuelle) sans conversion excessive de format et de structure et en totale transparence (pas d'API différente).

Le passage d'un objet de l'état temporaire à l'état persistant peut se faire par simple attachement à la base. La frontière entre objets de la base et objets de l'application est, contrairement aux approches intégrées, moins rigide et mieux contrôlable grâce à la compatibilité des types des objets temporaires et persistants.

Le SGBDO doit fournir les primitives pour créer, lire, écrire et détruire les objets persistants à partir d'un langage de programmation à objets.

Un objet persistant est créé avec un identifiant unique et permanent qui doit permettre de le retrouver rapidement sur le disque. Il est alors nécessaire de maintenir la correspondance entre identifiant et pointeur afin de manipuler les objets persistants dans l'espace temporaire.

Un objet persistant peut être créé par ^{3 4} :

3. OID se réfère à l'identifiant unique de l'objet.

4. Les opérateurs utilisés sont donnés à titre d'exemples.

13. Bases de données objets

```
OID = persist(pointeur);
```

Il peut devenir actif (en vue d'être manipulé) par :

```
pointeur = activate(OID);
```

Il peut être désactivé par :

```
OID = desactivate(pointeur);
```

Enfin, il est possible de le détruire par :

```
free(OID);
```

L'utilisation de ces primitives à partir des langages de programmation à objets fournit une forme de *persistance manuelle* (cf [23] par exemple pour un exemple d'utilisation sur le SGBDOO db4o [16]). D'autres types de persistance sont également possibles :

1. La persistance par héritage considère la persistance comme une propriété héritée automatiquement d'une classe racine de toutes les classes d'objets persistants appelée **Pobject**. Par exemple :

```
Personne extends Pobject...
```

```
Employe extends Personne
```

La création d'objet persistant est alors complètement transparente. Il est alors par contre nécessaire de réécrire un certain nombre de méthodes permettant d'assurer la persistance pour chaque classe. De plus, l'héritage multiple n'étant pas implanté « proprement » dans tous les langages de programmation orientés objets, cette solution peut être problématique (car on ne peut pas étendre dans ce cas une autre classe).

2. La persistance par référence associe la propriété de persistance directement à l'objet et non à sa classe. C'est une technique qui permet de définir la persistance directement par création explicite d'une racine de persistance ou par rattachement à un objet persistant. Par exemple :

```
paris : site;  
paris = new persistent ('paris');
```

```
martin : employe;  
martin = new employe('martin')
```

```
paris.affecter(martin) //martin devient persistant.
```

Ici, l'objet **paris** est persistant. Si l'objet **martin** est « stocké » dans l'objet **paris**, alors il devient automatiquement persistant.

Avec cette technique, chaque nom d'objet persistant doit être mémorisé dans un répertoire accessible par les programmes d'application.

Quelques éléments supplémentaires sur la persistance peuvent être trouvés dans [38] et [39].

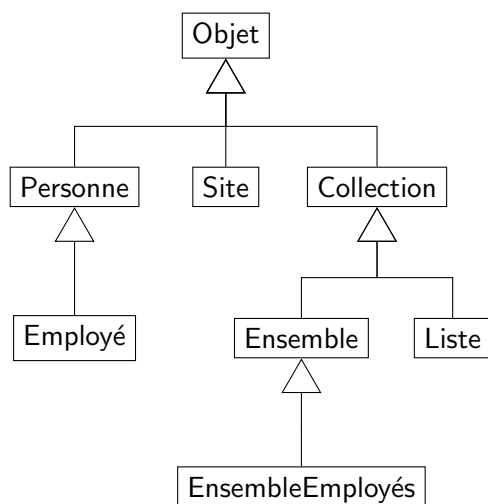


FIGURE 13.2. – Une hiérarchie de classes avec des collections

13.4.2. Support des collections

Un SGBDO doit fournir un support plus élaboré pour la manipulation des collections par une algèbre étendue aux objets complexes. Une collection est une abstraction permettant la manipulation d'ensembles d'objets. La classe **Collection** fournie avec le SGBDO permet donc le stockage et la manipulation ensembliste des objets de la base. Cette classe est paramétrée par un type (on crée une collection d'adresses ou de noms par exemple). Elle joue le rôle de conteneur. La figure 13.2 présente une hiérarchie partielle des classes contenues dans une base. La classe **Collection** est spécialisée en deux classes : **Liste** et **Ensemble**. La classe **Ensemble** est elle-même spécialisée en une classe **EnsembleEmployés** (pour des besoins particuliers, on n'a pas utilisé le paramétrage de la classe **Ensemble**, mais décider de créer une classe représentant un ensemble d'employés « en dur »).

L'utilisation des opérations associées à la classe **Collection** offre un niveau d'abstraction comparable à l'algèbre étendue à la manipulation d'objets complexes. Il est ainsi possible de combiner manipulation navigationnelle et manipulation ensembliste au sein d'un même langage.

13.4.3. Évolution des schémas

Comme le schéma d'une base de données à objets est un graphe de classes avec des associations, l'évolution du schéma devient une fonctionnalité importante. Elle doit entre autres, favoriser la réutilisation. Par exemple, il doit être aisé de spécialiser une classe en une autre classe.

Les domaines d'application des bases de données à objets exigent des possibilités de gestion de schémas beaucoup plus dynamiques et sophistiquées que celles offertes par les systèmes relationnels.

Les aspects dynamiques suggèrent que des modifications puissent être faites à tout

13. Bases de données objets

moment, sans impact sur les utilisateurs concurrents, et qu'elles deviennent visibles aussitôt. Les modifications de schémas agissent sur les définitions de classes et sur la structure d'héritage dans le graphe de classes. Les invariants de classe que le schéma doit respecter devront être conservés par toutes les mises à jour.

Les modifications de schémas supportées par un SGBDO sont classées en quatre catégories :

1. modification d'un attribut de classe : transparente si l'encapsulation est respectée ;
2. modification d'une opération de classe : doit être répercutée sur les opérations des programmes et éventuellement redéfinie dans les sous-classes ;
3. modification des liens d'héritage d'une classe ;
4. modification du graphe de classes.

Ces évolutions de schémas doivent préserver la cohérence du graphe de classes, des données et des programmes d'application. La liaison tardive ou liaison dynamique est un bon moyen pour éviter les modifications de programmes lors de la redéfinition d'opérations. Elle est présente dans la plupart des langages de programmation orientés objet.

Trois approches permettent de maintenir les objets cohérents en présence de modifications de schémas :

1. approche manuelle : interdiction de mise à jour d'un type dès que des instances de ce type existent. Il faut recopier les objets dans un nouveau type (assez lourd).
2. approche mise à jour immédiate : propage automatiquement et immédiatement les mises à jour du schéma sur les objets concernés.
3. approche mise à jour paresseuse : diffère la mise à jour des objets jusqu'au prochain accès en mémoire.

L'évolution de schémas est un besoin essentiel des SGBDO. Bien que plusieurs SGBDO apportent des solutions à la mise à jour dans les cas simples, les modifications complexes restent à la charge du programmeur. Le cas général ne peut être automatisé, mais peut être grandement assisté par des outils et des interfaces graphiques permettant la manipulation des schémas, de programmes et d'objets.

13.5. L'approche relationnel-objet

L'approche relationnel-objet consiste à étendre le modèle relationnel par les concepts « objets ». L'adjonction des fonctionnalités du modèle objet exige des extensions significatives de SQL pour supporter un modèle de types et d'objets complexes plus riche. Un groupe de travail (composé des grands éditeurs de SGBD, IBM, Oracle, Sybase, Informix, ...) de l'ANSI a standardisé ces extensions objets dans SQL :1999 (appelé également SQL99), successeur de SQL2.

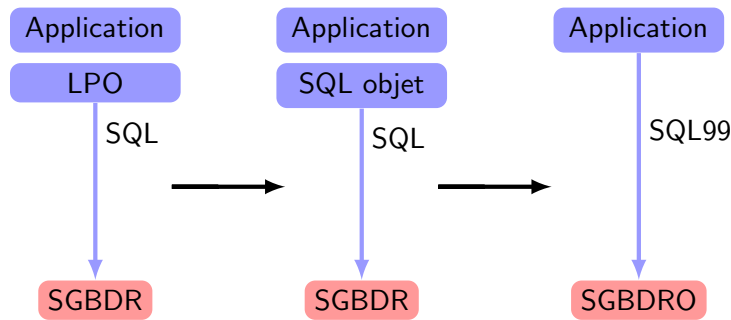


FIGURE 13.3. – Évolution des SGBD relationnel-objets

13.5.1. Évolution du relationnel à l'objet

La figure 13.3 illustre l'évolution à partir de l'approche SQL immergé vers le relationnel-objet qui minimise les effets sur les applications pour assurer la pérennité de l'existant.

L'étape « couche objet » repose sur l'extension objet de SQL. Elle réduit le recours à des langages à objets pour écrire l'application et simplifie donc l'écriture des programmes. Le SQL objet est simplement traduit en SQL. L'étape finale intègre les fonctionnalités objets dans le SGBD qui devient alors SGBDRO.

Le modèle relationnel repose sur deux concepts fondamentaux : les domaines de valeurs et les relations entre domaines. Pour obtenir un modèle relationnel-objet, on va donc modifier ces deux concepts.

- la définition de domaines devient spécifiée par la définition de types. Ces types peuvent être des types abstraits (avec des opérations) définis par l'utilisateur pour étendre des types déjà définis ou bien être des types structurés fournis par des constructeurs tels que table (relation), ligne (n-uplet ou tuple), liste, ensemble, tableau, ...
- le domaine et les relations entre domaines peuvent incorporer la notion d'identité essentielle à l'approche objets. Ainsi, un objet instance de type ou un n-uplet pourront être directement référencés à partir d'autres objets (ce qui n'est pas le cas dans le modèle relationnel). Les identifiants seront donc visibles (ils servent aussi aux accès indexés sur le disque). Cela permet également de construire des objets complexes en utilisant des combinaisons de tables, de types, de collections etc.

Notons que cette notion d'identité conserve une compatibilité ascendante avec SQL2. En effet, les adresses représentant les identifiants sont considérées comme des entiers et sont donc conformes aux spécifications des bases de données relationnelles.

Tous les exemples présentés dans ce qui suit ont été testés sur le SGBD PostgreSQL (cf. annexe A).

13.5.2. Définition des objets

SQL :1999 enrichit SQL avec la commande `CREATE TYPE` qui permet de créer des types simples ou structurés (*User Defined Types* ou UDT).

Syntaxe (création d'un UDT) :

```
CREATE TYPE T AS (  
    ...  
    nomChamp TypeChamp,  
    ...  
);
```

□

On peut bien évidemment utiliser d'autres UDT pour construire un UDT (i.e. un UDT peut apparaître comme type d'un champ d'un UDT).

Exemple 13.5.1 : On peut définir un type `Employe` comme suit.

```
CREATE TYPE Employe AS (  
    nom          CHAR(10),  
    date_naissance DATE  
);
```

Un UDT SQL :1999 correspond à la notion de classe et peut alors avoir des opérations associées. Pour déclarer une méthode associée à un type, on utilise la syntaxe suivante :

Syntaxe (déclaration d'une méthode d'un UDT) :

```
CREATE TYPE T AS (  
    ...  
    nomChamp TypeChamp,  
    ...  
)  
METHOD nomMethode() RETURNS Type;
```

□

Il suffit ensuite de définir la méthode grâce au mot-clé `CREATE METHOD` :

Syntaxe (création d'une méthode d'un UDT) :

```
CREATE METHOD nomMethode() RETURNS Type  
    Corps de la methode
```

□

Le corps de la méthode est écrit en utilisant un langage impératif (donc possédant des opérateurs de contrôle de flux complexes). C'est le même langage qui est utilisé pour construire des procédures PSM (cf. 10). On pourra se référer à [18] pour plus de détails.

Exemple 13.5.2 : On peut écrire une méthode `age` pour le type `Employe` qui permet de calculer l'âge de l'employé.

```

CREATE TYPE Employe AS (
    nom          CHAR(10),
    date_naissance DATE
)
METHOD age() RETURNS INTEGER(4);

CREATE METHOD age() RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
RETURN NULL ON NULL INPUT
BEGIN
    RETURN CAST(INTERVAL DAY (CURRENT_DATE - ADT_PERSONNE.PSR_DATE_NAISSANCE)
                AS FLOAT) / 365.2422
END;

```

On peut donc maintenant spécifier que les tuples d'une relation sont d'un certain type.

Syntaxe (déclaration de type) :

```

CREATE TABLE nomTable OF Type(
    ...
);

```

□

La relation possède comme attributs au moins les champs du type qui spécifie ses tuples. On peut bien évidemment ajouter d'autres attributs dans la définition de la table. Il est à noter que la définition de la clé primaire et des clés étrangères est de la responsabilité de la table, non du type défini.

Exemple 13.5.3 : On peut créer une table dont les tuples sont des employés, auxquels on a ajoutés une date d'arrivée dans l'entreprise.

```

CREATE TABLE listeEmployes OF Employe(
    date_arrivee DATE,
    PRIMARY KEY(nom)
);

```

Exemple 13.5.4 : On peut maintenant écrire une requête qui nous donne les noms des employés dont l'âge est inférieur à 35 ans.

```

SELECT l.nom()
FROM listeEmployes AS l
WHERE l.age() < 35;

```

13. Bases de données objets

La signification de la notation « pointée » sur l'attribut **nom** sera donnée dans la section [13.5.4](#). □

Une notion importante dans les langages orientés objets est la notion d'identité d'un objet (un objet a une identité unique). Une table dont les tuples sont définis comme étant d'un certain type peut avoir une « colonne » référence qui permet d'avoir un identifiant par tuple de la relation. Pour cela, on utilise la syntaxe suivante dans la déclaration de la table.

Syntaxe (référence) :

```
CREATE TABLE ... (  
    ...  
    REF IS attRef [SYSTEM GENERATED|DERIVED]  
    ...  
);
```

□

On construit donc la référence à partir d'un ou plusieurs attributs (par exemple la clé primaire de la table). Si on utilise le mot-clé **SYSTEM GENERATED**, c'est le SGBD qui est responsable de cette référence. Si on utilise le mot-clé **DERIVED**, le SGBD utilisera la clé primaire de la table comme référence.

Un type structuré peut être utilisé comme attribut d'une table (ce qui n'est pas autorisé en SQL2). Si on utilise un type structuré « simple », il n'y a aucun problème, on aura bien un attribut qui sera en fait composé des champs du type structuré (il faut toutefois faire attention à l'insertion de tuples et entourer l'instance du type composé de parenthèses). Par contre, on peut décider d'avoir un attribut dans une relation qui soit de type structuré et qui soit un tuple d'une relation de ce type. Dans ce cas, comment identifier le tuple ? On utilise alors une référence.

Syntaxe (référence vers un « même » type) :

```
attribut REF(Type) SCOPE Relation
```

□

Dans ce cas, **attribut** a pour type le type d'une référence vers un tuple de type **Type**. le scope permet de ne considérer que des tuples appartenant à la table **Relation** (qui doit bien entendu être de type **Type**).

Il est à noter que l'on peut également écrire des constructeurs et des destructeurs pour les types structurés. On pourra se référer à [\[28\]](#) pour plus de détails.

13.5.3. Les collections

Nous avons vu dans la section [13.4.2](#) qu'une des caractéristiques demandée à un SGBDO est le support des collections. Nous disposons maintenant de types complexes grâce à la commande **CREATE TYPE**, mais comment créer un attribut par exemple qui est un

ensemble d'objets? On dispose dans SQL :1999 d'un seul type de collection, les tableaux. On peut construire des tableaux avec un certain type (sauf d'autres tableaux) et une taille maximale du tableau. Les opérations habituelles sur les tableaux sont disponibles (accès à un élément par index, taille maximale, ...).

Syntaxe (déclaration d'un tableau) :

```
CREATE TABLE nomTable (
  ...
  att    Type ARRAY[nbMax],
  ...
);
```

□

L'accès aux éléments du tableau se fait de façon habituelle en utilisant l'opérateur [].

On peut noter que les tableaux permettent de coder, via les références des associations entre tables.

13.5.4. Manipulation des objets

Toute instance de type peut être un objet. Nous avons vu précédemment qu'un attribut d'une table pouvait être une instance d'un type structuré particulier ou une référence vers un tuple d'une relation d'un type particulier. Dans ces deux cas, comment accéder aux caractéristiques de ces objets, en particulier leurs champs?

Accès aux attributs d'un tuple avec un UDT

Supposons qu'une relation R soit de type T défini avec les champs c_1, \dots, c_n . Dans ce cas, la relation R possède n attributs, qui sont c_1, \dots, c_n . Par contre, un tuple t de la relation R n'a qu'un seul composant, et pas n : l'objet de type T lui-même. Dès lors, pour obtenir les valeurs des attributs de R pour t , il faut pouvoir accéder aux champs de l'objet. Pour cela, on utilise une notation classique « pointée » et un accesseur implicite au nom du champ concerné. La syntaxe est présentée sur un exemple générique.

Syntaxe (accès à un champ d'un tuple) :

```
SELECT t.c_i()
FROM R AS t
WHERE t.c_j()...
```

□

On voit donc que chaque attribut est accessible par une « méthode » qui porte son nom. De plus, on a besoin d'un nom de variable (ici t) pour le tuple pour pouvoir accéder aux champs de l'objet. On peut ainsi se ramener à l'appel de méthode, voir la section [13.5.2](#).

Accès à un objet en utilisant une référence

On peut également construire des tables dont un ou plusieurs attributs sont des références vers des objets (cf. la section 13.5.2). Supposons donc que nous ayons une relation R dont l'attribut att soit de type $REF(T)$. att est donc est donc une référence vers un tuple t de type T . Pour pouvoir accéder à l'objet référencé, on utilise la syntaxe suivante.

Syntaxe (déréférencement) :

`DEREF(att)`

□

Cette commande renvoie le tuple entier référencé par att . On peut également accéder à un champ particulier c_i de l'objet référencé via la syntaxe suivante :

Syntaxe (accès à un champ depuis une référence) :

`att->c_i`

□

13.6. L'approche orientée objet

Comme pour SQL :1999, il existe un groupe de standardisation créé par cinq éditeurs de SGBDOO (Object design, O2 Technology, Objectivity, Ontos, Versant) appelé ODMG. Ce groupe a produit un standard de bases de données à objets appelé ODMG-93 [5]. Cette proposition a été ensuite adoptée par l'OMG (*Object Management Group*) en 1994.

L'ODMG définit un modèle compatible avec celui de l'OMG ainsi que des interfaces indépendantes de tout langage de programmation pour traiter les objets. Le modèle de l'ODMG définit :

- un modèle de données pour représenter la base de données orientée objets,
- un langage de définition de données ODL (*Object Definition Language*),
- un langage de manipulation des objets OML (*Object Manipulation Language*),
- un langage d'interrogation d'objets OQL (*Object Query Language*).

Ce standard montre aussi l'intégration de ce modèle avec les langages de programmation C++ et Smalltalk. Une intégration à Java, JDO (*Java Data Object*), a également été soumise au *Java Community Process* et fait l'objet d'une JSR (*Java Specification Request*) [25]. Une référence complète sur l'ODMG est [5]

13.6.1. Le modèle à objets de l'ODMG

Le modèle de l'ODMG supporte des objets avec un état et un comportement. Les objets qui partagent les mêmes caractéristiques sont classifiés dans un même type. Tous les objets ont un identifiant (l'identifiant d'un littéral est sa valeur). On retrouve ici la définition classique des objets dans les langages à objets.

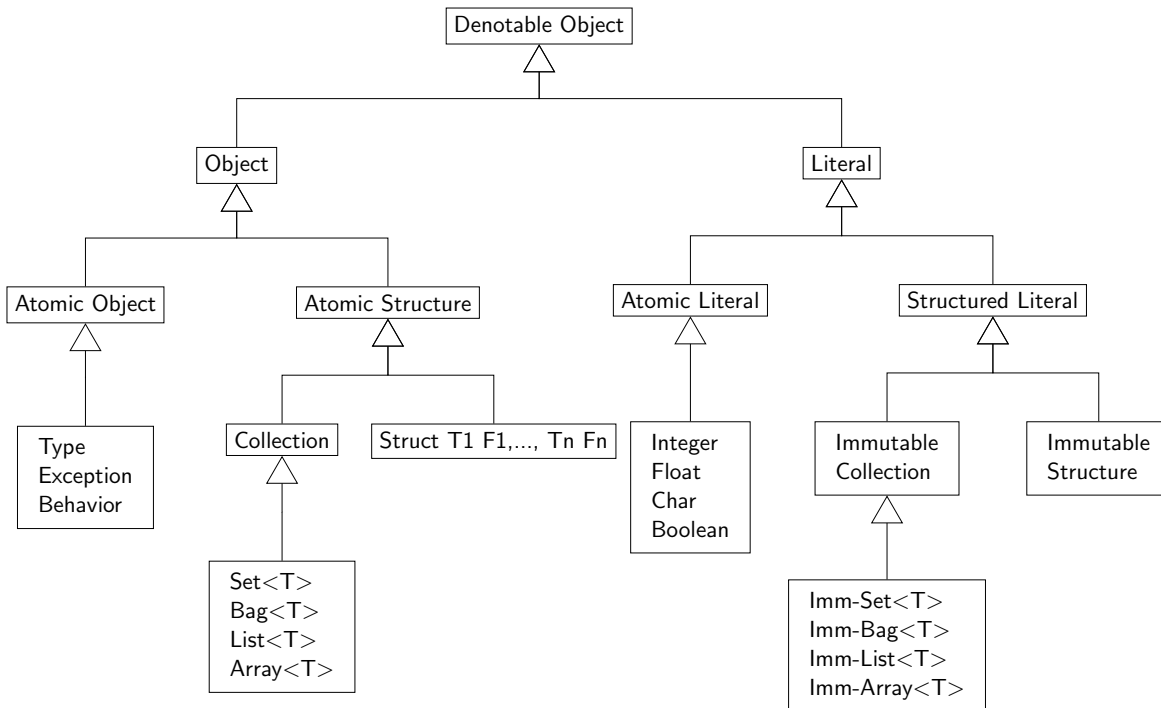


FIGURE 13.4. – Hiérarchie des types dans le modèle de l'ODMG

La durée de vie d'un objet est indépendante de son type. Elle est précisée lors de sa création. Un objet est dit *persistant* si sa durée de vie est supérieure à celle de la procédure qui a servi à le créer. Il est alors inséré dans la base de données et un emplacement lui est alloué sur le disque. *Un objet persistant doit être explicitement détruit.*

Les types sont organisés en hiérarchie d'héritage formant des liens de sous-typage. Lorsque l'héritage multiple est utilisé cette hiérarchie devient un graphe. Il existe une racine unique appelée *Denotable Object* qui est spécialisée en plusieurs objets atomiques ou structurés. Ce modèle de données est représenté sur la figure 13.4.

L'ensemble des opérations disponibles sur les objets est décrit dans le méta-modèle de l'ODMG présenté sur la figure 13.5.

Des opérations comme **create** (création d'objet), **delete** (suppression d'objet), **has_name?** (savoir si l'objet a un nom), **names** (renvoie la liste des noms d'objet), **exists** (savoir si un objet existe à partir d'un nom ou d'un identifiant), **same_as** (comparer deux noms d'objets ou un nom et un identifiant) permettent de travailler sur les objets.

Un attribut définit deux opérations **get_value** et **set_value**. Les attributs peuvent être multi-valués par des objets n-uplets (structures) ou des objets collection (**List**, **Set**, **Bag**, **Array**).

Les attributs dont les types sont des structures et/ou des collections permettent de coder les associations dans le modèle de données. Parmi les opérations relatives aux associations, on trouve **create** (qui établit une association), **add_member** (ajoute un membre à une association 1 : N ou M : N), **remove_member** (supprime un membre dans une association

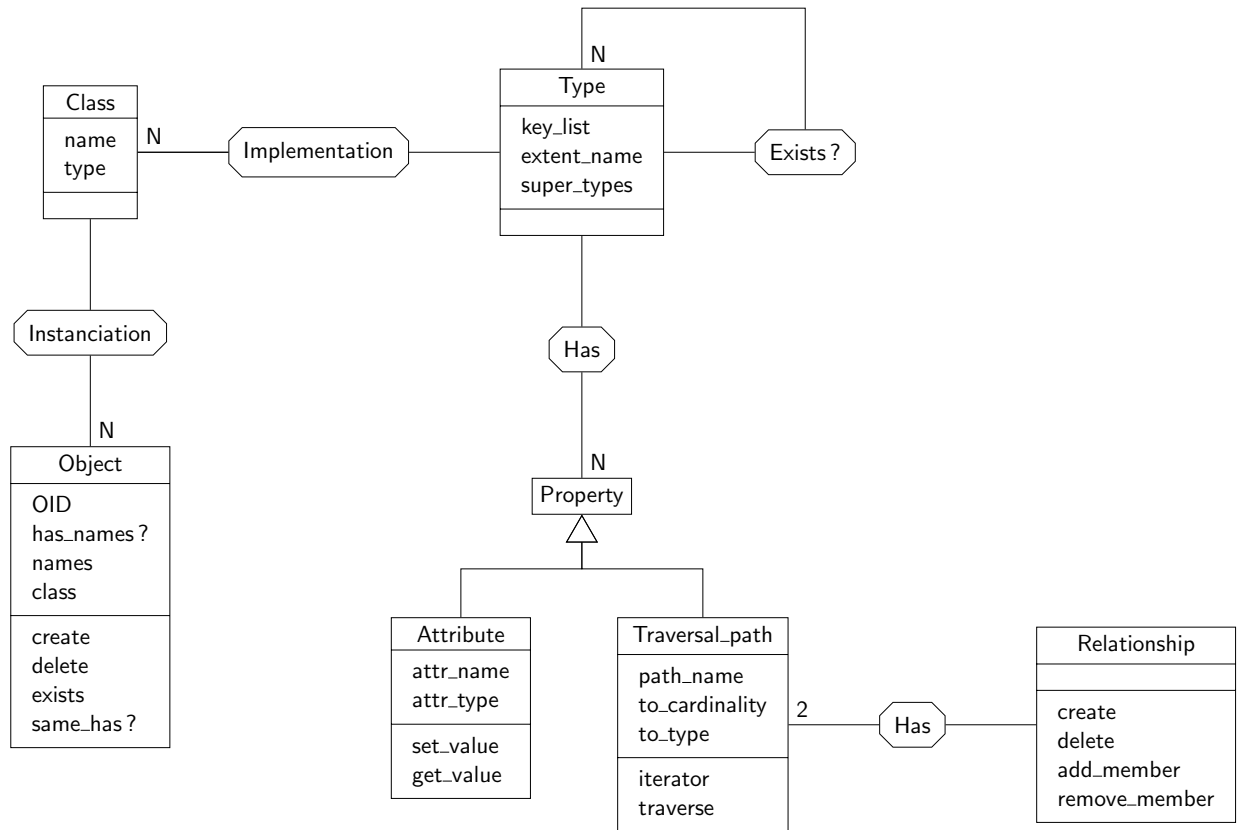


FIGURE 13.5. – Une partie du méta-modèle de l'ODMG

$1 : N$ ou $M : N$), **delete** (supprime une association), **traverse** (retrouve les objets cibles d'une association à partir d'un objet source), **create_iterator_for** (crée un itérateur pour parcourir une association multi-valuée) etc. Il faut remarquer la similitude avec les opérations classiquement utilisées dans le modèle réseau.

13.6.2. Définition d'objets avec ODL

ODL est un langage qui permet de définir les types des objets avec leurs champs et les signatures des opérations qui leur sont associées. Il ne permet pas de décrire l'implantation des opérations. Un schéma ODL doit pouvoir être supporté par tout SGBDO conforme au standard ODMG et pour tout langage pour lequel une interface existe. Cette interface intègre dans le langage de programmation concerné les constructions ODL pour définir les schémas et les constructions OML pour manipuler les objets de la base. Le langage de programmation sert à l'écriture des opérations (dont les signatures sont données en ODL) et des programmes d'application.

Nous allons voir dans ce qui suit comment utiliser ODL pour représenter une classe. Nous présentons la syntaxe générale d'une classe, puis nous précisons chacun des concepts.

Syntaxe :

```
class nomClasse extends C1 : ... : Cn
    (extent nomExtent key nomCle)
{
    attribute Type nomAtt1;
    attribute enum nomEnum {'String1', ..., 'Stringn'} nomAtt2;
    attribute Struct nomStruct {Type1 champ1, ..., Typen champn} nomAtt3;
    ...
    relationship [Set<C>|Bag<C>|List<C>|Array<C,i>|C] nomRelation
        inverse C::relation;
    ...
    TypeRetour nomMethode([in|out|inout] TypePar, ...) raises(nomException);
    ...
}
```

□

Nous pouvons maintenant détailler la construction d'une classe :

- une classe possède un nom introduit par le mot-clé **class**.
- une classe peut hériter de plusieurs autres classes. La sémantique de l'héritage est celle donnée habituellement aux langages orientés objets [29]. On introduit les classes dont la classe hérite par le mot-clé **extends** suivi des noms des classes séparés par des **:**.
- on précise ensuite le nom de l'*extension* de la classe. Il s'agit de l'ensemble d'objets instances de la classe qui existent à un instant donné. On utilise pour cela le mot-clé **extent**.

- il faut ensuite donner la clé primaire de la classe, c'est-à-dire quels sont les attributs qui vont permettre de déterminer une instance de la classe de façon unique. Il existe déjà pour cela l'identité de chacun des objets, donc ce paramètre est optionnel. On introduit une clé via le mot-clé **key**. Attention, **key att1, att2** signifie que **att1** ou **att2** peuvent être des clés pour la classe. Si l'on veut préciser que c'est le couple (**att1, att2**) qui est une clé, on utilise des parenthèses.
- on liste ensuite les attributs en utilisant le mot clé **attribute**. On précise ensuite leur type et leur nom. On voit ici que l'on peut construire plusieurs types :
 1. un type simple ;
 2. une énumération à laquelle on donne un nom **nomEnum**. Cette énumération pourra être ensuite utilisée avec le nom **nomClasse::nomEnum** ;
 3. un type structuré construit à partir d'autres types avec le mot-clé **Struct**. De la même façon que pour les énumérations, on pourra y faire référence depuis une autre classe via **nomClasse::nomStruct**.
- les relations, introduites par le mot-clé **relationship** représentent les relations existant entre les objets de différentes classes. Elles sont caractérisées par :
 - la cardinalité de la relation entre les objets. Un objet de **nomClasse** peut être lié à un seul objet de **nomRelation**. On utilise alors le type simple de la classe. En cas de multiplicité supérieure à 1, on peut utiliser les types paramétrés **Set**, **Bag**, **List** ou **Array** (qui prend en plus une taille en paramètre).
 - on précise ensuite quel est le nom de la relation dans la classe cible grâce au mot-clé **inverse**.
- les méthodes associées à une classe sont représentées par leur signature : type de retour, nom, liste des types des paramètres et exceptions qui peuvent être levées par la méthode. On introduit les exceptions par le mot-clé **raises**. On précise si un paramètre est passé par valeur ou par référence grâce aux mots-clés **in**, **out** et **inout**.

On peut remarquer que les relations peuvent également être représentées par des attributs (rien n'empêche un attribut d'être un ensemble d'objets par exemple). Toutefois, il est important de noter que les relations permettent de préciser quel est le nom de la relation dans la classe cible, ce qui n'est pas permis par un attribut. On choisira donc toujours de représenter un lien structurel entre plusieurs objets de différentes classes par des relations.

Exemple 13.6.1 : Construisons une classe permettant de représenter des sites accueillant des employés. On peut écrire la classe de la façon suivante :

```

class Site (extent sites, keys nom) {
  attribute string nom;
  attribute Employe chef;
  relationship Set<Employe> emps
    inverse Employe::affectation;
  void ajouter (in Employe)
    raises (deja_affecte);
}

```

```
};
```

On construit maintenant la classe `Employe` :

```
class Employe (extent employes, keys nom) {
    attribute string nom;
    attribute string adresse;
    relationship Site affectation inverse Site::emps;
    void affecter (in Site)
        raises (deja_affecte, affectation_incorrecte);
};
```

On peut également spécialiser la classe `Employe` en une classe `Ingenieur` qui aura un attribut supplémentaire.

```
class Ingenieur:Employe (extent ingenieurs) {
    attribute string projet;
};
```

13.6.3. Interface avec un langage : cas de C++

L'interface ODMG avec C++ comprend deux extensions de C++ : ODL-C++ et OML-C++. L'adaptation du modèle ODMG à C++ respecte la syntaxe et la sémantique de C++, en éliminant le dysfonctionnement entre le langage de programmation et la base de données à objets.

Les concepts du modèle ODMG sont fournis par une bibliothèque de classes (`Persistent_Object`, `Database`, `Ref`, `Collection`, ...). Le modèle de persistance combine la persistance par héritage et la persistance par référence.

La seule différence syntaxique est l'ajout du mot-clé `inverse` pour représenter les associations.

L'exemple suivant donne la représentation ODL-C++ du schéma ODL précédent donné dans la section 13.6.2.

```
class Employe : public Persistent_Object {

    String nom;
    String adresse;

    Ref<Site> affectation inverse Site::emps;

public:
    void affecter(Ref<Site> s);

}

class Ingenieur : Employe {
```

```
        String projet;
    }

    class Site : public Persistent_Object {

        String nom;
        Ref<Employe> chef;

        Set<Ref<Employe>> emps inverse employe::affectation;

    public:
        void ajouter(Ref<Employe> e );
    }
```

La classe `Persistent_Object` fournit les opérations pour créer (**new**) et détruire les objets persistants ou temporaires. La classe `Ref` sert à manipuler les références d'objets en surchargeant les opérateurs de manipulation des pointeurs de C++ comme `->`.

13.6.4. Manipulation des objets avec OQL

OQL est le langage de requêtes qui supporte le modèle objet proposé par l'ODMG. Sa syntaxe est proche de SQL et peut être adaptée à celle du langage de programmation. OQL a été conçu pour être facilement intégré à un langage de programmation comme C++, Smalltalk ou Java.

OQL permet de manipuler des objets de manière déclarative sans se limiter à l'accès ensembliste. Une requête OQL est une fonction qui, appliquée à une expression désignant des objets, renvoie un objet résultat. De nombreuses expressions sont possibles, comme les expressions arithmétiques, les constructions d'objets, de collections ou de structures, les expressions de définition de requêtes, les expressions de manipulation de collections, Le fait de pouvoir combiner directement des expressions d'un langage hôte et d'un langage de requête est un grand avantage.

Accès à un élément d'un objet

L'accès à un élément d'un objet se fait classiquement par la notation pointée.

Syntaxe : Soit a un objet et p une propriété du type de cet objet. Alors, l'application de p à a se fait par :

$a.p$

□

Évidemment, p peut être de plusieurs « types » différents :

— si p est un attribut, alors $a.p$ renvoie la valeur de cet attribut ;

- si p est une relation, alors $a.p$ renvoie l'objet ou la collection d'objets associés à a par la relation p ;
- si p est une méthode, alors $a.p(\dots)$ est le résultat de l'application de p à a .

Clause **SELECT - FROM - WHERE**

On retrouve dans OQL une clause **SELECT - FROM - WHERE** similaire à celle qui existe dans SQL. Sa syntaxe générale est la suivante.

Syntaxe :

```
SELECT [DISTINCT] a.p
FROM Collection a
WHERE C(a)
ORDER BY ...
```

□

La clause **SELECT** est suivie d'une liste d'expression construites avec la notation pointée. La clause **FROM** est suivi d'un ensemble de déclarations de variables typées avec une collection. La plupart du temps, cette collection est en fait un extent (cf. 13.6.2). La clause **WHERE** est suivie d'une expression booléenne qui ne peut manipuler que des variables déclarées dans la clause **FROM**.

Le résultat de cette requête est un *bag*, c'est-à-dire un multi-ensemble non ordonné. On peut demander à obtenir un ensemble plutôt qu'un *bag* en utilisant le mot-clé **DISTINCT** après **SELECT**. On peut demander à obtenir une liste en utilisant la clause **ORDER BY**. On voit ici que la syntaxe d'OQL est très proche de celle de SQL.

Expressions quantifiées

Il existe deux expressions OQL permettant de tester universellement ou existentiellement une propriété sur une collection.

Pour tester si les éléments d'une collection C vérifient tous une propriété P , on utilise la syntaxe suivante.

Syntaxe :

```
FOR ALL x IN C : P(x)
```

□

Pour tester s'il existe au moins un élément d'une collection C vérifiant une propriété P , on utilise la syntaxe suivante.

Syntaxe :

```
EXISTS x IN C : P(x)
```

□

Agrégations et regroupements

On peut, tout comme dans SQL utiliser des expressions d'agrégations dans OQL. Dans SQL, ces expressions s'appliquaient à une colonne particulière d'une relation. Dans OQL, les opérateurs d'agrégations s'appliquent à une collection dont les membres sont d'un certain type :

- l'opérateur **COUNT** peut être appliqué à n'importe quelle collection ;
- les opérateurs **SUM** et **AVG** peuvent être appliqués à des collections de type numérique ;
- les opérateurs **MIN** et **MAX** peuvent être appliqués à des collections dont le type est « comparable ».

L'opérateur **GROUP BY** utilise la syntaxe suivante :

Syntaxe :

GROUP BY *f*₁:*e*₁, ..., *f*_{*n*}:*e*_{*n*}

où chaque *f*_{*i*} est un nom d'attribut et *e*_{*i*} une expression. On appelle *f*_{*i*} : *e*_{*i*} un *attribut de partition*. □

Chaque *e*_{*i*} peut utiliser des variables apparaissant dans la clause **FROM** de la requête. Comment construire alors le résultat de la requête? Supposons que la clause **FROM** concerne *k* variables x_1, \dots, x_k . Chaque x_i parcourt une collection C_i . Pour chaque tuple $\langle m_1, \dots, m_k \rangle \in C_1 \times \dots \times C_k$ qui vérifie les expressions exprimées dans la clause **WHERE**, on évalue les expressions de la clause **GROUP BY** et on obtient une liste de valeurs $e_1(m_1, \dots, m_k), \dots, e_n(m_1, \dots, m_k)$ qui représente le groupe auquel $\langle m_1, \dots, m_k \rangle$ appartient.

La valeur retournée par la clause **GROUP BY** est en fait une collection intermédiaire **Struct** (**f**₁:*v*₁, ..., **f**_{*n*}:*v*_{*n*}, **partition**:*P*). Les *n* premières valeurs de la structure représentent le groupe et le dernier champ **partition** représente les $\langle m_1, \dots, m_k \rangle$ qui appartiennent au groupe. La clause **SELECT** s'applique ensuite sur ces différentes structures avec des restrictions identiques au cas de SQL.

De la même façon qu'en SQL, on peut utiliser une clause **HAVING** pour restreindre les tuples de chaque partition.

Notons également que comme OQL est très intégré au langage de programmation hôte, on peut affecter des variables du langage de programmation aux résultats d'une requête OQL si le type de la variable est compatible avec le résultat fourni par la requête.

13.7. Exemples de systèmes de gestion de bases de données orientées objets

Les SGBDRO actuels supportent tous des extensions de SQL, sans que la syntaxe ne soit (malheureusement) exactement celle de SQL :1999 (qui reste pourtant un standard). Cependant, la plupart des fonctionnalités de ces SGBDRO restent les mêmes que celles décrites dans ce chapitre. Attention par contre, certaines fonctionnalités, comme l'héritage entre UDT par exemple, ne sont pas toutes disponibles. Nous nous concentrerons donc dans ce chapitre sur les SGBDOO.

13.7.1. OPAL de GEMSTONE

OPAL [20] s'appuie sur SmallTalk et offre une persistance par héritage. Le modèle de données de OPAL est une extension simple de Smalltalk (modèle tout objet). La classe **Object** de Smalltalk est étendue avec des classes permettant la manipulation d'objets persistants et les transactions. Un objet de OPAL obtient un identifiant persistant lors de sa création. Une classe **Collection** fournit les classes de manipulation des agrégats. Afin de pouvoir compiler les programmes d'application OPAL, Smalltalk, non typé, est enrichi par les contraintes de types qui associent à chaque attribut le nom d'une classe OPAL donnant son type.

Le modèle de persistance est un modèle par héritage de la classe **Object** de GEMSTONE. Seules les instances de classes OPAL peuvent être gérées par le SGBDO.

GEMSTONE fournit une interface de programmation (API) pour des langages comme C, C++, Smalltalk. Elle permet à un programme d'application de créer (**instantiate**), lire (**fetch**), ou écrire (**store**) des objets de la base avec conversion automatique de format, de lire des méta-informations (**fetch info**) et d'envoyer des messages (**send**) ou des instructions Smalltalk (**execute**) à un objet OPAL. Dans ce dernier cas, l'opération OPAL est exécutée directement par le SGBDO car Smalltalk est interprété.

13.7.2. Objectstore

OBJECTSTORE [31] étend C++ et offre une persistance par référence. Le modèle de données d'Objectstore est une extension de C++ intégrant des objets (instances de classes) et des valeurs (structures et littéraux). Ce modèle a servi de base à la définition de l'interface ODMG avec C++.

Chaque objet d'une classe C++ gérée par Objectstore a un identifiant persistant et peut être directement accédé à partir de son adresse virtuelle par une technique propriétaire. Les associations binaires sont supportées par des liens inverses. Une bibliothèque de classes C++ fournit des opérations pour gérer les collections, les transactions et les exceptions.

Objectstore offre deux types d'interface aux langages C++ et C : une bibliothèque accessible par macros et un langage de plus haut niveau, le DML, qui doit être pré-compilé (le pré-compilateur fait partie du SGBDO).

Un schéma est un fichier de définitions (*header*) contenant des classes C++, utilisant des expressions DML pour spécifier les collections et les liens inverses.

Le modèle de persistance ressemble à celui de l'ODMG. La persistance est spécifiée uniquement lors de la création (par une première instruction **new**) et exige l'attachement à un nom persistant ou à un objet persistant (par une seconde instruction **persistent**).

13.7.3. Orion

Les deux prochains SGBD présentés dans ce chapitre s'appuient sur un modèle à objets de plus haut niveau sémantique qu'un modèle basé sur les langages à objets.

Le modèle objet de Orion s'inspire de Lisp et de Smalltalk dans la mesure où tout est considéré comme un objet (l'objet est l'unique moyen de représentation d'entités

conceptuelles). Il offre une persistance par héritage.

L'état d'un objet est défini par la valeur de ses propriétés. Un objet peut être primitif, auquel cas il ne contient qu'une valeur, ou plus complexe auquel cas il possède un identifiant unique et des propriétés pouvant référencer d'autres objets. Un objet est complètement défini par ses propriétés (son état) et ses opérations. Une opération Orion est une fonction Lisp qui opère sur l'objet et qui restitue son état. L'état d'un objet et ses opérations ne sont pas directement accessibles à un utilisateur. L'interface unique d'accès à un objet est définie par les messages auxquels l'objet peut réagir (comme dans Smalltalk).

Orion offre une hiérarchie de classes primitives avec la classe **Object** qui est la racine de trois classes : **TypePrimitif**, **Collection** et **Classe**. Cette dernière est une métaclasse qui fournit des primitives pour concevoir de nouvelles classes et contient des objets qui sont des classes.

Orion propose deux concepts importants de modélisation : les objets composés (*composite objects*) et les versions d'objets.

Un objet composé est organisé sous forme d'une hiérarchie d'objets composants. Il modélise la relation fait_partie_de (**part_of**) entre objets, différente de la relation est_un (**is_a**) fournie par la hiérarchie de classe. Les objets composés permettent le maintien automatique de l'intégrité référentielle.

Un objet peut posséder un certain nombre de versions. Le fait qu'un objet puisse avoir des versions doit être déclaré au niveau de sa classe. Une version peut être dérivée de l'objet original ou d'une version de l'objet. Le support de la gestion des versions est motivé par les applications techniques.

L'unique point d'entrée d'Orion est une extension simple du langage Lisp avec des possibilités orientées objets et des fonctions de bases de données. La syntaxe de ces fonctions combine celle de Lisp et d'autres langages de programmation à objets.

13.7.4. O2

O2 [3] offre un modèle de données objets avec valeurs et objets complexes. L'originalité du modèle est sa capacité à traiter les valeurs avec la même généralité que pour les objets. Une valeur a un type qui est atomique ou construit (tuple, liste, ensemble). Un objet a un identifiant, une valeur donnant son état et des opérations applicables. Il appartient alors à une classe qui est définie par son type d'objet et ses opérations.

L'héritage multiple est supporté, les conflits de noms sont résolus par renommage explicite. O2 supporte aussi la liaison dynamique. Contrairement aux autres SGBDO qui supporte l'héritage structurel, O2 offre une sémantique d'héritage plus riche basée sur l'inclusion de type. L'héritage par inclusion de type est particulièrement intéressant pour les classes collections.

L'unité de définition de données est le schéma qui comprend les définitions de types, de classes, d'opérations et de noms persistants. La notion de schéma O2 permet de regrouper les classes pour mieux contrôler la visibilité du graphe de classes. Un schéma correspond alors à un module et peut importer ou exporter des définitions depuis ou vers d'autres schémas.

Le modèle de persistance est par référence. Il permet à tout objet ou toute valeur O2 de persister indépendamment de son type. La persistance est obtenue par attachement à une racine persistante, déclarée dans le schéma par un nom global associé à une valeur ou à un objet O2.

O2 supporte des interfaces avec des langages comme C ou C++. Le langage O2C est essentiellement une intégration de O2 dans C qui sert alors à l'écriture des opérations.

En revanche, le couplage avec C++ repose sur un mécanisme d'import/export de classes O2. L'outil `O2link` convertit une classe C++ en une classe O2 et génère les opérations pour créer (`O2_new`), lire (`O2_read`) et écrire (`O2_write`) de façon transparente des objets C++ dans une classe O2. L'outil `ExportToC++` fait l'opération inverse. Il convertit une classe O2 en une classe C++ et ajoute les opérations pour créer, lire et écrire les objets C++. De plus O2 supporte le langage de requête OQL de l'ODMG qui peut être appelé depuis O2C ou C++ ou en mode interactif.

13.8. Conclusion

Nous avons donné les fonctions d'un SGBDO. Par comparaison aux SGBD relationnels, les SGBDO

- offrent un plus grand niveau d'abstraction ;
- peuvent gérer les opérations partagées comme les données et supportent les objets complexes grâce à un modèle plus riche qui combine les concepts des modèles à objets et des modèles sémantiques ;
- apportent une solution génériques aux applications complexes qui étaient jusqu'alors supportées par des systèmes dédiés.

Les SGBDO offrent les fonctions de persistance aux langages à objets en facilitant l'intégration du langage de bases de données et du langage de programmation. La persistance des objets s'obtient par une technique d'attribution de la persistance par héritage ou par référence ou une combinaison des deux. Elle s'accompagne d'opérations algébriques de manipulation des collections et de facilité d'évolution de schéma.

Deux approches à la définition d'un SGBDO ont émergé : l'approche relationnel-objet avec le standard SQL :1999 et l'approche orientée objet avec le standard ODMG.

- l'approche relationnel-objet enrichit le modèle relationnel et SQL avec un langage de types pour la définition d'objets complexes avec sous-typage. L'avantage de cette approche est la compatibilité ascendante avec SQL standard et sa traduction dans une algèbre pour objets complexes. Cependant, comme pour SQL, l'intégration avec un langage de programmation implique des conversions de types et de données et complique la programmation (deux langages différents). Un nouveau standard est en cours de finalisation : SQL :2003.
- l'approche orienté-objet est caractérisée par le standard ODMG compatible avec l'OMG. L'ODMG définit un modèle et des interfaces indépendantes de tout langage de programmation pour traiter des objets persistants (ODL, OML et OQL). Ces interfaces s'adaptent à des langages à objets sans causer de dysfonctionnement. OQL est langage de requêtes à objets proche mais différent de SQL. Cependant un

13. Bases de données objets

groupe de travail a été établi pour étudier l'intégration de OQL dans SQL :1999.

A. Utilisation du SGBD PostgreSQL

PostgreSQL [35] est un SGBD relationnel¹ libre (cf. [42]). Il est historiquement issu de POSTGRES, un SGBD développé à l'Université de Californie à Berkeley. La documentation de PostgreSQL se trouve en ligne [36]. Une forte communauté d'utilisateurs de PostgreSQL est présente sur Internet, voir par exemple [13] pour la communauté francophone. On remarquera également que PostgreSQL est utilisé non seulement dans des projets libres (Creative Commons, Debian), mais également dans l'industrie plus traditionnelle ou des gros organismes étatiques : Skype, University of California at Berkeley (évidemment...), *State Department* américain, Fujitsu, Cisco (cf. [35], onglet « About », puis « Featured users » ainsi que l'onglet « Témoignages » dans [13]). Nous allons présenter rapidement dans ce qui suit l'utilisation de ce SGBD.

Les exemples qui seront utilisés ici fonctionnent au Centre Informatique de SUPAERO. On pourra se référer à [19] pour plus d'informations sur l'installation des différents logiciels.

Nous choisirons la base `bdmexico86` disponible au Centre Informatique de SUPAERO comme exemple dans ce qui suit. Le propriétaire de cette base est l'utilisateur `michel` dont le mot de passe est `platini`.

Pour des raisons de lisibilité, les captures d'écran ont été placées en fin de chapitre.

A.1. Respect du standard SQL

Un des problèmes les plus importants avec les SGBD est de vérifier qu'ils respectent la norme SQL (au moins SQL-92...). PostgreSQL étant une implantation libre et donc dépendant de contributeurs non rémunérés pour la plupart, certaines fonctionnalités de SQL ne sont pas disponibles. Pour en avoir une liste complète, on peut se référer à [36] (appendice « *SQL conformance* »).

On peut toutefois citer quelques limitations :

- pas de choix des niveaux d'isolation des transactions ;
- pas d'assertions ;
- pas de requêtes dans les clauses **CHECK** (pour ne pas pénaliser le système : si l'on base une contrainte sur une requête, il faut vérifier les contraintes à chaque fois qu'un des éléments choisi par la requête est changé) ;
- pas de type de données complexe : BLOB...
- une partie des fonctionnalités des types XML ;
- restrictions dans les *triggers* et syntaxe différente (cf. section suivante)
- certains langages ne sont pas disponibles : ADA, COBOL etc.

1. En fait PostgreSQL est même un SGBD relationnel objet.

A.2. Petits détails de syntaxe et autres

PostgreSQL propose d'autres noms de type que ceux vus en cours : **character varying** à la place de **VARCHAR**, **character** à la place de **CHAR** en particulier. Les types vus dans le cours seront toujours disponibles comme alias, on peut donc les utiliser.

Comme vu dans le chapitre 8, les chaînes de caractères sont encadrées par des guillemets anglais simples : 'chaîne' par exemple. On peut également protéger les noms d'attributs, de relation etc., surtout s'ils comportent des espaces ou des caractères spéciaux, en utilisant des guillemets anglais doubles : **SELECT "mon attribut"FROM "ma relation";** par exemple.

A.3. Écriture de fonction et de *trigger*

A.3.1. Fonctions dans PostgreSQL

Comme nous l'avons vu dans le chapitre 10, il est possible de stocker des fonctions ou des procédures « côté serveur », i.e. dans le SGBD. Ces fonctions peuvent être ensuite appelées dans une requête pour simplifier un calcul par exemple.

Pour créer une fonction sur le serveur, on utilise le mot-clé **CREATE FUNCTION**. On préférera utiliser la commande **CREATE OR REPLACE FUNCTION** qui permet de créer une fonction ou de la remplacer si elle existe déjà.

Syntaxe (création d'une fonction) :

```
CREATE OR REPLACE FUNCTION nomFunction(par1 Type1, ..., parn Typen)
    RETURN TypeRetour AS $NOMFONCTIONS$

    -- corps de la fonction

$NOMFONCTIONS$ LANGUAGE plpgsql;
```

□

Comme dans tout langage de programmation, on donne un nom à la fonction, on précise ses paramètres et leurs types² et un type de retour. Le mot **NOMFONCTION** est utilisé en interne par PostgreSQL et importe peu.

Pour écrire le corps de la fonction que l'on est en train de créer, il faut choisir le langage que l'on va utiliser pour écrire cette fonction. PostgreSQL en propose plusieurs : PL/pgSQL, PL/Tcl, PL/Perl, et PL/Python. Nous allons choisir ici de travailler avec PL/pgSQL, puisqu'il se rapproche beaucoup de SQL. Il faut donc déclarer dans la base de données l'utilisation de ce langage³ :

2. On peut également ne pas donner de nom aux paramètres et les récupérer avec des variables spéciales, mais cela n'est pas conseillé.

3. Attention, il faut avoir des droits de super-utilisateur pour faire cela (essentiellement pour des raisons de sécurité). Les étudiants n'auront donc pas à utiliser cette commande normalement.

Syntaxe (déclaration de l'utilisation de PL/pgSQL) :

```
CREATE LANGUAGE plpgsql;
```

□

Un programme écrit en PL/pgSQL est composé de *blocs* définis comme suit :

Syntaxe (blocs en PL/pgSQL) :

```
DECLARE
  -- declaration de variables
BEGIN
  -- corps du bloc
END;
```

□

Le bloc permet donc de déclarer un certain nombre de variables, puis d'effectuer des opérations. On dispose d'un certain nombre d'opérateurs, dont les plus utilisés sont présentés dans le tableau. On peut bien sûr utiliser des blocs de qualification (cf. exemple ci après) pour instancier des variables. On se référera à [36] pour plus de renseignements.

Opérateurs	Signification
:=	affectation
RETURN	retour d'une expression
IF ... THEN ... ELSE ... END IF;	conditionnelle
LOOP ... END LOOP;	boucle sans condition se finissant par EXIT ou RETURN
WHILE cond LOOP ... END LOOP;	boucle tant que
FOR var IN ... LOOP ... END LOOP;	boucle pour
FOR var IN requete LOOP ... END LOOP;	boucle pour utilisant une requête : var prend successivement les valeurs de la requête et est habituellement déclarée avec le type record

TABLE A.1. – Opérateurs « classiques » de PL/pgSQL

Exemple A.3.1 : Supposons que nous disposions de deux numéros d'identification d'étudiants et que nous voulions connaître quel est celui qui a la meilleure moyenne de notes contenues dans une table `Notes(id, module, note)`⁴. On veut donc écrire une fonction `bestStudent` qui nous renvoie le numéro d'identification du meilleur étudiant. Voici la fonction correspondante écrite avec PL/pgSQL :

4. Évidemment, tout cela peut être fait très simplement avec l'opérateur `MAX...`. De plus, nous utilisons ici des variables intermédiaires qui ne servent à rien.

A. Utilisation du SGBD PostgreSQL

```
CREATE OR REPLACE FUNCTION bestStudent(id1 INTEGER, id2 INTEGER)
    RETURNS INTEGER AS $BESTSTUDENT$
DECLARE
    moy1 REAL;
    moy2 REAL;
BEGIN
    moy1 := AVG(note) FROM Notes WHERE id = id1;
    moy2 := AVG(note) FROM Notes WHERE id = id2;

    IF (moy1 > moy2) THEN
        RETURN id1;
    ELSE
        RETURN id2;
    END IF;
END;
$BESTSTUDENT$ language plpgsql;
```

A.3.2. Syntaxe de l'écriture d'un *trigger*

Nous allons revenir sur l'exemple de *trigger* proposé dans la section 9.3.2 pour décrire comment le construire avec PostgreSQL (la syntaxe diffère et il faut écrire une fonction spéciale). Rappelons le *trigger* proposé :

```
CREATE TRIGGER TriggerMoyenneLogiciel
AFTER UPDATE OF prix ON Logiciel
REFERENCING
    OLD TABLE AS AncienneTable,
    NEW TABLE AS NouvelleTable
FOR EACH STATEMENT
WHEN (10000 > (SELECT AVG(prix) FROM Logiciel))
BEGIN
    DELETE FROM Logiciel
    WHERE (nom, classe, conceptions, provenance, revendeur,
        IN NouvelleTable);
    INSERT INTO Logiciel
        (SELECT * FROM AncienneTable);
END;
```

Tout d'abord, avec PostgreSQL le code exécuté dans un *trigger* provient obligatoirement d'une *fonction utilisateur*, que nous venons de voir, renvoyant un type *trigger*. De plus, on ne peut pas utiliser des variables pour référencer l'ancien tuple et le nouveau, les noms seront obligatoirement **NEW** et **OLD**. Enfin, il ne peut pas y avoir de clause **WHEN**, il faudra utiliser la fonction utilisateur pour faire les vérifications. Supposons que l'on dispose d'une fonction `trigger_moyenne()`, le *trigger* va alors s'écrire :

```
CREATE TRIGGER TriggerMoyenneLogiciel
AFTER UPDATE ON Logiciel
FOR EACH STATEMENT
EXECUTE PROCEDURE trigger_moyenne();
```

Pour écrire la fonction `trigger_moyenne()`, on utilise PL/pgSQL comme vu précédemment⁵ :

```
CREATE OR REPLACE FUNCTION trigger_moyenne() RETURNS trigger AS $trigger_moyenne$
DECLARE
    moy FLOAT;
BEGIN
    moy := AVG(prix) FROM Logiciel;
    IF (moy > 10000) THEN
        RAISE EXCEPTION 'problem with average price';
    END IF;
    RETURN NULL;
END;
$trigger_moyenne$ LANGUAGE plpgsql;
```

On peut alors vérifier que

```
INSERT INTO Logiciel VALUES ('Oracle10', 'SGBD', 'Oracle', 'USA', 'Oracle', 15000);
```

ne fonctionne pas, mais que :

```
INSERT INTO Logiciel VALUES ('Oracle10', 'SGBD', 'Oracle', 'USA', 'Oracle', 15000),
('PostgreSQL2', 'SGBD', 'Oracle', 'USA', 'Oracle', 0);
```

fonctionne correctement.

On utilise ici l'instruction `RAISE EXCEPTION` pour pouvoir lever une exception si l'insertion n'est pas correcte.

Une fonction *trigger* peut retourner :

- `NULL` pour signaler que l'insertion en cours doit être ignorée (la levée d'une exception produit le même résultat) ;
- une ligne qui a la structure attendue par la relation.

On consultera bien évidemment [36] pour plus de détails.

A.4. pgAdmin

pgAdmin [33] est un outil graphique permettant d'administrer et d'interagir avec PostgreSQL. Il est très souple et complet (il permet par exemple d'avoir une explication sur les requêtes réellement faites lors d'une interrogation de la base). On le lance soit en ligne de commande depuis un terminal avec la commande `pgadmin3`, soit depuis les menus de votre environnement s'il y est disponible. On obtient alors une fenêtre identique à celle présentée sur la figure A.1.

5. On n'utilise pas ici les variables `NEW` et `OLD` car elles n'ont pas de sens pour un *trigger* par *statement*

A. Utilisation du SGBD PostgreSQL

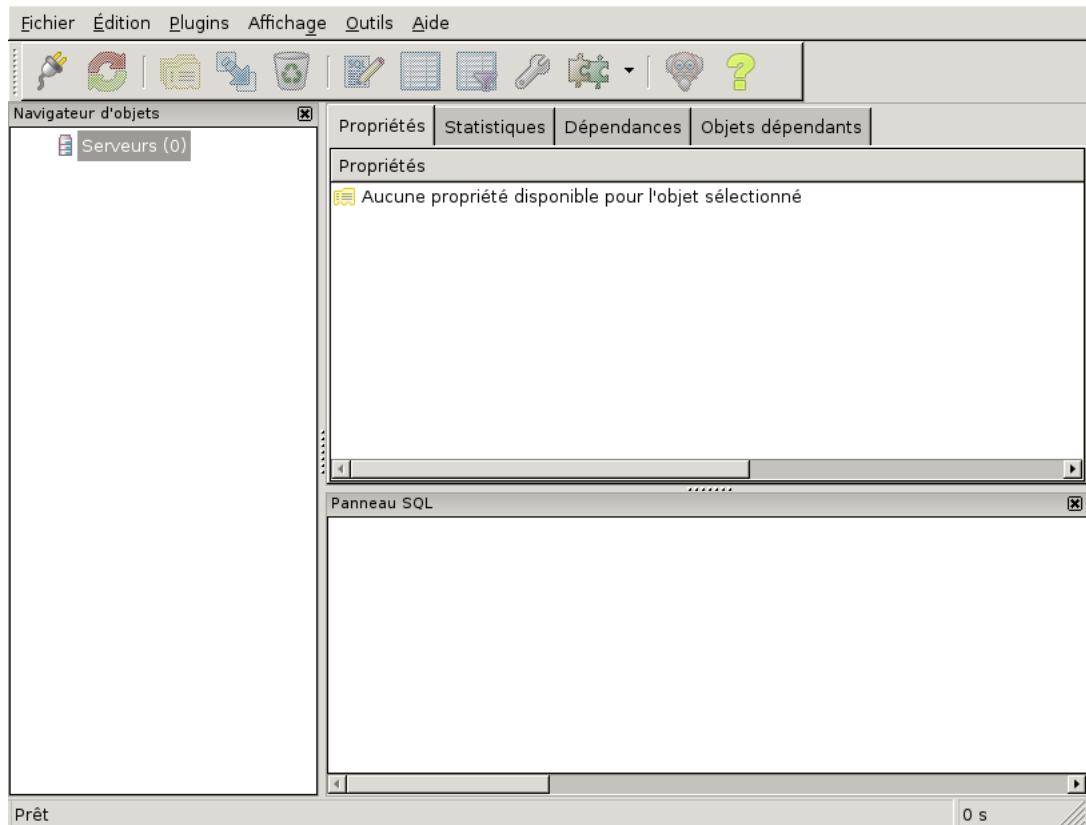


FIGURE A.1. – La fenêtre initiale de pgAdmin

A.4.1. Connexion à une base de données

Pour se connecter à une base de données, il suffit de cliquer sur l'icone « prise de courant ». On obtient alors une fenêtre de dialogue que l'on complète avec les informations nécessaires. La figure A.2 présente le dialogue rempli pour une connexion à la machine `postgresql` pour l'utilisateur `michel`.

On peut alors voir quelles sont les bases disponibles sur le serveur comme montré sur la figure A.3. On ne peut bien évidemment accéder qu'aux bases sur lesquelles on a les droits nécessaires.

On peut alors voir les détails de la base `bdmexico86` comme montré sur la figure A.4. On remarquera que la base est organisée en *schémas* et que le schéma par défaut est `public`⁶. On trouvera donc sous ce schéma les relations, fonctions, *triggers*, ... dont on a besoin.

Enfin, il est toujours possible d'utiliser les menus contextuels sur un objet particulier via le clic-droit de la souris. On peut ainsi visualiser plus facilement le contenu d'une

6. La notion de schéma n'est pas abordée en cours, mais il s'agit intuitivement d'un mécanisme permettant une organisation logique des bases de données (un espace de nommage, comme les paquetages en Java par exemple) et le partage des composants des bases de données entre plusieurs utilisateurs.

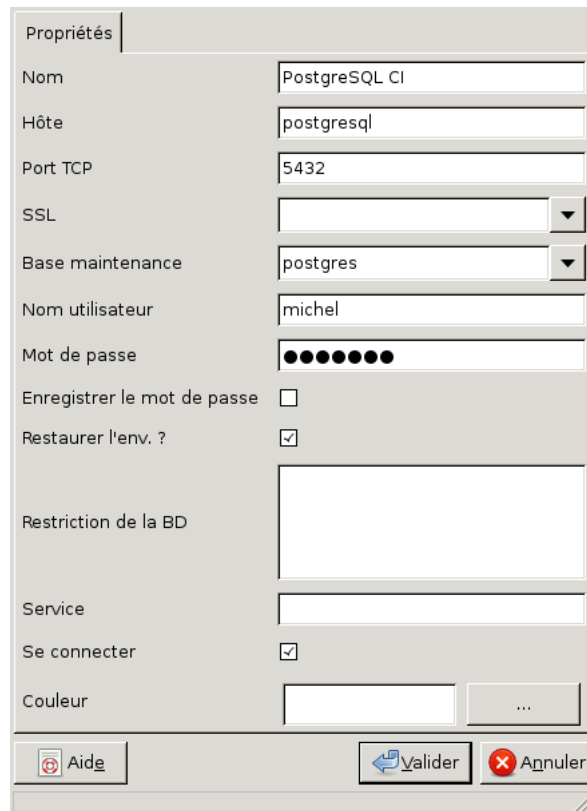


FIGURE A.2. – Dialogue de connexion à la machine `postgresql` pour l'utilisateur `michel`

relation via le sous-menu « Afficher les données » disponibles sur la table `Match` comme présenté sur la figure A.5 (on remarquera qu'une des icônes principales symbolisant une table permet également de le faire).

A.4.2. Écriture d'une requête SQL

Pour écrire une requête SQL, il suffit d'utiliser l'éditeur de script SQL. Celui-ci permet d'écrire des commandes SQL sur la base de données en disposant entre autres de la coloration syntaxique. On peut également sauvegarder ses requêtes dans un fichier. La figure A.6 présente l'éditeur de script après exécution d'une requête. On peut même demander quel est le plan d'exécution de la requête choisi par le compilateur de PostgreSQL. La figure A.7 présente un tel plan. Ceci peut être utile pour bien optimiser ses requêtes.

Remarque (importante) : Il est plus pratique d'utiliser l'éditeur de requêtes SQL, même pour créer des tables par exemple (alors que l'on dispose de dialogues graphiques pour le faire) : on peut plus facilement conserver son code et le modifier, surtout dans un travail préliminaire. □

A. Utilisation du SGBD PostgreSQL

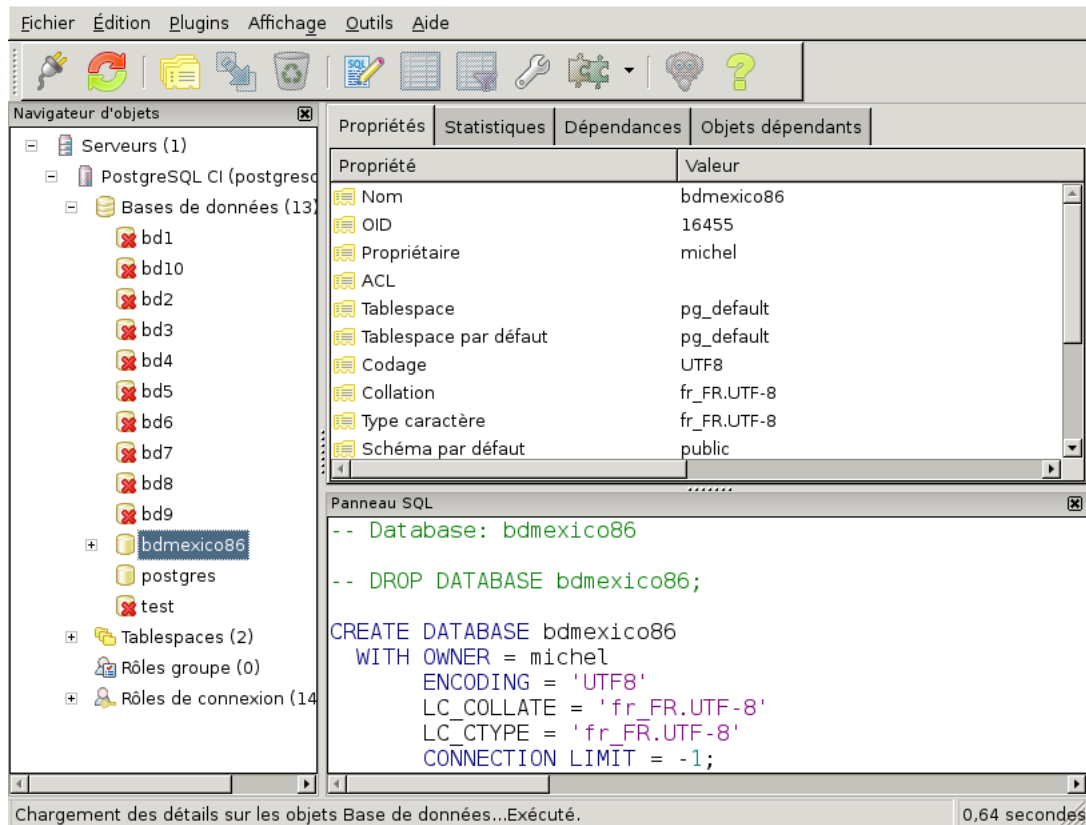


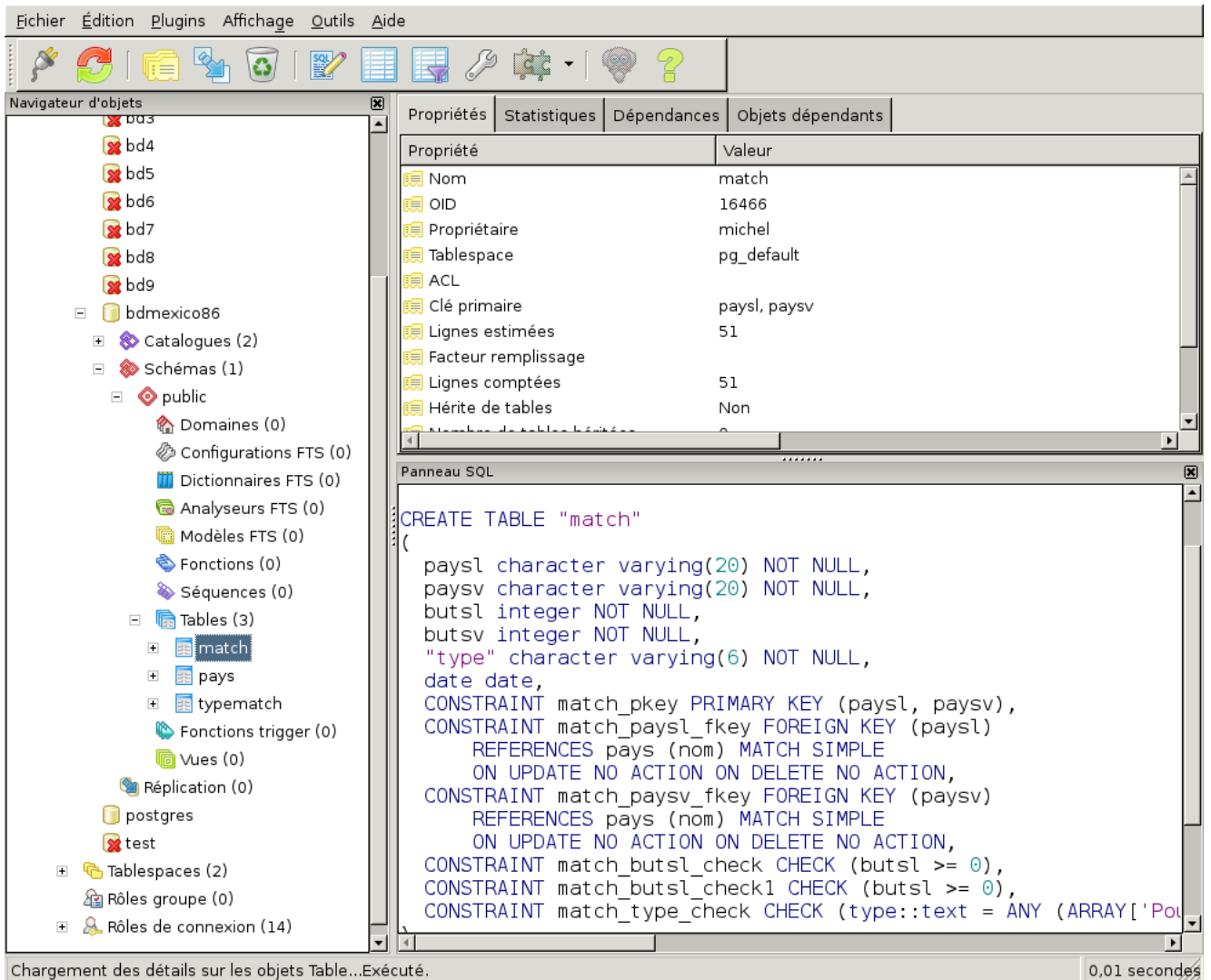
FIGURE A.3. – Les bases de données disponibles sur le serveur et quelques détails sur bdmexico86

A.5. Utilisation interactive via le terminal

PostgreSQL fonctionne en mode client/serveur : un serveur est exécuté sur une machine. Les différents utilisateurs utilisent des clients (du simple moniteur en ligne aux langages de programmation 10) qui transmettent les commandes SQL au serveur.

Le moniteur `psql` peut être exécuté par tous les utilisateurs PostgreSQL ayant accès à une base de données. Les autres programmes sont réservés à l'administrateur PostgreSQL et aux utilisateurs disposant de droits avancés (création d'utilisateur et de base de données).

Les différents programmes et en particulier le moniteur interactif ont besoin d'accéder au serveur. Le nom de la machine sur laquelle celui-ci est exécuté peut être fourni en argument dans la ligne de commande. On peut également utiliser la variable d'environnement `PGHOST`. Par défaut, le serveur sera cherché sur la machine locale `localhost`.

FIGURE A.4. – Détails de la base `bdmexico86`

A.5.1. Commandes réservées aux utilisateurs privilégiés

Ces commandes sont exécutées dans un terminal UNIX. Il faut disposer de droits privilégiés pour pouvoir les exécuter.

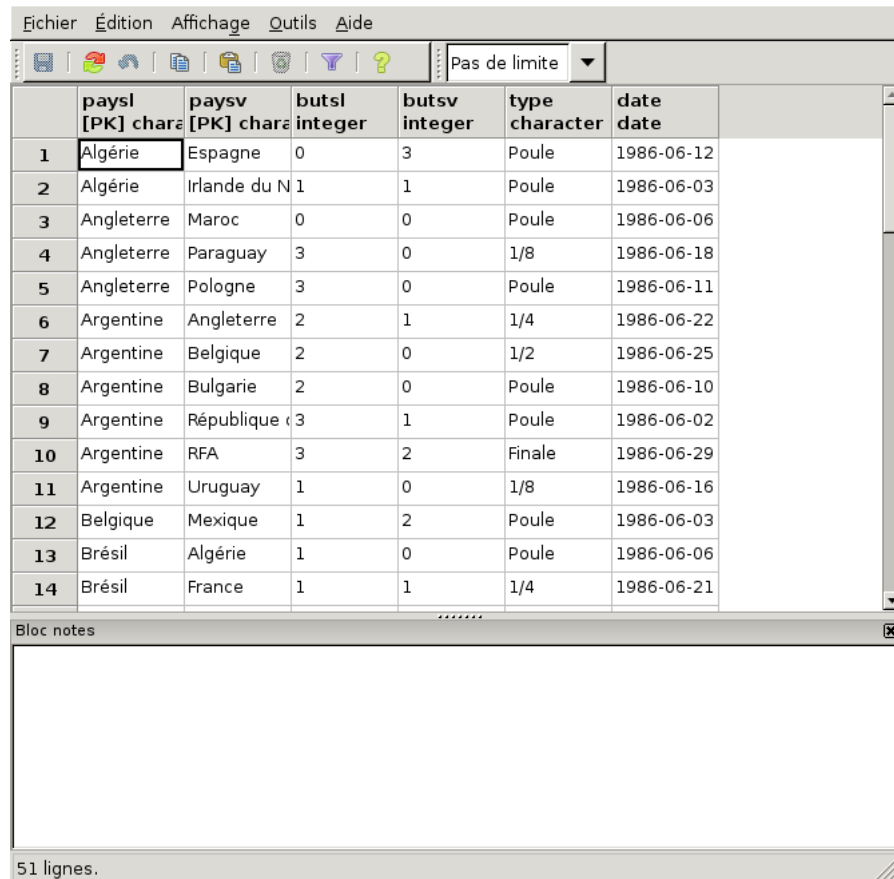
On peut créer et supprimer une base de données (il s'est également possible d'imposer un mot de passe et un utilisateur responsable de la base) :

Syntaxe (création et destruction d'une base) :

```
createdb nom_base
```

```
dropdb nom_base
```

A. Utilisation du SGBD PostgreSQL



	paysl [PK] chara	paysv [PK] chara	butsl integer	butsv integer	type character	date date
1	Algérie	Espagne	0	3	Poule	1986-06-12
2	Algérie	Irlande du N	1	1	Poule	1986-06-03
3	Angleterre	Maroc	0	0	Poule	1986-06-06
4	Angleterre	Paraguay	3	0	1/8	1986-06-18
5	Angleterre	Pologne	3	0	Poule	1986-06-11
6	Argentine	Angleterre	2	1	1/4	1986-06-22
7	Argentine	Belgique	2	0	1/2	1986-06-25
8	Argentine	Bulgarie	2	0	Poule	1986-06-10
9	Argentine	République c	3	1	Poule	1986-06-02
10	Argentine	RFA	3	2	Finale	1986-06-29
11	Argentine	Uruguay	1	0	1/8	1986-06-16
12	Belgique	Mexique	1	2	Poule	1986-06-03
13	Brésil	Algérie	1	0	Poule	1986-06-06
14	Brésil	France	1	1	1/4	1986-06-21

Bloc notes

51 lignes.

FIGURE A.5. – Contenu de la relation Match

□

On peut créer et supprimer des utilisateurs PostgreSQL :

Syntaxe (création et suppression d'utilisateur) :

```
createuser nom_user
```

```
dropuser nom_user
```

□

Vous pouvez utiliser les manuels en ligne (commande `man dropuser` par exemple) pour obtenir plus d'informations sur les options disponibles.

A.5.2. Le moniteur interactif

Le moniteur interactif permet d'ouvrir une session sur une base de données gérée par PostgreSQL.

Éditeur SQL Constructeur graphique de requêtes

```
SELECT * FROM Match;
```

Bloc notes
Quelques notes sur la requête en cours

Panneau sortie

Sortie de données Expliquer (Explain) Messages Historique

	paysl character	paysv character	butsl integer	butsv integer	type character	date date
1	Bulgarie	Italie	1	1	Poule	1986-05-31
2	Argentine	République	3	1	Poule	1986-06-02
3	Italie	Argentine	1	1	Poule	1986-06-05
4	République	Bulgarie	1	1	Poule	1986-06-05
5	République	Italie	2	3	Poule	1986-06-10
6	Argentine	Bulgarie	2	0	Poule	1986-06-10
7	Belgique	Mexique	1	2	Poule	1986-06-03
8	Paraguay	Irak	1	0	Poule	1986-06-04

OK. Unix Ligne 1 Col 21 Caract. 51 lignes. 90 ms

FIGURE A.6. – Le résultat d’une requête sur la relation `Match`

Syntaxe :

```
psql -h hote nom_base nom_user
```

□

Exemple A.5.1 : Pour se connecter en tant qu’utilisateur `eleve2` sur la base `bd2`, on utilise la syntaxe suivante :

```
psql -h serv-sun1 bd2 eleve2
```

On obtient alors une invite de commandes. On peut envoyer des requêtes SQL directement à la base en utilisant le moniteur. Il existe toutefois quelques commandes spéciales qui sont résumées dans le tableau

A. Utilisation du SGBD PostgreSQL

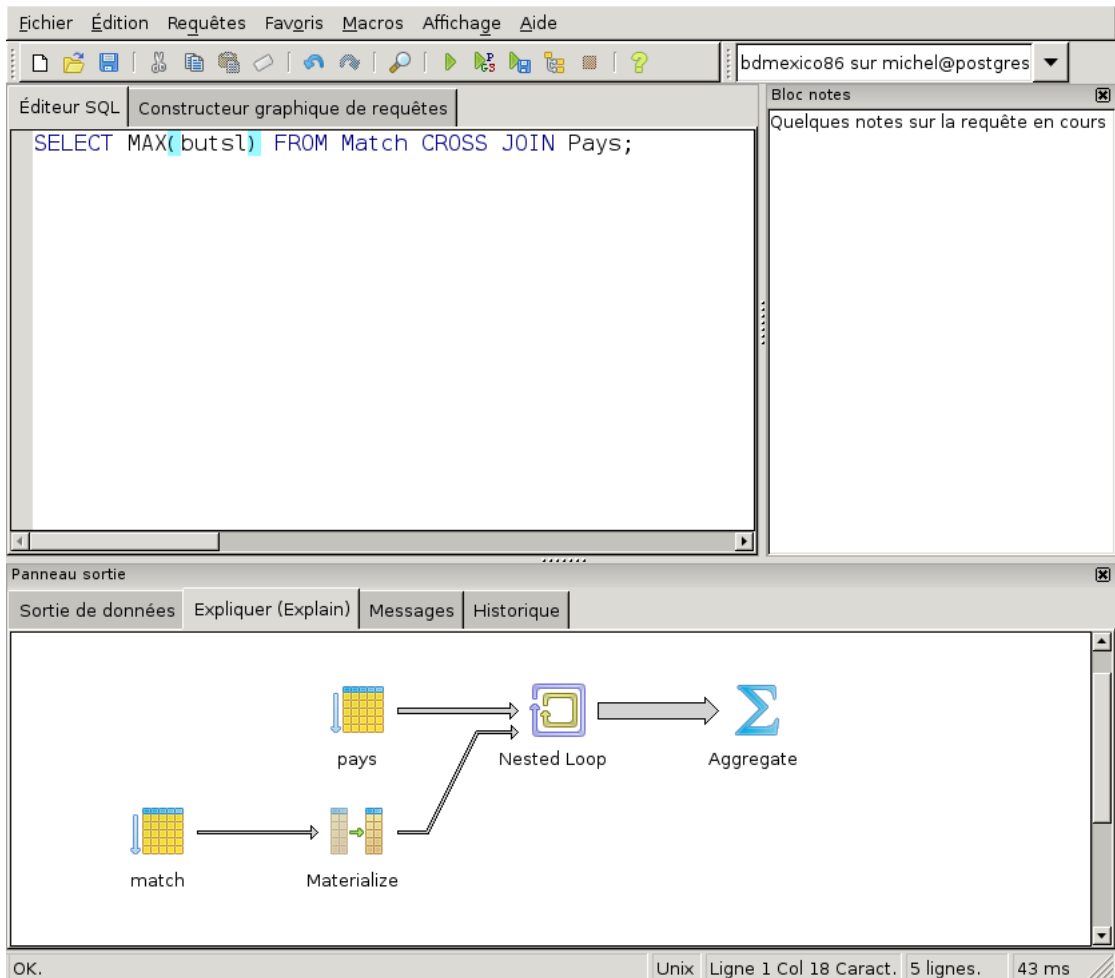


FIGURE A.7. – Une explication détaillée des opérations effectuées par PostgreSQL

Attention, le moniteur interactif n'est pas forcément très facile à utiliser. Il vaut mieux utiliser pgAdmin présenté en section A.4 si l'on n'est pas à l'aise avec la ligne de commande.

A.6. Utilisation avec Java

Pour utiliser PostgreSQL avec Java, on pourra se référer au chapitre 10. Les principes qui y sont donnés sont génériques et valables pour n'importe quel système de gestion de bases de données. On utilisera pour PostgreSQL un driver JDBC de type 4 [37]. Comme il s'agit d'un driver de type 4, il s'agit d'un ensemble de classes Java regroupées sous la forme d'un fichier JAR. Il suffit donc d'inclure ce fichier dans son CLASSPATH pour pouvoir utiliser le driver. On pourra également se référer à [19] pour trouver un exemple simple.

Commande	Effet
<code>\?</code>	liste les commandes <code>psql</code> possibles
<code>\dt</code>	liste les tables existant dans la base
<code>\h</code>	affiche l'aide sur les commandes SQL
<code>\h command</code>	affiche l'aide de <code>command</code>
<code>\i file.sql</code>	interprète les commandes SQL de <code>file.sql</code>
<code>\q</code>	quitte le client <code>psql</code>

TABLE A.2. – Commandes du moniteur

Bibliographie

- [1] W. ARMSTRONG. « Dependency structures of data base relationships ». In : *Proceedings of International Federation for Information Processing Congress (IFIP)*. 1974, p. 580–583.
- [2] C. W. BACHMAN. « The Programmer as Navigator ». In : *Communications of the ACM* 16.11 (1973). ACM Turing Award lecture, p. 653–658.
- [3] F. BANCILHON, C. DELOBEL et P.C. KANELLAKIS, éd. *Building an Object-Oriented Database System, The Story of O2*. Morgan Kaufmann, 1992.
- [4] G. BOOCH, J. RUMBAUGH et I. JACOBSON. *The Unified Modeling Language reference manual*. Addison-Wesley, 2004.
- [5] R. G. G. CATTELL et al., éd. *The Object Data Standard : ODMG 3.0*. Morgan Kaufmann, 2000.
- [6] J. CELKOS. *Column : SQL for Smarties*. DataBase Management Systems Journal Online. URL : <http://www.dbmsmag.com/artin301.html#A000009%22%22>.
- [7] J. CELKOS. *SQL for Smarties : Advanced SQL Programming*. 2^e éd. Morgan Kaufmann, 1999.
- [8] J. CELKOS. *SQL Puzzles and Answers*. Morgan Kaufmann, 1997.
- [9] P. CHEN. « The Entity-Relationship Model - Toward a Unified View of Data ». In : *ACM Transactions on Database Systems* 1.1 (1976), p. 9–36. URL : <http://bit.csc.lsu.edu/~chen/pdf/erd.pdf>.
- [10] P. M. CHEN et al. « RAID : High-Performance, Reliable Secondary Storage ». In : *ACM Computing Surveys* 26.2 (1994), p. 145–186. URL : <http://citeseer.ist.psu.edu/244988.html>.
- [11] E. J. CODD. « A relational model of data for large shared data banks ». In : *Communications of the ACM* 13.6 (1970), p. 377–387. URL : <http://www.acm.org/classics/nov95/toc.html>.
- [12] E. J. CODD. *The relational model for database management : version 2*. Addison-Wesley Publishing, 1990.
- [13] *Communauté francophone de PostgreSQL*. URL : <http://www.postgresql.fr>.
- [14] DATAMODEL.ORG. *Rules of Data Normalization*. URL : <http://www.datamodel.org/NormalizationRules.html>.
- [15] *DB2 for z/OS and OS/390 : Ready for Java*. IBM Redbooks. URL : <http://www.redbooks.ibm.com/abstracts/sg246435.html>.

Bibliographie

- [16] *db4objects : native Java and .NET open source object database engine*. URL : <http://www.db4o.com/>.
- [17] Ulrich DREPPER. *What Every Programmer Should Know About Memory*. 21 nov. 2007. URL : <http://www.akkadia.org/drepper/cpumemory.pdf>.
- [18] H. GARCIA-MOLINA, J.D. ULLMAN et J. WIDOM. *Database Systems : the Complete Book*. Prentice Hall, 2002.
- [19] C. GARION. *Logiciels utilisés à SUPAERO : installation et utilisation*. URL : <http://personnel.supaero.fr/garion-christophe/CI>.
- [20] *GemStone*. URL : <http://www.gemstone.com/>.
- [21] G. A. GIBSON, J. S. VITTER et J. WILKES. « Strategic directions in storage I/O issues in large-scale computing ». In : *ACM Computing Surveys* 28.4 (1996), p. 779–793. URL : <http://portal.acm.org/citation.cfm?id=242223.242300&coll=GUIDE&dl=GUIDE&CFID=32165634&CFTOKEN=12524069>.
- [22] J. N. GRAY et A. REUTER. *Transaction processing : concepts and techniques*. Morgan Kaufmann, 1993.
- [23] R. GREHAN. *Complex Object Structures, Persistence, and db4o*. URL : <http://www.db4o.com/about/productinformation/whitepapers/db4o%20Whitepaper%20-%20Complex%20Object%20Structures.pdf>.
- [24] T. HAERDER et A. REUTER. « Principles of transaction-oriented database recovery - a taxonomy ». In : *Computing Surveys* 15.4 (1983), p. 287–317.
- [25] *Java Data Objects (JDO)*. URL : <http://java.sun.com/products/jdo/index.jsp>.
- [26] J. S. KNOWLES et D. M. R. BELL. « The CODASYL model ». In : *Databases - Role and Structure*. Sous la dir. de P. M. STOCKER, P. M. D. GRAY et M. P. ATKINSON. CUP, 1984.
- [27] P. MATHIEU. *Des bases de données à l'internet*. In French. Vuibert, 2000.
- [28] J. MELTON. *Advanced SQL :1999 : Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann, 2003.
- [29] B. MEYER. *Object-Oriented Software Construction*. 2^e éd. Prentice Hall, 1997.
- [30] Organisation Internationale de NORMALISATION. *ISO/IEC 9075-10 :2008 Technologies de l'information – Langages de base de données – SQL – Partie 10 : Liaisons de langage objet (SQL/OLB)*.
- [31] *ObjectStore*. URL : <http://www.progress.com/realtime/products/objectstore/index.ssp>.
- [32] T. W. OLLE. *The CODASYL Approach to Data Base Management*. Wiley, 1978.
- [33] *pgAdmin*. URL : <http://www.pgadmin.org/>.

- [34] E. PINHEIRO, W.-D. WEBER et L. A. BARROSO. « Failure Trends in a Large Disk Drive Population ». In : *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*. 2007. URL : http://labs.google.com/papers/disk_failures.pdf.
- [35] *PostgreSQL*. URL : <http://www.postgresql.org/>.
- [36] *PostgreSQL 8.4 Documentation*. URL : <http://www.postgresql.org/docs/8.4/interactive/>.
- [37] *PostgreSQL JDBC Driver*. URL : <http://jdbc.postgresql.org/>.
- [38] B. REINWALD et al. « Storing and using objects in a relational database ». In : *IBM Systems Journal* 35.2 (1996), p. 172–191. URL : <http://www.research.ibm.com/journal/sj/352/reinwald.pdf>.
- [39] V. SRINIVASAN et D. T. CHANG. « Object persistence in object-oriented applications ». In : *IBM Systems Journal* 36.1 (1997). URL : <http://www.research.ibm.com/journal/sj/361/srinivasan.html>.
- [40] A. TANENBAUM. *Architecture de l'ordinateur*. Dunod, 2006.
- [41] A. TANENBAUM. *Systèmes d'exploitation*. 2^e éd. In French. Dunod, 2003.
- [42] *The GNU General Public License*. URL : <http://www.gnu.org/licenses/gpl.html>.
- [43] J. S. VITTER. « External memory algorithms ». In : *Proceedings of the 7th Annual ACM Symposium on Principles of Database Systems*. 1998, p. 119–128.
- [44] WIKIPEDIA. *Object-Relational Impedance Mismatch*. URL : http://en.wikipedia.org/wiki/Object-Relational_Impedance_Mismatch.