UNIVERSITÉ LUMIÈRE LYON 2
ÉCOLE DOCTORALE INFORMATIQUE ET MATHÉMATIQUES

**THÈSE**

pour obtenir le grade de

**DOCTEUR EN INFORMATIQUE**

présentée par

**Marouane Hachicha**

# Modélisation de hiérarchies complexes dans les entrepôts de données XML et traitement des problèmes d'additivité dans l'analyse en ligne XOLAP

préparée au sein du Laboratoire ERIC - Université Lumière Lyon 2




**sous la direction du**

**Pr Jérôme Darmont**

Soutenue publiquement le **26 Novembre 2012** devant le jury :

| | | |
|---|---|---|
| Melle Sihem Amer-Yahia | DR, Laboratoire d'Informatique de Grenoble | (rapporteur) |
| M. Olivier Teste | Pr, Université Toulouse 2 | (rapporteur) |
| Mme Maryvonne Miquel | MCF HDR, INSA Lyon | (examinateur) |
| M. Esteban Zimányi | Pr, Université Libre de Bruxelles - Belgique | (examinateur) |
| M. Jérôme Darmont | Pr, Université Lumière Lyon 2 | (directeur de thèse) |

# Résumé

Depuis son apparition en 1998, le langage XML (*eXtensible Markup Language*) est devenu un standard pour la modélisation et l'échange de données. En effet, XML permet de modéliser des structures de données qui ne sont pas facilement représentées dans les systèmes relationnels. Dans ce contexte, les entrepôts de données XML représentent aujourd'hui la base de plusieurs applications décisionnelles qui exploitent des données hétérogènes (peu structurées et provenant des sources multiples) aux structures complexes comme par exemple des hiérarchies complexes.

Dans ce mémoire, nous proposons une nouvelle solution XOLAP (XML-OLAP) en temps réel qui traite les problèmes d'additivité dus aux hiérarchies complexes. Tout d'abord, nous proposons un nouveau modèle de données : les arbres de données multidimensionnels, qui permet de modéliser les faits, les dimensions, les mesures et les hiérarchies complexes d'un entrepôt de données XML. Pour pouvoir interroger les arbres de données multidimensionnels, nous modélisons les requêtes utilisateur à l'aide de modèles d'arbre XML. Nous proposons ensuite un nouvel algorithme de regroupement et d'agrégation pour la résolution en temps réel des problèmes d'additivité dans les hiérarchies complexes. Nous généralisons enfin cet algorithme à un nouvel opérateur XOLAP de forage vers le haut (*roll-up*).

Finalement, nous validons nos propositions de manière expérimentale. Pour cela, nous étendons le banc d'essais XWeB en introduisant des hiérarchies complexes dans son schéma. La comparaison de notre approche à une approche de référence montre que la surcharge due à l'exécution en temps réel de notre approche est tout à fait acceptable et que nos algorithmes sont susceptibles de passer à l'échelle.

**Mots clés :** Entrepôts de données XML, XOLAP, Hiérarchies complexes, Agrégation, forage vers le haut, Problèmes d'additivité, Regroupement multiple.

# Abstract

Since its inception in 1998, the eXtensible Markup Language (XML) has emerged as a standard for data representation and exchange over the Internet. XML provides an opportunity for modeling data structures that are not easily represented in relational systems. In this context, XML data warehouses nowadays form the basis of several decision-support applications exploiting heterogeneous data (little structured and coming from various sources) bearing complex structures, such as complex hierarchies.

In this thesis, we propose a novel XOLAP (XML-OLAP) approach that automatically detects and processes summarizability issues at query time, without requiring any particular expertise from the user. Thus, at the logical level, we choose XML data trees, so-called multidimensional data trees, to model the multidimensional structures (facts, dimensions, measures and complex hierarchies) of XML data warehouses. In order to query multidimensional data trees, we model user queries as XML pattern trees. Then, we introduce a new aggregation algorithm to address summarizability issues in complex hierarchies. On the basis of this algorithm, we propose a novel XOLAP roll-up operator.

Finally, we experimentally validate our proposal and compare our approach with the reference approach for addressing summarizability issues in complex hierarchies. For this sake, we extend the XML warehouse benchmark XWeB with complex hierarchies to generate XML data warehouses with scalable complex hierarchies. The results of our experiments show that the overhead induced by managing hierarchy complexity at run-time is totally acceptable and that our approach is expected to scale up well.

**Keywords :** XML data warehouses, XOLAP, Complex hierarchies, Aggregation, Summarizability issues, Grouping.

# Remerciements

J'exprime tout d'abord tous mes remerciements à mon directeur de thèse, M. Jérôme Darmont, qui m'a donné la chance de suivre cette thèse, qui m'a soutenu et encouragé depuis mon stage de Licence en 2005, et jusqu'à la dernière minute de cette thèse de doctorat. Je l'en remercie très chaleureusement.

Je tiens aussi à remercier Melle Sihem Amer-Yahia et M. Olivier Teste, qui m'ont fait l'honneur de porter intérêt à ma thèse et d'en être les rapporteurs. Je remercie également Mme Maryvonne Miquel et M. Esteban Zimányi d'avoir accepter d'être les examinateurs de ce travail.

Je tiens également à exprimer mes remerciements aux membres du laboratoire ERIC, en particulier mes collègues de bureau : Anna, Emna, Oksana, Diala, Ahmed, Doulkifli, Hoa, Rashed et Sandro, avec qui j'ai partagé des moments inoubliables. Un merci très spécial à Chantola et Hadj avec qui j'ai eu le plaisir de travailler et de publier. Je voudrais également remercier le personnel administratif avec qui on peut partager un petit café bien chaud, et des discussions intéressantes surtout, de bon matin : Corinne, Delphine, Jennifer, Stéphanie, Valérie, Cyril et Julien.

Merci à Mme Fadila Bentayeb et M. Stéphane Lallich pour leurs encouragements et leurs conseils lors des différentes étapes la rédaction de ce mémoire. Merci également à tous les membres de notre groupe de travail, l'équipe SID. Je n'oublie pas de remercier les jeunes docteurs Cécile Favre, Kafil Hajlaoui et Ounas Asfari pour leurs conseils et leur soutien.

Lors de mes cinq années de thèse, j'ai eu le plaisir de découvrir le métier

d'enseignant dont j'espère continuer à exercer. Merci alors à mon responsable de monitorat, M. Omar Boussaid, ainsi qu'à mes responsables et collègues de l'IUT Lumière : Mme Aurélie Mercier, Mme Claire Lhomme, Mme Hélène Chanvillard et mon pote Anis Ben Khedher.

Un Très Grand Merci à mes amis, surtout ceux qui étaient présents à côté de moi ces dernières années de thèse : Luana, Maiana, Yulia & Sébastien, Pauline & Pierre (et leur beau bébé Martin), Samuel, Abdullah, Anouar, Bachir, Maher et Nader. 1000 Merci à Shéshé, ma meilleure amie.

*Grazie 1000 a tutti i miei amici Italiani* : Annarita, Lucia, Andrea, Ottavio, Roberto, Alessandro, Gianvito, Michele A. *e* Michele S. *Grazie 1000 a* Angela Bonifati.

Merci à mes parents, ma sœur Amira et mes deux frères Khaled et Hamza qui, malgré l'éloignement, ont cru en moi et m'ont soutenu. 1000 Merci et 1000 Bisous.

Enfin, Merci à Anissa pour son amour, son soutien et le deuxième souffle qu'elle a toujours su apporter avec tendresse et sérénité, dans les moments les plus difficiles surtout. Merci à sa famille également.

# Table des matières

**1 Introduction** **1**
1.1 Contexte . . . . . 1
1.2 Problématique . . . . . 4
1.3 Contributions . . . . . 5
1.3.1 Modèle d'arbre pour données multidimensionnelles . . . . 6
1.3.2 Opérateur XOLAP de forage vers le haut . . . . . 7
1.4 Organisation du mémoire . . . . . 8

**2 Définitions préalables** **9**
2.1 Définitions . . . . . 9
2.1.1 Entrepôt de données . . . . . 9
2.1.2 Hiérarchie complexe . . . . . 12
2.2 Exemple récurrent . . . . . 15

**3 État de l'art** **17**
3.1 Entreposage de données XML . . . . . 18
3.1.1 Modélisation en étoile des entrepôts de données XML . . . 18
3.1.2 Construction d'entrepôts de données XML à partir de sources variées . . . . . 20
3.1.3 Modèle unifié d'entrepôt de données XML . . . . . 22
3.1.4 Discussion . . . . . 23
3.2 De l'OLAP à l'XOLAP . . . . . 26
3.2.1 Opérateurs OLAP classiques . . . . . 27
3.2.2 Algèbres OLAP . . . . . 30
3.2.3 Approches XOLAP . . . . . 39
3.3 Problèmes d'additivité dans les modèles multidimensionnels . . . 50

3.3.1 Définition de l'additivité . . . . . 50
3.3.2 Contraintes d'additivité . . . . . 51
3.3.3 Modélisation des hiérarchies de dimensions complexes . . . 55
3.3.4 Normalisation des modèles multidimensionnels . . . . . 57
3.3.5 Transformation de données multidimensionnelles . . . . . 61
3.3.6 Solutions en temps réel . . . . . 63
3.3.7 Discussion . . . . . 64
3.4 Modèles d'arbres XML . . . . . 71
3.4.1 Définitions et modèle d'arbre de base . . . . . 71
3.4.2 Modèles d'arbres étendus . . . . . 75
3.4.3 Discussion . . . . . 83
3.4.4 Minimisation des modèles d'arbres XML . . . . . 87
3.4.5 Approches holistiques . . . . . 93
3.4.6 Autres utilisations . . . . . 104
3.4.7 Synthèse générale . . . . . 107

**4 Traitement des problèmes d'additivité dans les hiérarchies complexes 111**
4.1 Problématique et motivation . . . . . 112
4.2 Stratégies utilisées pour garantir l'additivité . . . . . 113
4.2.1 Résolution des problèmes d'additivité dans les hiéra-rchies non-strictes . . . . . 114
4.2.2 Résolution des problèmes d'additivité dans les hiéra-rchies non-couvrantes . . . . . 115
4.2.3 Résolution des problèmes d'additivité dans les hiéra-rchies complexes . . . . . 116
4.3 Arbres de données multidimensionnels . . . . . 116
4.3.1 Métamodèle . . . . . 118
4.3.2 Exemple d'instanciation du métamodèle . . . . . 119
4.4 Modèle d'arbre pour données multidimensionnelles . . . . . 120
4.5 Algorithmes . . . . . 124
4.5.1 Algorithme de regroupement et d'agrégation . . . . . 124
4.5.2 Forage vers le haut . . . . . 127
4.6 Discussion . . . . . 133

**5 Implémentation et validation expérimentale 139**
5.1 Implémentation . . . . . 140
5.1.1 Implémentation de *MakeCovering* . . . . . 140
5.1.2 Implémentation de *MakeStrict* . . . . . 142
5.2 Banc d'essais . . . . . 143
5.2.1 XWeB . . . . . 143
5.2.2 Extension du schéma de XWeB par des hiérarchies complexes 144
5.2.3 Modèle physique XML . . . . . 153
5.2.4 Extension de la charge de requêtes de XWeB . . . . . 156
5.3 Expériences . . . . . 156
5.3.1 Conditions expérimentales . . . . . 156
5.3.2 Plan d'expérience . . . . . 159
5.3.3 Résultats expérimentaux . . . . . 161

**6 Conclusion et perspectives 173**
6.1 Bilan général . . . . . 173
6.2 Perspectives de recherche . . . . . 175

# Table des figures


1.1 Modélisation multidimensionnelle des ventes d'une entreprise . . . 2
1.2 Modélisation de données multidimensionnelles en arbres de données 7
1.3 Exemple de modèle d'arbre . . . . . 7

2.1 Exemple d'entrepôt modélisé en étoile . . . . . 10
2.2 Exemple d'entrepôt modélisé en flocon de neige . . . . . 10
2.3 Exemple d'entrepôt modélisé en constellation . . . . . 11
2.4 Exemple de hiérarchie non-onto . . . . . 14
2.5 Exemple d'entrepôt de ventes . . . . . 15

3.1 Modélisation des faits ventes à l'aide de *XML-star schema* . . . . 19
3.2 Structures des documents (a) XCubeSchema.xml, (b) XCubeDimension.xml et (c) XCubeFact.xml . . . . . 20
3.3 Exemple de requête XQuery de création de dimension *temps* [RRT05] 21
3.4 Architecture de X-Warehousing [BBCA06] . . . . . 22
3.5 Architecture globale de l'entrepôt de données XML de Mahboubi *et al.* [MRL+09] . . . . . 25
3.6 Structure du document XCubeQuery.getFacts [HBH03] . . . . . 26
3.7 Exemple de requête décisionnelle XQuery [MRL+09] . . . . . 26
3.8 Exemple de cube de données de ventes . . . . . 27
3.9 Exemple de rotation du cube selon la dimension *livre* . . . . . 28
3.10 Exemple de découpage de cubes en tranches selon le *livre* "Verte" 29
3.11 Exemple de storage de données XML sous format relationnel [Kit10] 41
3.12 Exemple de forage vers le haut topologique (*topological rollup*) [Kit10] 42
3.13 Exemple de cube de données linguistiques [Egh10] . . . . . 43

3.14 Architecture d'intégration de données XML et relationnelles de Jensen *et al.* [JMP01] . . . . . 44
3.15 Modèle multidimensionnel d'entrepôt XML-OLAP [PHS05] . . . . 46
3.16 Notations utilisées par Malinowski et Zimányi [MZ06] . . . . . 56
3.17 Solution au niveau logique pour les hiérarchies non-onto [MZ06] . 57
3.18 Contraintes d'intégrité (a) et les dimensions minimales de Hurtado *et al.* (b) [HGM05] . . . . . 58
3.19 Exemple de relation fait-dimension non-couvrante (a) et solution de Mazón *et al.* (b) [MLT08] . . . . . 60
3.20 Exemple de hiérarchie non-stricte (a) et solution de Pedersen *et al.* (b) [PJD99] . . . . . 62
3.21 Trois cas de hiérarchies non-couvrantes de Horner et Song [HS05] 64
3.22 Exemples de (a) l'outil OLAP TARGIT [PRP02a, PPP04] et de (b) l'outil OLAP visuel de Mansmann et Scholl [MS06, MS07] . . 67
3.23 Arbre de données (a), modèle d'arbre (b) et arbre résultat (c) . . 75
3.24 Modèle d'arbre généralisé (a) et arbres résultats (b) . . . . . 76
3.25 Modèle d'arbre annoté (a) et arbre résultat (b) . . . . . 77
3.26 Modèle d'arbre annoté ordonné (a) et arbre résultat (b) . . . . . 78
3.27 Modèle d'arbre global . . . . . 79
3.28 Modèle de graphe . . . . . 80
3.29 Nœuds-opérateurs logiques . . . . . 81
3.30 Modèle d'arbre avec formule étendue . . . . . 82
3.31 Modèle d'arbre de Lu *et al.* . . . . . 83
3.32 Exemple de minimisation de modèle d'arbre à l'aide de l'algorithme CIM d'Amer-Yahia *et al.* [AYCLS01] . . . . . 90
3.33 Exemple de minimisation de modèle d'arbre à l'aide de l'algorithme *ChaseMinimizeFBST* . . . . . 93
3.34 Exemples d'étiquetage par région (en rouge) et Dewey (en vert) . 95
3.35 Exemple d'étiquetage Dewey dynamique [XLWB09] . . . . . 97
3.36 Modèle d'arbre (a), listes d'étiquettes par région (b) et résultat (c) 99
3.37 Comparaison de *TwigStack* et *TwigStackList* [Lu10] . . . . . 100
3.38 Comparaison de *TJFast* et *TJFast+L* [Lu10] . . . . . 102
3.39 Premier exemple de modèle d'arbre (à gauche) et d'arbre de données (à droite) utilisés dans *TreeMatch* [LLBW11] . . . . . 102

3.40 Deuxième exemple de modèle d'arbre (à gauche) et d'arbre de données (à droite) utilisés dans *TreeMatch* [LLBW11] . . . . . 103
3.41 Optimalité de *TreeMatch* [Lu10] . . . . . 104
3.42 Collection de modèles d'arbre $C$ (a) et modèle d'arbre fréquent $r$ (b) 106

4.1 Exemple de hiérarchie complexe *livre-catégorie* . . . . . 114
4.2 Métamodèle d'arbres de données multidimensionnels . . . . . 118
4.3 Exemple d'arbre de données multidimensionnel . . . . . 119
4.4 Modèle d'arbre pour données multidimensionnelles . . . . . 122
4.5 Processus de forage vers le haut . . . . . 128
4.6 Formule associée à la requête $Q_1$ . . . . . 128
4.7 Formule associée à la requête $Q_2$ . . . . . 129
4.8 Requête $Q_1$ : étape 1 . . . . . 129
4.9 Requête $Q_1$ : étape 2 . . . . . 130
4.10 Requête $Q_1$ : étape 3 . . . . . 131
4.11 Requête $Q_1$ : étape 4 . . . . . 132
4.12 Résultat de la requête $Q_1$ . . . . . 133
4.13 Requête $Q_2$ : étape 1 . . . . . 134
4.14 Requête $Q_2$ : étape 2 . . . . . 134
4.15 Requête $Q_2$ : étape 3 . . . . . 135
4.16 Résultat de la requête $Q_2$ . . . . . 135

5.1 Principe d'utilisation de XWeB [Mah08] . . . . . 144
5.2 Schéma conceptuel de l'entrepôt XWeB [Mah08] . . . . . 145
5.3 Requête Q1 exprimée en XQuery [Mah08] . . . . . 145
5.4 Schéma de XWeB exprimé en arbre de données multidimensionnel 147
5.5 Exemple de hiérarchie simple de XWeB . . . . . 150
5.6 Exemple de génération de hiérarchie non-couvrante . . . . . 151
5.7 Exemple de génération de hiérarchie non-stricte . . . . . 152
5.8 Exemple de génération de hiérarchie complexe . . . . . 154
5.9 Exemple de fait *sale* modélisé en XML . . . . . 154
5.10 Modélisation de (a) hiérarchies non-strictes, (b) non-couvrantes et (c) complexes en XML . . . . . 155
5.11 Temps d'exécution des requêtes en fonction du nombre de faits et de dimensions, avec `QBS` . . . . . 161

5.12 Comparaison de QBS et Pedersen sur des hiérarchies simples . . . 162
5.13 Temps de traitement de l'additivité et du *matching* des groupes dans QBS . . . 163
5.14 Temps de traitement de l'additivité (surcharge) et du *matching* des groupes dans Pedersen . . . 164
5.15 Comparaison de QBS et Pedersen sur des hiérarchies non-couvrantes de type 5 % . . . 165
5.16 Comparaison de QBS et Pedersen sur des hiérarchies non-couvrantes de type 50 % . . . 166
5.17 Comparaison de QBS et Pedersen sur des hiérarchies non-strictes de type 5 % . . . 167
5.18 Comparaison de QBS et Pedersen sur des hiérarchies non-strictes de type 50 % . . . 168
5.19 Comparaison de QBS et Pedersen sur des hiérarchies complexes de type 5 % . . . 168
5.20 Comparaison de QBS et Pedersen sur des hiérarchies complexes de type 50 % . . . 169
5.21 Évaluation des trois types de hiérarchies de type 5 % dans les requêtes avec QBS . . . 170
5.22 Évaluation des trois types de hiérarchies de type 50 % dans les requêtes avec QBS . . . 170

# Liste des tableaux


3.1 Comparaison des entrepôts de données XML . . . . . 23
3.2 Comparaison des algèbes OLAP . . . . . 37
3.3 Comparaison des approches XOLAP . . . . . 48
3.4 Hiérarchies non-strictes et problèmes d'additivité . . . . . 53
3.5 Hiérarchies non-couvrantes et problèmes d'additivité . . . . . 54
3.6 Classification des associations entre niveaux hiérarchiques d'une dimension [MLT08] . . . . . 55
3.7 Classification des associations fait-dimension [MLT08] . . . . . 55
3.8 Comparaison des solutions pour les problèmes d'additivité . . . . 65
3.9 Caractéristiques des différentes modèles d'arbre : synthèse . . . . 87

4.1 Récapitulatif de notre solution et les solutions pour les problèmes d'additivité similaires . . . . . 137

5.1 Spécification de la charge de XWeB [Mah08] . . . . . 146
5.2 Niveaux hiérarchiques de la dimension *part* . . . . . 147
5.3 Dimensions et niveaux hiérarchiques accédés par les requêtes utilisées dans nos expériences . . . . . 156
5.4 Tailles des données utilisées dans les expériences (en Ko) . . . . . 157

# Chapitre 1

# Introduction

## 1.1 Contexte

Les systèmes d'information ont connu, depuis presque une trentaine d'années maintenant, une longue suite d'innovations liée au développement des différentes infrastructures (client-serveur, internet/intranet, etc.) et des outils de représentation et de gestion de données (relationnel, objets, etc.). Parallèlement à ce développement, les entreprises à travers le monde n'ont pas cessé de se développer elles-aussi. Certaines entreprises ont des filiales dans tous les pays du monde ou presque. Ces entreprises ont besoin de nouveaux systèmes d'information qui leur permettent de gérer leurs grandes masses de données d'une façon efficace, de mettre à jour leurs bases de données en temps réel si nécessaire, et surtout de les historiser pour pouvoir les utiliser dans un processus d'aide à la décision. Ces systèmes d'information sont plus connus sous le nom de *systèmes d'information décisionnels*.

Les données d'entreprise sont alors qualifiées de *décisionnelles* à partir du moment où certains moyens (matériels et/ou immatériels) sont mis en œuvre pour les collecter, les consolider, les modéliser, les restituer et les analyser dans un but d'aide à la décision. Les outils de collecte de données sont plus connus sous le nom d'outils ETL (*Extract*, *Transform*, *Load* en Anglais ou Extraction, Transformation, Chargement en Français). Par exemple, supposons que nous voulions analyser les ventes d'une entreprise. Il nous faut dans ce cas collecter toute information qui puisse nous aider dans le processus d'analyse, comme les données concernant les clients (noms, adresses, etc.), les fournisseurs (noms, adresses, etc.) ou les produits

(noms de produits, dates de commandes, etc.).

Afin de les préparer au stockage et à l'analyse, les données décisionnelles sont modélisées, au niveau conceptuel, d'une façon dite multidimensionnelle. La *modélisation multidimensionnelle* consiste à représenter les données décisionnelles selon leurs niveaux d'agrégation. Un sujet d'analyse (*fait*) est représenté en fonction de plusieurs axes d'analyse (*dimensions*). Plus précisément, un fait est le croisement des $n$ dimensions qui interviennent dans le processus d'analyse. Les dimensions peuvent être hiérarchisées. À l'instar de cette représentation, les données décisionnelles peuvent être appelées *données multidimensionnelles*.

Soit l'exemple de la Figure 1.1, qui représente un modèle conceptuel multidimensionnel conçu en UML [1].

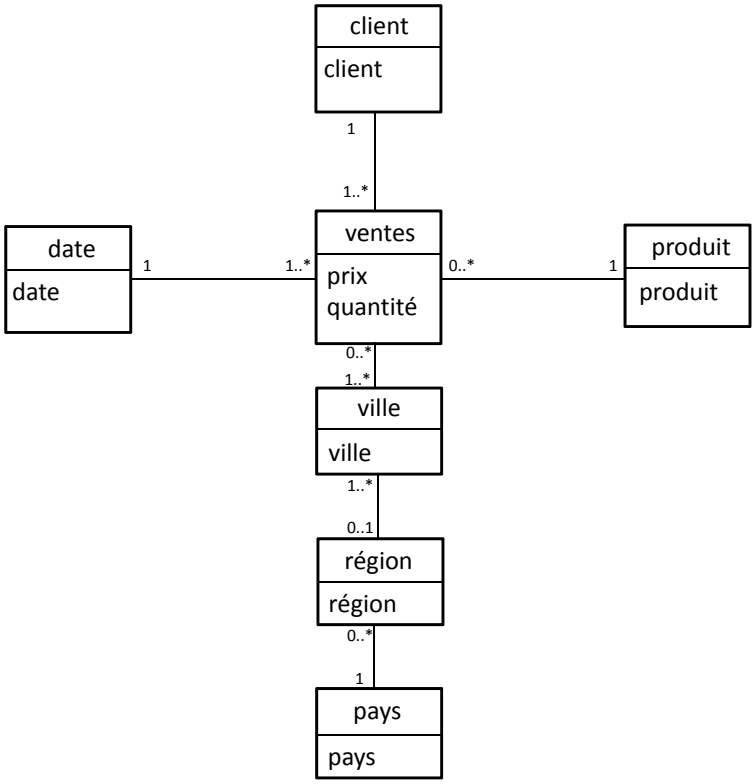



FIGURE 1.1 – Modélisation multidimensionnelle des ventes d'une entreprise

Ce modèle représente, d'une façon multidimensionnelle, les ventes d'une en-

1. `http://www.uml.org`

treprise. Nous y distinguons quatre dimensions :

- la dimension *géographique* organisée en hiérarchie : *ville-région-pays* (du plus détaillé au plus général),
- la dimension *client*,
- la dimension *produit*,
- la dimension *date*.

Les faits sont ici les *ventes*. Les attributs des *ventes*, *prix* et *quantité* d'un produit vendu à un client et à une date donnés, sont plus communément appelés *mesures*.

Les données multidimensionnelles sont habituellement stockées dans des *entrepôts de données* (*data warehouses* en Anglais) [KR02, Inm05]. Une fois entreposées, les données multidimensionnelles sont prêtes à l'analyse. Par définition, un entrepôt de données est en effet une collection de données orientées sujet, intégrées, non volatiles, historisées et organisées pour supporter un processus d'aide à la décision [Inm05]. Au niveau logique, les entrepôts de données sont classiquement modélisés dans des schémas en étoile où une table centrale, la *table de faits*, est reliée à l'ensemble des *tables de dimensions*.

Les outils d'analyse de données entreposées les plus populaires sont ceux qui sont conçus sur la base des technologies OLAP (*On-Line Analytical Processing*) [CCS93]. Par définition, OLAP est un ensemble d'opérateurs de manipulation multidimensionnelle sur lesquels sont construits ces outils d'analyse. Revenons à notre exemple d'analyse des ventes d'une entreprise (Figure 1.1). Comme exemple d'analyse, nous pouvons calculer la somme des ventes pour un client donné durant les cinq dernières années. Classiquement, au niveau physique, les tables de faits et de dimensions peuvent être implémentées dans les systèmes de gestion de bases de données relationnelles [PRP02a]. Dans ce cas, les requêtes OLAP sont alors définies dans des extensions du langage SQL [PRP02a].

Toutefois, XML (*eXtensible Markup Language*[2]) est devenu ces dernières années un standard pour la modélisation et l'échange de données. En effet, XML permet de modéliser des données qui ne sont pas facilement représentées dans les systèmes relationnels [BCC+05]. Depuis quelques années, une tendance vers l'entreposage de données XML se dégage. L'entreposage XML est défini comme la représentation des données décisionnelles dans des documents XML en respectant les principes

2. `http://www.w3.org/XML`

de la modélisation multidimensionnelle (les relations entre faits et dimensions).

Afin d'analyser les données multidimensionnelles dans les entrepôts de données XML avec la technologie OLAP, nous avons besoin d'outils d'analyse inspirés de l'OLAP classique et qui prennent en compte la structure flexible des données XML dans le processus d'analyse. Dans la littérature, cette approche est connue sous le nom de XML-OLAP [PHS05] ou XOLAP.

## 1.2 Problématique

Les travaux sur XOLAP que nous trouvons dans la littérature ne sont pas de véritables algèbres XOLAP, mais forment plutôt des tentatives intéressantes de définition d'un environnement XOLAP et des notions associées [JMP01, PRP02a, PRP02b, PP03, PPP04, PHS05, WJLS07]. La majorité de ces travaux consistent à stocker les données XML dans des systèmes de gestion de bases de données relationnels afin de pouvoir les interroger avec des extensions des langages de requêtes pour données relationnelles [PRP02a]. Même si certains d'entre eux stockent les données XML dans des systèmes de gestion de bases de données (SGBD) natifs XML et les interrogent avec le langage XQuery [3], leur contribution se limite généralement à un seul opérateur XOLAP, comme l'opérateur *cube* proposé par Wiwatwattana *et al.* [WJLS07].

En plus de leur rareté, les opérateurs XOLAP présentés dans la littérature ne prennent pas en compte les structures complexes que peuvent prendre les données multidimensionnelles. Les relations entre les niveaux hiérarchiques des dimensions peuvent en effet être complexes. Par exemple, soit la dimension *géographique* de la Figure 1.1. Les associations entre les différents niveaux hiérarchiques de cette dimension (*ville-région-pays*) reflètent le cas réel où certains pays du monde ne contiennent pas de régions (comme Monaco [4], qui est à la fois une ville et un état) ; elles reflètent aussi d'autres cas réels comme les pays qui contiennent des communes mais pas de région (comme Saint-Marin [5]). La complexité de ces hiérarchies (modélisant des cas réels) les rendent d'autant plus sensibles aux erreurs d'analyse. Par exemple, supposons que nous voulons analyser les ventes par ville, par région

3. `http://www.w3.org/TR/xquery/`
4. `http://fr.wikipedia.org/wiki/Monaco`
5. `http://fr.wikipedia.org/wiki/Saint-Marin`

et par pays. La prise en compte de ces structures complexes dans le processus d'analyse est nécessaire ; sinon, les résultats seront erronés.

Les hiérarchies complexes sont un sujet très sensible dans les entrepôts de données et l'OLAP ; et le fait de ne pas les prendre en compte en concevant l'entrepôt de données (et/ou les opérateurs OLAP) peut être à l'origine d'erreurs d'analyse et, par la suite, la cause principale de la faillite de tout le projet décisionnel. Les problèmes dus à la présence des hiérarchies complexes sont connus dans la littérature sous le nom de *problèmes d'additivité* [MLT09]. Le terme additivité, que nous utilisons dans ce mémoire, est une traduction réductrice du terme anglais *summarizability*. Malheureusement, les travaux qui présentent un environnement XOLAP ne prennent pas en compte les problèmes d'additivité ni dans la conception de l'entrepôt de données XML ni dans les opérateurs XOLAP présentés.

Les problèmes d'additivité sont donc essentiellement générés suite au mauvais choix d'instances de dimensions d'une hiérarchie complexe. En outre, ce mauvais choix peut être fait d'une façon involontaire au moment de l'analyse. C'est alors le rôle du concepteur du projet décisionnel de faire attention au moment de la conception de son entrepôt et/ou de son outil d'analyse, en modélisant ce type de hiérarchies, pour éviter ces erreurs.

## 1.3 Contributions

Les propositions XOLAP présentées dans la littérature n'ont pas profité de la flexibilité qu'offre XML pour définir des opérateurs XOLAP spécifiques [JMP01, PRP02a, PRP02b, PP03, PPP04, PHS05, WJLS07]. De plus, dans toutes ces approches, nous remarquons l'absence de traitement des hiérarchies complexes pour prendre en compte les problèmes d'additivité.

Dans cette thèse, nous nous sommes donc concentrés sur la définition d'un nouvel environnement XOLAP qui permet de modéliser et de traiter les données multidimensionnelles exprimées selon des hiérarchies complexes, et de prendre en compte les problèmes d'additivité. Nous modélisons les données multidimensionnelles dans des arbres de données et nous présentons :

1. un modèle d'arbre pour données multidimensionnelles qui permet de parcou-

rir les arbres de données multidimensionnelles et de repérer les hiérarchies complexes,

2. un opérateur XOLAP de forage vers le haut (*roll-up*) basé sur notre modèle d'arbre et qui prend en compte les problèmes d'additivité dans les hiérarchies complexes afin de reproduire des résultats d'agrégation correctes.

### 1.3.1 Modèle d'arbre pour données multidimensionnelles

Dans notre travail, nous choisissons XML pour modéliser les données multidimensionnelles. Un document XML est généralement vu comme un arbre ordonné, étiqueté et enraciné [JLST01]. Au niveau conceptuel, nous modélisons les données multidimensionnelles à l'aide d'arbres de données. En fait, l'arbre est l'une des structures de données les plus utilisés en informatique. Dans le domaine des bases de données semi-structurées, la structure d'un document XML correspond essentiellement à un arbre. Une des particularités des arbres de données est que ces derniers permettent de modéliser des collections d'objets similaires (par exemple, des livres) sous une seule racine. Dans un arbre de données, les données similaires peuvent être décrites de façon hétérogène, c'est-à-dire que chaque élément peut être décrit d'une façon qui peut être différente des autres éléments de même nature dans le même arbre.

Soit l'exemple d'arbre de données de la Figure 1.2(a), inspiré de la Figure 1.1 et modélisant les ventes des livres. Les deux livres de cet arbre de données ne sont pas décrits de la même façon. Le premier livre, "La vie sauve", est écrit par deux auteurs contrairement au deuxième livre, "Verte". De plus, on ne connait pas l'année de publication de "La vie sauve" ; par contre, on la connait pour "Verte". Le fichier XML correspondant à cet arbre de données est fourni dans la Figure 1.2(b). Il est clair, via cet exemple, que les arbres de données (comme XML) offrent une grande flexibilité de modélisation de données en général, et des données multidimensionnelles dans notre travail.

Pour manipuler ces arbres de données, nous nous inspirons des algèbres XML (comme l'algèbre XML TAX [JLST01]) et de plusieurs autres approches XML qui ont choisi les modèles d'arbre (*tree patterns* en Anglais) pour interroger les arbres de données [AYCLS01, GC07, HD12]. Par définition, un modèle d'arbre représente une requête sous forme d'arbre dont l'ensemble de nœuds et de relations

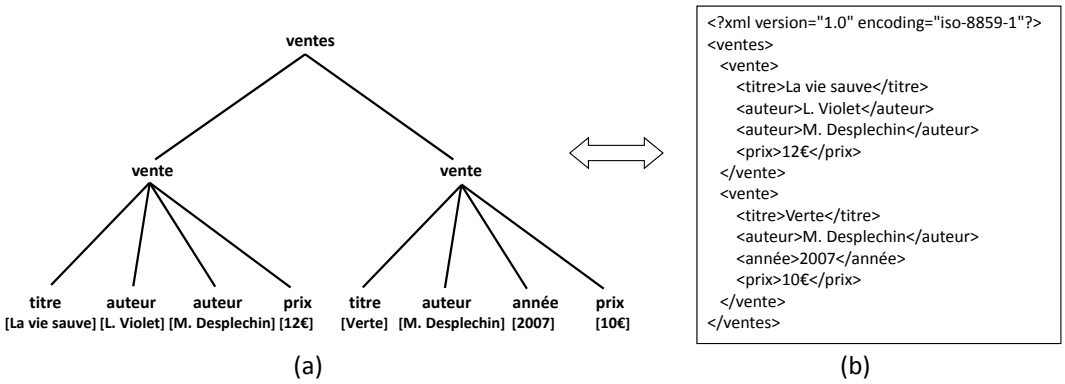



FIGURE 1.2 – Modélisation de données multidimensionnelles en arbres de données

entre elles (représentées par des arêtes) appartient à celui de l'arbre de données à interroger. Par exemple, soit le modèle d'arbre de la Figure 1.3(a). Ce modèle d'arbre a pour but de sélectionner les titres de tous les livres de la Figure 1.2(a). Les arbres résultats (*Witness Trees* en Anglais [JLST01]) sont illustrés dans la Figure 1.3(b). Dans cette thèse, nous proposons un modèle d'arbre plus évolué que celui de la Figure 1.3(a) pour modéliser des requêtes sur les arbres de données multidimensionnelles.

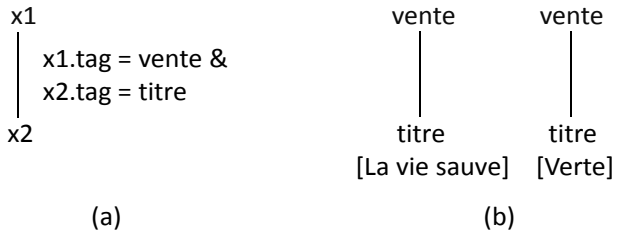



FIGURE 1.3 – Exemple de modèle d'arbre

### 1.3.2 Opérateur XOLAP de forage vers le haut

Comme nous l'avons indiqué dans la Section 1.3.1, les arbres de données peuvent supporter des structures hétérogènes. Ces structures hétérogènes s'illustrent dans toutes les formes possibles que peuvent prendre les relations entre certains nœuds de l'arbre. Des relations entre nœuds de différents niveaux reliés entre

eux sont plus communément appelées hiérarchies. Outre les hiérarchies simples (de type un à plusieurs), les arbres de données peuvent aussi supporter les hiérarchies complexes (de types autres qu'un à plusieurs).

À notre connaissance, l'additivité n'est pas traitée dans les approches XOLAP présentées dans la littérature. Certaines approches ont même préféré ignorer ce problème en proposant des opérateurs XOLAP pour les hiérarchies simples seulement [WJLS07]. Or, le problème d'additivité est un sujet très sensible dans l'OLAP et le négliger peut être la cause principale d'échec de tout projet décisionnel. L'opérateur XOLAP de forage vers le haut, que nous présentons dans cette thèse, est conçu sur la base du modèle d'arbre pour données multidimensionnelles (Section 1.3.1). La grande particularité de notre opérateur est qu'il prenne en compte les problèmes d'additivité. En premier lieu, cet opérateur parcourt l'arbre de données et détecte la complexité dans les hiérarchies. En cas de hiérarchie complexe, notre opérateur utilise les options que nous définissons au niveau du modèle d'arbre afin de calculer l'agrégation. Quel que soit importe le degré de complexité de la hiérarchie, notre opérateur retourne toujours des résultats d'analyse corrects.

## 1.4 Organisation du mémoire

Le reste de ce mémoire est organisé comme suit. Dans le Chapitre 2, nous formalisons les définitions sous-jacentes à nos travaux et nous donnons un exemple récurrent d'entrepôt de données que nous utilisons dans tout ce mémoire. Un état de l'art des travaux concernant l'entreposage de données XML, les approches XOLAP, les problèmes d'additivité dans les modèles multidimensionnels ainsi que les modèles d'arbre XML est présenté dans le Chapitre 3. Nous détaillons notre première contribution, un modèle d'arbre pour données multidimensionnelles, dans le Chapitre 4. Plus précisément, dans ce chapitre, nous présentons notre modèle de données pour représenter les données multidimensionnelles et le modèle d'arbre adapté à ces données. Ensuite, nous présentons la démarche que nous adoptons pour résoudre les problèmes d'additivité dans les hiérarchies complexes. Les résultats des expériences que nous avons menées pour valider nos propositions sont présentés dans le Chapitre 5. Enfin, nous concluons ce mémoire dans le Chapitre 6. Nous y dressons le bilan de nos contributions et nous présentons nos perspectives de recherche.

# Chapitre 2

# Définitions préalables

Dans ce chapitre, nous définissons formellement les entrepôts de données, les faits, les dimensions et les hiérarchies (spécialement les hiérarchies complexes) en premier lieu. En second lieu, nous donnons un exemple d'entrepôt de données que nous utilisons d'une façon récurrente dans le reste de ce mémoire.

## 2.1 Définitions

### 2.1.1 Entrepôt de données

La conception d'un entrepôt de données se base sur deux concepts fondamentaux : les faits et les dimensions. Les faits à analyser sont décrits selon plusieurs dimensions. Tout modèle multidimensionnel doit respecter ce principe. Classiquement, les entrepôts de données sont modélisés *en étoile*, *en flocon de neige* ou *en constellation* [KR02].

Dans un *modèle en étoile*, l'élément central représente les faits et les branches sont les dimensions. Soit le modèle d'entrepôt en étoile présenté dans la Figure 2.1 qui représente des ventes de livres. Les faits sont *ventes*. Les dimensions sont *temps*, *livre* et la dimension *géographique*.

Dans un *modèle en flocon de neige*, le principe est le même que pour le modèle en étoile. Mais en plus, une des dimensions, au moins, est représentée en hiérarchie.

Dans le modèle en flocon de neige de la Figure 2.2, les dimensions *temps* et *géographique* sont représentées en hiérarchies. De plus, la dimension *temps* est organisée en une hiérarchie alternative [MS07]. Une dimension est dite organisée en

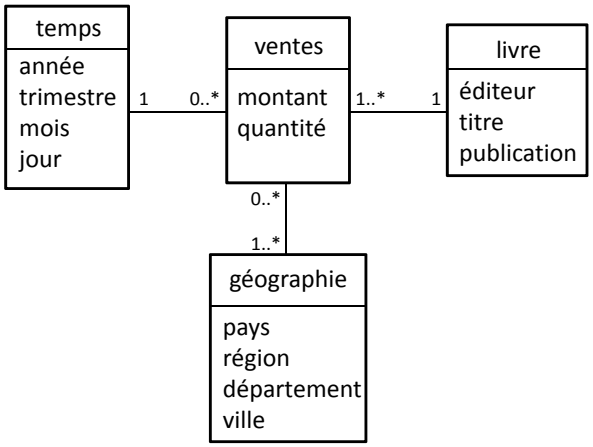



FIGURE 2.1 – Exemple d'entrepôt modélisé en étoile

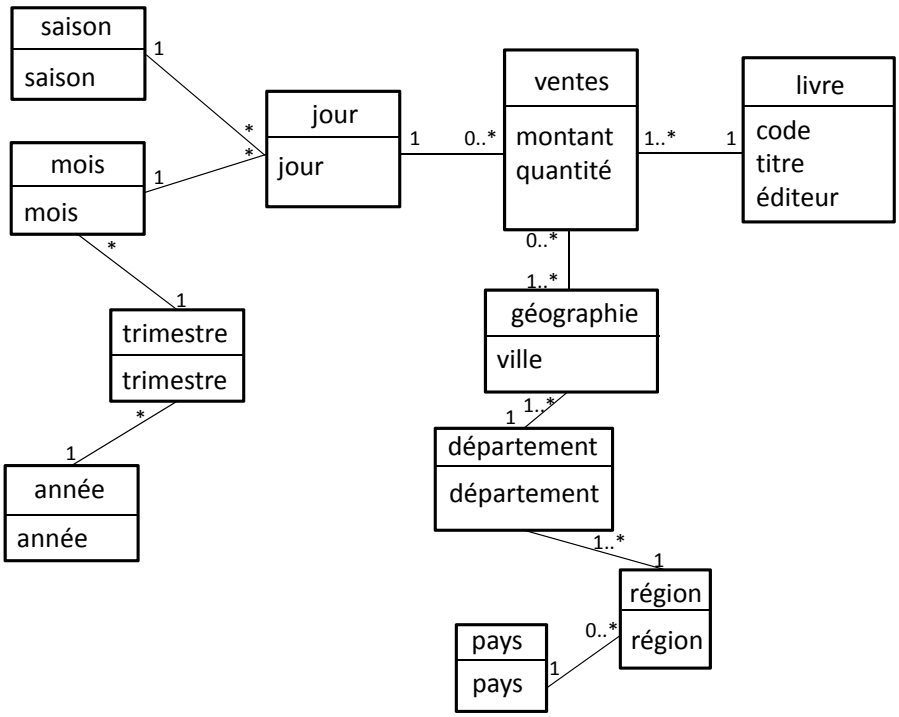



FIGURE 2.2 – Exemple d'entrepôt modélisé en flocon de neige

*hiérarchie alternative* lorsqu'une ou plusieurs sous-hiérarchies dépendent d'un ou de plusieurs niveaux hiérarchiques de cette dimension [MS07]. Dans la dimension

*temps* de la Figure 2.2, deux sous-hiérarchies dépendent de *jour* : *jour-mois-année* et *jour-saison*.

Le troisième modèle, le *modèle en constellation*, est basé aussi sur le modèle en étoile, mais rassemble plusieurs tables de faits qui partagent les mêmes dimensions. Dans le modèle en constellation de la Figure 2.3, nous distinguons deux faits reliés à la dimension *géographique* : *ventes* et *prescriptions*.

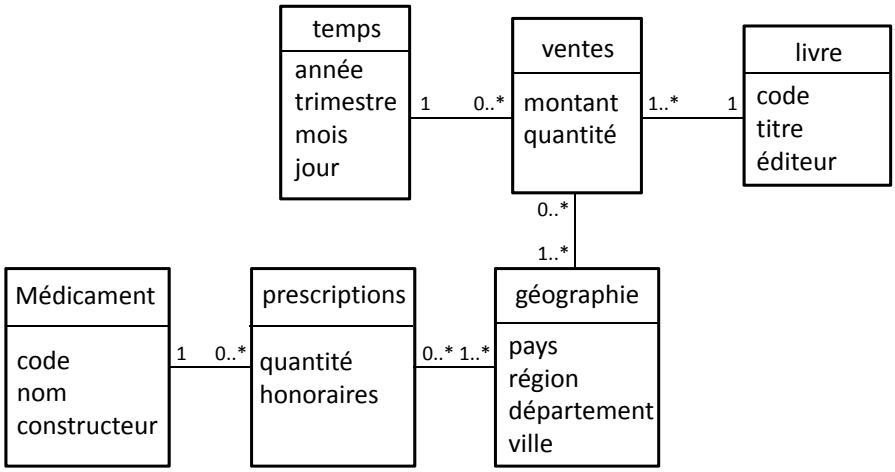



FIGURE 2.3 – Exemple d'entrepôt modélisé en constellation

Formellement, un entrepôt de données $E$ modélisé en flocon de neige est défini par un triplet $E = (\mathbb{F}, \mathbb{D}, *)$ où :

- $\mathbb{F} = \{F_i\}_{i=1,f}$ est l'ensemble des faits (avec $f = |\mathbb{F}|$) ;
- $\mathbb{D} = \{D_i\}_{i=1,d}$ est l'ensemble des dimensions (avec $d = |\mathbb{D}|$) ;
- $* : \mathbb{F} \rightarrow \mathbb{D}$ est une fonction qui associe chaque fait à l'ensemble des dimensions qui le décrivent [Tes09].

### Dimension et hiérarchie

Une dimension est définie comme un axe d'observation selon lequel un fait est observé. Une dimension peut comporter plusieurs hiérarchies impliquant différents niveaux de précision dans la description des faits. Ces hiérarchies permettent d'observer des mesures selon plusieurs niveaux de granularité et de construire des agrégats à partir des faits du niveau le plus fin [KR02].

$\forall i \in [1, d]$, une dimension $D_i \in \mathbb{D}$est définie par une hiérarchie constituée d'un ensemble de $n_i$ niveaux : $D_i = \{H_{ij}\}_{j=1,n_i}$. Par convention, nous désignons $H_{i1}$ comme le niveau de granularité le plus fin.

$\forall j \in [1, n_i]$, un niveau hiérarchique $H_{ij}$ est défini en intention par $H_{ij} = (ID_{ij}, \{A_{ijk}\}_{k=1,a_{ij}}, R_{ij})$, où :

- $ID_{ij}$ est l'attribut identifiant de $H_{ij}$ ;
- $\{A_{ijk}\}$ est un ensemble de $a_{ij}$ attributs dits membres de $H_{ij}$ ;
- $R_{ij}$ est un attribut référençant un niveau hiérarchique de granularité supérieure à celle de $H_{ij}$ (notion de *roll-up*).

Soit $dom()$ une fonction qui associe à tout attribut son domaine de définition. Les instances de $H_{ij}$ sont des n-uplets de la forme $(\sigma_{ij}, \{\alpha_{ijk}\}_{k=1,a_{ij}}, \rho_{ij})$, où :

- $\sigma_{ij} \in dom(ID_{ij})$ ;
- $\alpha_{ijk} \in dom(A_{ijk})\ \forall k \in [1, a_{ij}]$ ;
- $\rho_{ij} \in dom(ID_{ij'})$ avec $j' \in [1, n_i]$.

**Fait**

Un fait représente un sujet d'analyse et est caractérisé par une ou plusieurs mesures ou indicateurs ayant des propriétés d'additivité [KR02].

Formellement, un fait $F_i \in \mathbb{F}$ est défini en intention par $F_i = (\{\Delta_i\}_{i=1,d}, \{M_j\}_{j=1,m})$, où :

- $\{\Delta_i\}$ est un ensemble de $d$ attributs référençant des instances des niveaux hiérarchiques $H_{i1}$ de chaque dimension $D_i \in \mathcal{D}$ ;
- $\{M_j\}$ est un ensemble de $m$ attributs mesures (ou indicateurs) caractérisant les faits.

Les instances de $F$ sont des n-uplets de la forme $(\{\delta_i\}_{i=1,d}, \{\mu_j\}_{j=1,m})$, où :

- $\delta_i \in dom(ID_{i1})\ \forall i \in [1, d]$ ;
- $\mu_j \in dom(M_j)\ \forall j \in [1, m]$.

### 2.1.2 Hiérarchie complexe

Nous qualifions une hiérarchie de dimension $D_i$ de complexe si elle est à la fois non-stricte et non-couvrante.

### Hiérarchie non-stricte

Une hiérarchie est dite non-stricte [Tor03, ASS06, MZ06] ou à arcs multiples [Riz07] lorsque l'attribut $R_{ij}$ est multivalué. En d'autres termes, d'un point de vue conceptuel, une hiérarchie est non-stricte si l'association entre deux niveaux hiérarchiques est de type plusieurs-à-plusieurs et non de type un-à-plusieurs. Par exemple, dans une dimension décrivant des produits, un produit donné peut appartenir à plusieurs catégories et non une seule.

De même, une association plusieurs-à-plusieurs entre faits et instances de dimension peut exister [Tor03]. Par exemple, dans un entrepôt de données de ventes, un fait peut être associé à une combinaison d'offres promotionnelles plutôt qu'une seule. Formellement, ce sont ici les attributs $\Delta_i$ ($\forall i \in [1, d]$) qui peuvent être multivalués.

### Hiérarchie non-couvrante

Une hiérarchie est dite non-couvrante [Tor03, ASS06, MZ06], incomplète [PR00, DPJ03] ou effilochée (*ragged*) [Riz07] si l'attribut $R_{ij}$ permet de relier un niveau hiérarchique $H_{ij}$ à un niveau $H_{ij}{}'$ en "sautant" un ou plusieurs niveaux intermédiaires, c'est-à-dire : $\rho_{ij} = \sigma_{ij}{}'$ et $\exists H_{ij}{}'' \in D_i \ / \ \rho_{ij}{}'' = \sigma_{ij}{}'$. Ce cas arrive, par exemple, si dans une dimension décrivant des magasins, la hiérarchie *magasin-ville-région-pays* permet qu'un magasin soit localisé dans une région donnée sans être associé à une ville (cas des magasins situés en zone rurale).

De même, les faits peuvent être décrits à des niveaux de granularité hétérogènes. Par exemple, toujours dans notre entrepôt de données de ventes, le volume de vente peut être connu au niveau des magasins dans une partie du monde (comme l'Europe), mais seulement à un niveau plus agrégé (comme le pays) dans d'autres zones géographiques. Cela signifie que, $\forall i \in [1, d]$, $\delta_i \in dom(ID_{ij})$ avec $j \in [1, n_i]$ (la contrainte $j = 1$ est levée).

Les hiérarchies non-onto [1] représentent un cas particulier des hiérarchies non-couvrantes [PJD01]. Une hiérarchie est dite non-onto lorsque la distance entre le plus haut et le plus bas niveaux de granularité de la hiérarchie varie d'un cas à un autre. Dans une hiérarchie non-onto, les niveaux manquants ne peuvent jamais

1. Plusieurs termes sont utilisés dans la littérature pour désigner les hiérarchies non-onto : hétérogènes [HM01], non-équilibrées [HLBS02] ou asymétriques [MZ06].

être des niveaux intermédiaires (contrairement aux hiérarchies non-couvrantes). Par exemple, soit la hiérarchie *agence-agence centrale-branche-banque* (du plus détaillé au plus général) de la Figure 2.4. Les branches sont organisées au niveau départemental et les agences centrales au niveau ville. Dans le cas d'une grande ville comme Lyon, une agence centrale ne suffit pas et plusieurs agences sont nécessaires pour répondre aux besoins des clients. Toutefois, aucune agence ne dépend d'aucune agence centrale.

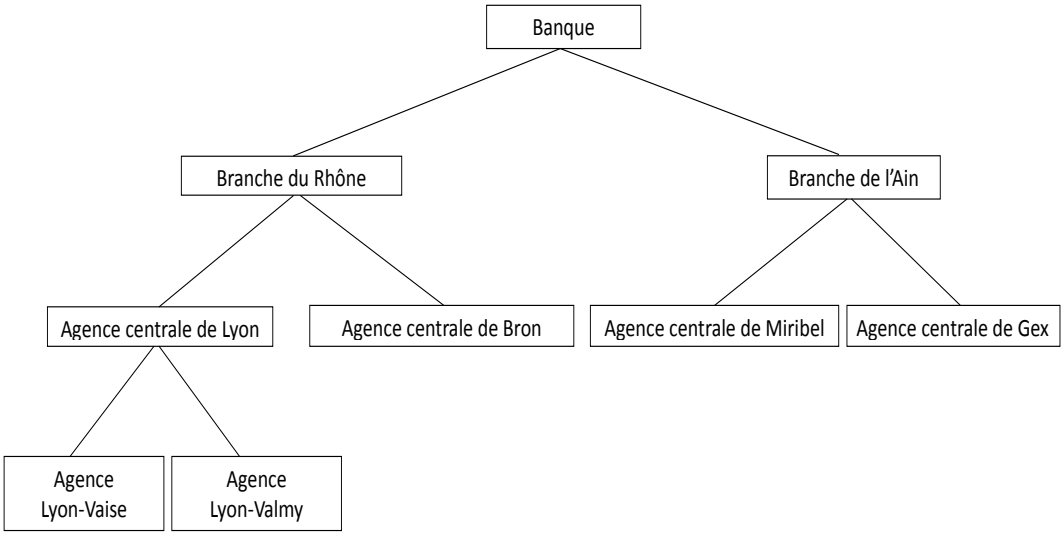



FIGURE 2.4 – Exemple de hiérarchie non-onto

### Discussion

La notion de hiérarchie effilochée (*ragged hierarchy*) a différentes acceptions dans la littérature. Par exemple, Beyer *et al.* la définissent comme une hiérarchie à la fois non-stricte et non-couvrante [BCC+05] alors que Rizzi la définit comme uniquement non-couvrante [Riz07]. C'est pourquoi nous lui préférons et définissons les nouveaux termes de hiérarchie complexe. Malinowski et Zimányi utilisent des termes voisins de manière interchangeable : hiérarchie généralisée [MZ06] et hiérarchie généralisée complexe [MZ08]. Toutefois, ces hiérarchies incluent les hiérarchies non-couvrantes, mais pas les hiérarchies non-strictes.

## 2.2 Exemple récurrent

La Figure 2.5 représente le modèle conceptuel UML d'un entrepôt de ventes d'une librairie. La table *ventes* est la table de faits. La mesure est le *prix* du livre. La librairie en question a des filiales dans plusieurs pays du monde. Les dimensions sont : *lieu* de vente (hiérarchie *ville-région-pays*), temps (hiérarchie *jour-mois-année*), *livre* et *catégorie*. Notons que nous modélisons la dimension *catégorie* d'une façon récursive. Les relations entre les dimensions *livre* et *catégorie* signifient qu'un livre peut appartenir à plusieurs catégories, et que chaque catégorie peut contenir plusieurs livres. D'un autre côté, chaque catégorie peut appartenir à plusieurs catégories.

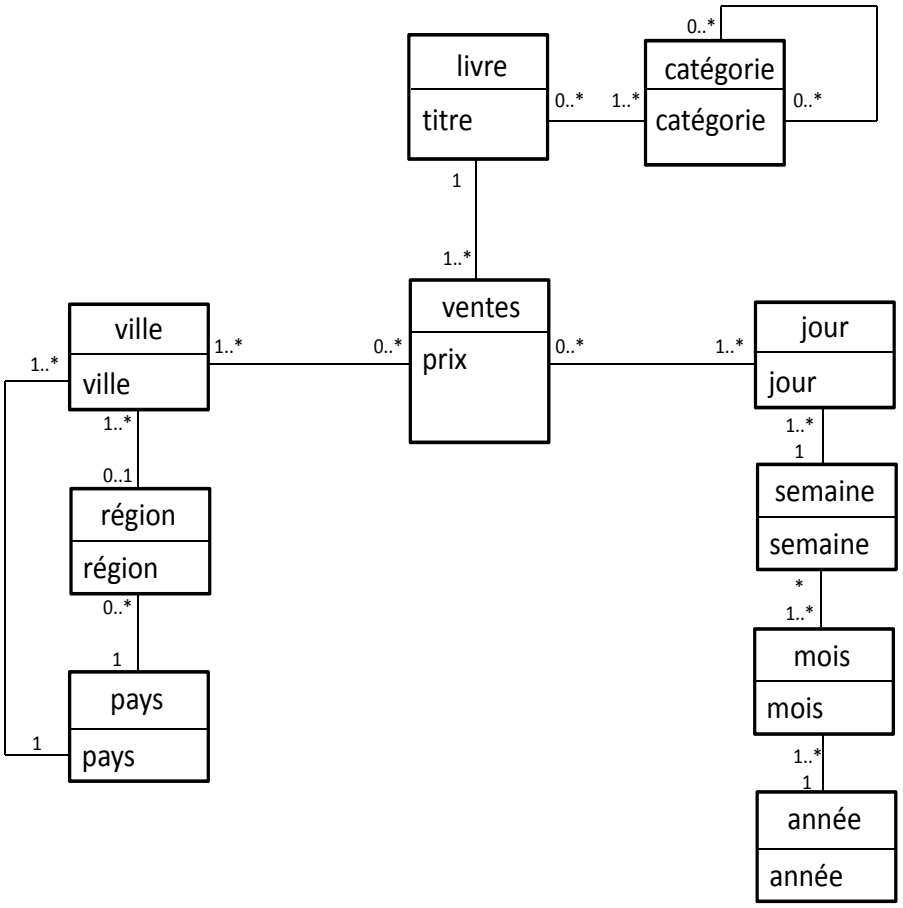



FIGURE 2.5 – Exemple d'entrepôt de ventes

# Chapitre 3

# État de l'art

Notre travail traite des analyses XOLAP sur des données multidimensionnelles exprimées selon des hiérarchies complexes. Plus précisément, notre proposition se résume à un opérateur de forage vers le haut (*roll-up*) basé sur un modèle d'arbre (*tree pattern*) pour données multidimensionnelles. Cet opérateur permet de résoudre les problèmes d'additivité dus à la modélisation des données XML multidimensionnelles en hiérarchies complexes. Afin de mieux aborder les problèmes liés à notre proposition, nous présentons dans la Section 3.1 de ce chapitre une étude bibliographique des travaux qui traitent de l'entreposage de données XML. Dans la Section 3.2, nous étudions les approches OLAP et XOLAP présentées dans la littérature. Plus précisément, nous commençons par étudier les différentes algèbres OLAP. Ensuite, nous présentons les approches XOLAP pour illustrer comment les concepts OLAP classiques ont été mis en œuvre pour l'analyse multidimensionnelle des données XML.

Dans la Section 3.3, nous étudions le traitement de l'additivité dans les modèles multidimensionnels. En premier lieu, nous donnons la définition des problèmes d'additivité. Nous verrons par la suite comment les hiérarchies complexes peuvent être à l'origine des problèmes d'additivité dans les analyses OLAP. Nous finissons par une étude des différentes solutions proposées dans la littérature pour résoudre les problèmes d'additivité. Finalement, la Section 3.4 est consacrée à une étude approfondie des modèles d'arbre présentée dans la littérature. Dans cette section, nous mettons l'accent sur l'efficacité de ces modèles pour traduire des requêtes complexes sur les arbres de données XML.

## 3.1 Entreposage de données XML

Mahboubi *et al.* classifient les travaux sur l'entreposage des données XML en deux familles : (1) les travaux qui ont proposé un modèle conceptuel et les méthodes nécessaires pour la construction d'*entrepôts de données XML* et (2) les travaux qui traitent de l'*entreposage de documents XML* et considèrent un entrepôt XML comme une collection de documents XML [MHD09].

Dans cette section, nous ne nous intéressons qu'aux travaux qui ont proposé des entrepôts de données XML. Les travaux de la deuxième famille [Xyl01, BB03, NRDR05, RCD05, ZWLZ05] ne sont pas pertinents pour notre travail puisque nous modélisons les données XML de la même façon que dans la première famille. Dans ce qui suit, nous classifions les entrepôts de données XML dans trois familles : les entrepôts XML modélisés en étoile, les entrepôts XML conçus à partir de sources variées et l'entrepôt XML unifié de Mahboubi *et al.* [MRL+09].

### 3.1.1 Modélisation en étoile des entrepôts de données XML

Pokorný propose l'approche *XML-star schema* qui extrait, en premier lieu, les données sources provenant de bases de données OLTP (*On-Line Transaction Processing*) [Pok02]. Les données XML sont validées par des DTD (*Document Type Definition*[1]) extraites des données sources. Ensuite, les entrepôts de données XML sont modélisés par des schémas en étoile. Afin de relier d'une façon logique ces documents XML, Pokorný introduit et emploie les contraintes d'intégrité référentielles.

Le modèle de Pokorný supporte aussi les hiérarchies de dimensions. Dans la Figure 3.1, nous modélisons les faits *ventes* et les dimensions *géographique* et *livre* de la Figure 2.5 à l'aide de la méthode de Pokorný. Cet exemple illustre les faits et les dimensions validés par des DTD, et comment ces mêmes faits et dimensions sont reliés ensemble à l'aide des contraintes d'intégrité référentielles (représentées par des flèches).

Dans un deuxième travail, Hümmer *et al.* proposent l'entrepôt de données XML *XCube* pour la transmission et le partage de données multidimensionnelles [HBH03]. Dans XCube, trois types de documents sont définis :

1. `http://www.w3.org/TR/html4/sgml/dtd.html`

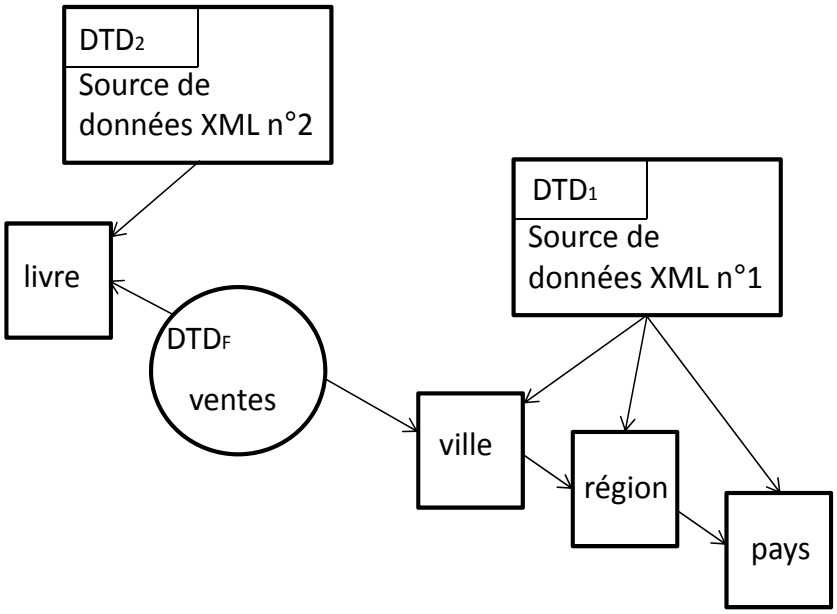



FIGURE 3.1 – Modélisation des faits ventes à l'aide de *XML-star schema*

1. *XCubeSchema.xml* décrit la structure multidimensionnelle d'un cube de données (les faits et les dimensions), comme le montre la Figure 3.2(a). En plus des faits et des dimensions (spécifiés sous le nœud *CubeSchema*), les niveaux hiérarchiques d'une dimension sont spécifiés à l'aide du nœud *classSchema*. Notons que les lignes discontinues représentent les relations père-enfant. Les relations père-enfant marquées avec le symbole $*$ dans la Figure 3.2 signifient que le nœud père peut avoir de un à plusieurs nœuds enfants.
2. *XCubeDimension.xml* modélise toutes les dimensions, comme le montre la Figure 3.2(b). Chaque dimension est décrite par son niveau hiérarchique (*Level*), ses attributs (*attribute*) et la référence vers le niveau hiérarchique supérieur (*rollUp*).
3. *XCubeFact.xml* spécifie les faits, tout en respectant la définition de base d'un fait. Un fait est décrit par les identificateurs des dimensions et les mesures (Figure 3.2(c)). Dans un fait, un nœud *cube* représente un cube de données et est constitué d'un ensemble de nœuds *cell* qui décrivent les faits du cube.

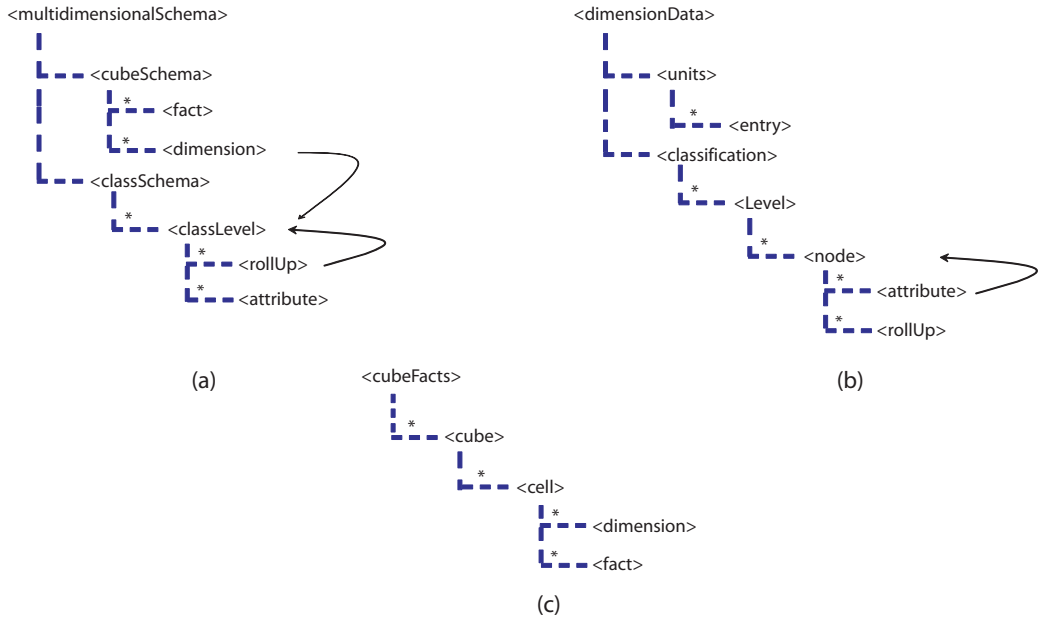



FIGURE 3.2 – Structures des documents (a) XCubeSchema.xml, (b) XCubeDimension.xml et (c) XCubeFact.xml

### 3.1.2 Construction d'entrepôts de données XML à partir de sources variées

Rusu *et al.* se basent sur le langage XQuery pour construire un entrepôt de données XML [RRT05]. La méthode proposée couvre toutes les étapes du processus d'ETL : l'extraction, la transformation et le chargement des données XML dans l'entrepôt. Toutes ces étapes sont assurées à l'aide de XQuery. Les auteurs définissent des règles de chargement pour faire face aux problèmes de conflits rencontrés lors du chargement des données dans l'entrepôt. Ces conflits sont de deux types : (1) au *niveau schéma* lorsque ce dernier ne correspond totalement pas aux données, et (2) au *niveau données* suite aux irrégularités structurelles des données XML. Les faits et les dimensions sont stockés dans des documents XML également construits par des requêtes XQuery. Un exemple de requête XQuery qui construit une dimension *temps* est présenté dans la Figure 3.3.

Golfarelli *et al.* proposent une approche semi-automatique pour la construc-

```
let $b:=0 document {
for $t in distinct-values(doc("libraryBooks.xml")//borrowing_date)
let $b:=$b+1
return <borrowtime>
<timekey>$b</timekey>
<borrowdate>$t</borrowdate>
<month>get-month-from-date($t)</month>
</borrowtime> }
```

FIGURE 3.3 – Exemple de requête XQuery de création de dimension *temps* [RRT05]

tion de magasins de données depuis des sources XML sur le Web [GRV01]. Un magasin de données est un extrait d'un entrepôt orienté métier qui permet de faire des analyses plus rapides sur un volume de données plus restreint. Avant de commencer la conception de l'entrepôt de données, les sources XML sont validées par des schémas XML [2]. Afin de concevoir l'entrepôt de données XML, la méthode de Golfarelli *et al.* se résume en trois étapes. En premier lieu, les faits (les sujets d'analyse) sont identifiés, d'une façon manuelle, depuis les schémas XML utilisés dans le processus de validation. Puis, pour chaque fait identifié, un arbre d'attributs est construit. Cet arbre représente les relations entre l'élément fait et le reste des éléments des données XML (comme les dimensions). Enfin, l'arbre de données qui vient d'être construit est réarrangé afin que les dimensions et les mesures soient définies au sein de ces arbres.

Inspirés par ce travail, Boussaid *et al.* définissent un entrepôt de données XML par un schéma XML et proposent par la suite la plateforme X-Warehousing pour l'entreposage de *données complexes* [BBCA06]. Les différentes étapes de l'approche X-Warehousing sont illustrées dans la Figure 3.4. En entrée de X-Warehousing, les besoins d'analyse utilisateurs et les documents XML contenant les données à analyser sont modélisés dans des schémas XML. Ensuite, ces schémas XML sont transformés en arbres d'attributs avec la méthode de Golfarelli *et al.* [GRV01]. Un arbre d'attributs final est construit suite à la fusion de ces deux arbres d'attributs. La fusion est assurée à l'aide des techniques d'élagage (*pruning*)

2. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`

et/ou de greffe (*grafting*) proposées par Golfarelli et Rizzi [GR99]. Le schéma XML résultat de la fusion est finalement transformé en un document XML de sortie. L'avantage de X-Warehousing par rapport à l'approche de Golfarelli *et al.* [GRV01] est que les étapes d'ETL sont effectués d'une façon complètement automatique.

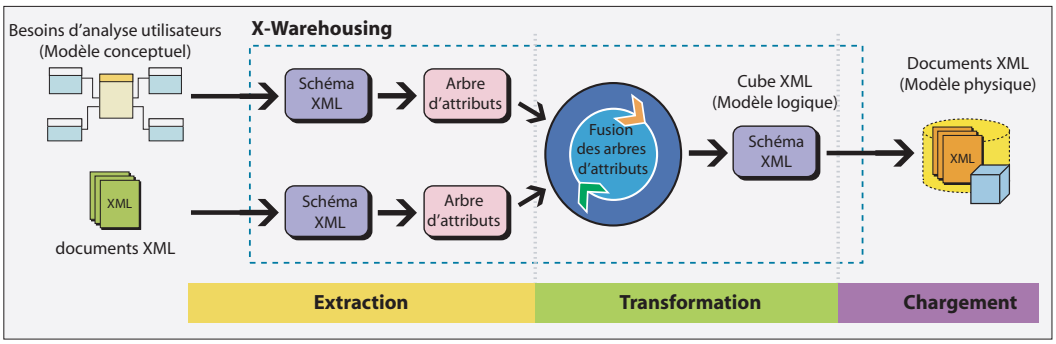



FIGURE 3.4 – Architecture de X-Warehousing [BBCA06]

Enfin, Vrdoljak *et al.* proposent une méthode pour la construction d'entrepôts de données XML issues du Web [VBR03]. À l'exemple de Golfarelli *et al.* [GRV01], les auteurs valident les données sources à l'aide de schémas XML déduits à partir des sources des données. Ces schémas XML sont ensuite transformés en graphes de données. La troisième étape est l'identification des faits. Enfin, pour chaque fait, on identifie les dimensions et le mesures avant de produire en sortie le schéma logique de l'entrepôt.

### 3.1.3 Modèle unifié d'entrepôt de données XML

Mahboubi *et al.* jugent nécessaire de disposer d'un modèle de référence pour les entrepôts de données XML [MRL+09]. Leur motivation principale provient du fait qu'aucun standard de modélisation des entrepôts de données XML n'est défini dans la littérature. Pour cela, Mahboubi *et al.* unifient les travaux qui traitent de l'entreposage de données XML (les entrepôts de données XML et les entrepôts de documents XML [MHD09]) et étendent, par la suite, les entrepôts de données XML présentés dans la littérature. Les auteurs proposent un modèle d'entrepôt de données XML basé sur un schéma conceptuel en constellation et permettant l'entreposage de données complexes [DBRA05]. Le modèle en question représente les

faits par un ensemble de documents XML et chaque dimension par un document XML. Plus précisément, il est composé des documents XML suivants :

- *dw-model.xml* qui stocke les métadonnées de l'entrepôt, c'est-à-dire son schéma. Ce document décrit la structure des documents faits (mesures et références aux dimensions) et dimensions (les niveaux hiérarchiques de chaque dimension et les attributs qui les caractérisent) ;
- $facts_f.xml$ est un ensemble de documents XML qui stocke les faits (les identificateurs des dimensions et les valeurs des mesures) ;
- un ensemble de documents XML, $dimension_d.xml$, qui représente les instances de chaque dimension $d$ de l'entrepôt.

### 3.1.4 Discussion

Dans le Tableau 3.1, nous évaluons les travaux sur l'entreposage de données XML selon quatre critères de comparaison : le modèle de données utilisé par l'approche, la représentation des faits et des dimensions, l'utilisation de la technologie ETL pour le chargement de l'entrepôt, et la présence (ou pas) d'un outil d'analyse associé à l'entrepôt de données.

| | **Modèle de données** | **Repr. des faits et des dim.** | **ETL** | **Outil d'analyse** |
|---|---|---|---|---|
| [GRV01] | non communiqué | non communiqué | oui | non |
| [Pok02] | en étoile | contraintes d'intégrité référentielles | oui | oui |
| [VBR03] | non communiqué | non communiqué | oui | non |
| [HBH03] | en étoile | document *XCubeSchema.xml* | non | oui |
| [RRT05] | en étoile | références des dimensions dans les faits | oui | non |
| [BBCA06] | en flocon de neige | non | oui | non |
| [MRL+09] | en étoile | attribut *dim-id* | oui | non |

TABLE 3.1 – Comparaison des entrepôts de données XML

**Modèle de données et représentation des faits et des dimensions.** Une première partie des travaux sur l'entreposage de données XML a choisi de modéliser les données multidimensionnelles dans un seul fichier XML [BBCA06]. D'autres ont choisi de modéliser les faits et les dimensions dans des documents XML séparés [Pok02, HBH03, RRT05, MRL+09]. Les travaux de Golfarelli *et al.* [GRV01] et de Vrdoljak *et al.* [VBR03] sont les seuls travaux qui ne présentent pas de modèle multidimensionnel. Le modèle de données suivi dans la majorité des approches est le modèle en étoile. Boussaid *et al.* utilise une dérivation du modèle en étoile : le modèle en flocon de neige [BBCA06].

Modéliser les données multidimensionnelles en XML signifie que, dans le ou les documents XML, il faut assurer les relations entre faits et dimensions. Or, dans les modèles multidimensionnels où les faits et les dimensions sont représentées dans un seul fichier XML, aucune technique n'est nécessaire pour la représentation des faits et des dimensions. Les dimensions sont représentées comme les éléments enfants des éléments faits, de même pour les éléments d'une dimension hiérarchisée [BBCA06].

Dans les autres modèles multidimensionnels qui représentent les faits et les dimensions dans des fichiers séparés, les auteurs définissent les méthodes nécessaires assurer les relations entre faits et dimensions dans les fichiers XML. Par exemple, Mahboubi *et al.* définissent un attribut *dim-id* dans chaque élément *dimension* du document de faits [MRL+09]. L'attribut *dim-id*, appelé clé de référence virtuelle par l'auteur, fait référence à l'attribut *id* du document correspondant à la dimension en question. La Figure 3.5 illustre l'architecture globale de l'entrepôt de données XML de Mahboubi *et al.* [MRL+09].

**Utilisation des outils d'ETL.** À l'exception du travail de Hümmer *et al.*, les travaux sur l'entreposage de données XML ont employé les méthodes d'ETL afin de charger les entrepôts de données XML. Les méthodes adoptées différent d'une approche à l'autre : semi-automatique dans certaines approches comme dans le travail de Golfarelli *et al.* [GRV01] ou automatique comme dans X-Warehousing [BBCA06]. Le travail de Hümmer *et al.* est le seul travail [HBH03] qui n'utilise pas un processus d'ETL dans son approche puisqu'il vise à partager des données déjà entreposées.

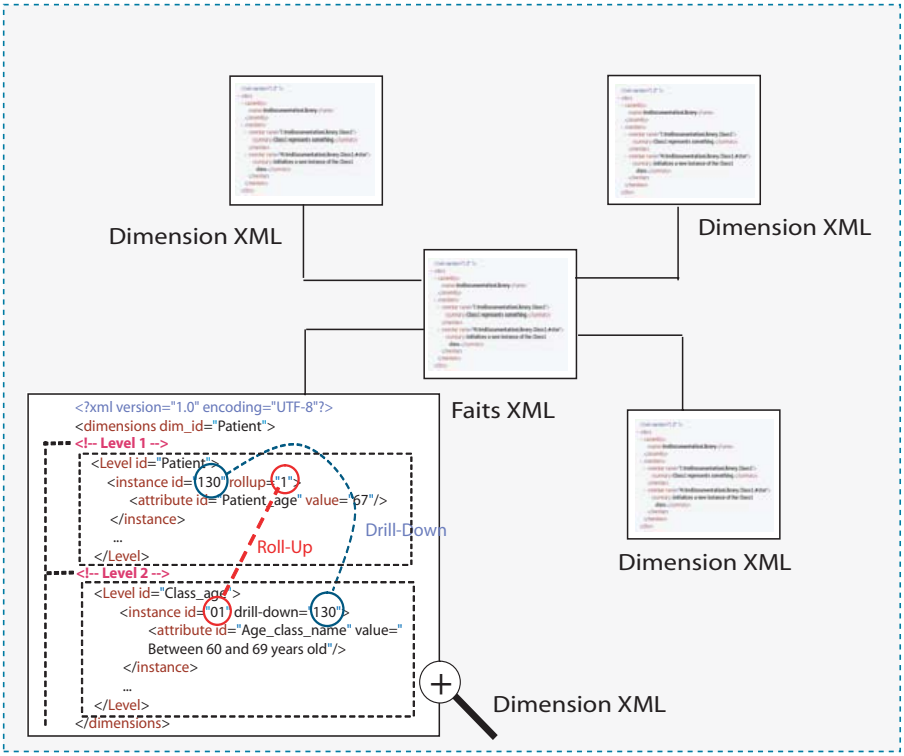



FIGURE 3.5 – Architecture globale de l'entrepôt de données XML de Mahboubi *et al.* [MRL+09]

**Analyse.** En plus des entrepôts de données XML, certaines approches proposent (ou utilisent) un langage d'interrogation XML pour réaliser des requêtes décisionnelles sur les données XML entreposées.

Dans le but de rendre son approche plus expressive, Pokorný propose de spécifier des restrictions sur les hiérarchies de dimensions [Pok02]. Pour réaliser cet objectif, l'auteur utilise le langage XQL [3].

Hümmer *et al.* utilisent le langage de requêtes XCubeQuery pour formuler des requêtes sur un cube de données et de ses méta-données sur le réseau [HBH03]. Ce langage permet d'interroger les faits et les dimensions (incluant les hiérarchies). La Figure 3.6 représente un exemple de sortie d'une requête XCubeQuery sur le document des faits. L'exemple de la Figure 3.6 montre que le format du fichier de sortie est différent du format du document de faits en entrée de la Figure 3.2(c).

3. `http://www.w3.org/TandS/QL/QL98/pp/xql.html`

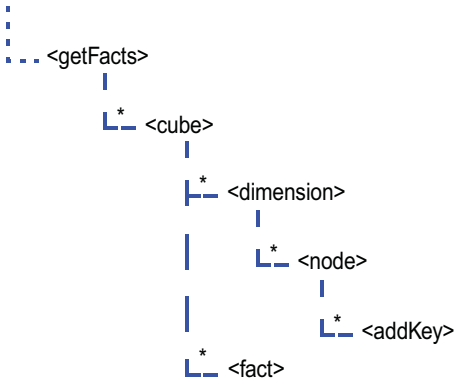



FIGURE 3.6 – Structure du document XCubeQuery.getFacts [HBH03]

Enfin, Mahboubi *et al.* utilisent le langage XQuery pour la formulation des requêtes d'analyse [MRL+09]. La Figure 3.7 donne un exemple de requête XQuery décisionnelle de type agrégation, jointure et sélection.

**for** \$x **in** document($fact_{sales}.xml$)//FactDoc/Fact,
\$y **in** document($dimension_{customer}.xml$)
//dimensions[@dim-id='Customer']/Level/instance
**where** \$y/attribute[@id='c_city']/@value='Lyon'
**and** \$x/dimension[@dim-id='Customer']/@value-id=\$y/@id
**return** name='cust_name', **aggregation**(sum, quantity)

FIGURE 3.7 – Exemple de requête décisionnelle XQuery [MRL+09]

## 3.2 De l'OLAP à l'XOLAP

Dans cette section, notre objectif est de montrer comment sont nés des environnements XOLAP à partir des algèbres OLAP proposées dans la littérature. Nous commençons donc par définir les opérateurs OLAP classiques. Dans une seconde étape, nous étudions et discutons les différentes algèbres OLAP. Nous étudions ensuite les environnements XOLAP, et notamment les modèles de données et les opérateurs proposés [HMD07].

### 3.2.1 Opérateurs OLAP classiques

Dans ce qui suit, nous définissons les opérateurs OLAP classiques, classés par famille. Chaque opérateur prend en entrée un cube OLAP et fournit un autre cube en sortie. Par définition, un cube OLAP représente dans ses cellules des faits à analyser (matérialisés par des mesures numériques) en fonction de dimensions (axes d'analyse) décrites par des attributs membres et susceptibles d'être hiérarchisées.

Soit l'entrepôt de ventes de la Figure 2.5. Dans le cube de données de la Figure 3.8, nous représentons des faits *ventes* décrits selon les dimensions *ville*, *livre* et *année*.

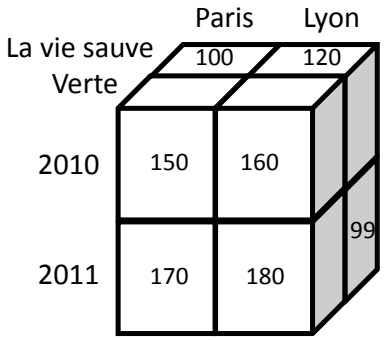



FIGURE 3.8 – Exemple de cube de données de ventes

Chacune de ces dimensions est caractérisée par deux modalités : *ville*(Paris, Lyon), *livre*(La vie sauve, Verte) et *année*(2010, 2011). La dimension *catégorie* n'est pas représentée ici. La mesure est *prix*.

#### Opérateurs structurels

**Rotation (*Rotate*/*Pivot*) :** *Rotate*(C, D), dit aussi *Pivot*, permute les dimensions d'un cube C autour d'une d'entre elles (D). La *rotation* est une sorte de sélection de faces et non de membres ; elle déplace deux dimensions ou plus dans le cube sans modifier les valeurs des mesures correspondantes. Dans la Figure 3.9, nous représentons une rotation du cube de la Figure 3.8 selon l'axe *produit* en appliquant ce principe.

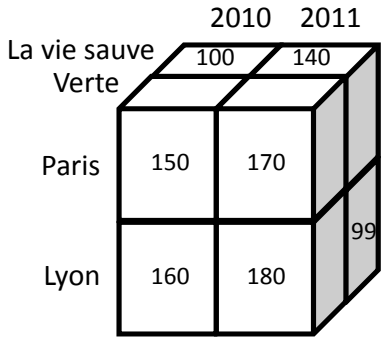



FIGURE 3.9 – Exemple de rotation du cube selon la dimension *livre*

**Changement de position des modalités d'une dimension (*Switch*) :** *Switch*(C, D, A, O) échange les positions d'un ensemble d'attributs A au sein d'une dimension D d'un cube C selon l'ordre O. Comme la rotation, cette opération se caractérise principalement par un effet visuel, puisque les valeurs des mesures ne sont pas modifiées avec le changement de position des faits correspondants.

**Transformation d'une dimension en mesure (*Push*) :** *Push*(C, D, A) consiste à combiner un ensemble A de membres d'une dimension D aux mesures du cube C, c'est-à-dire à transformer ces membres en contenu de cellules de C.

**Transformation d'une mesure en dimension (*Pull*) :** Opérateur réciproque de *Push*, *Pull*(C, M, $f$, D) transforme une mesure M, à l'aide d'une fonction de conversion $f$, en dimension D du cube C selon les exigences du modèle de données support de l'algèbre qui la met en œuvre.

**Opérateurs ensemblistes**

**Découpage du cube en tranches (*Slice*) :** *Slice*(C, D, A) dissocie une ou plusieurs "tranches" d'un cube de données multidimensionnel C selon un ensemble d'attributs A d'une seule dimension D, et ce pour tous les attributs des autres dimensions. Un cube peut en effet être vu comme un ensemble de tranches superposées les unes sur les autres (horizontalement) ou les unes à côté des autres (verticalement). Le *découpage du cube en tranches* s'effectue donc dans l'un de ces

deux sens. Dans la Figure 3.10, nous découpons le cube *ventes* de la Figure 3.8 selon le *livre* "Verte" en appliquant ce principe.

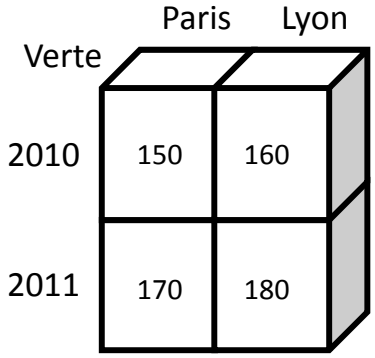



FIGURE 3.10 – Exemple de découpage de cubes en tranches selon le *livre* "Verte"

**Extraction d'un dé de cube (*Dice*) :** *Dice*(C, P) extrait un sous-cube du cube de données initial C selon un ensemble P de prédicats définis sur ses dimensions. Cette opération est assurée en sélectionnant les mesures du cube pour un certain nombre d'attributs dimensionnels, qui doit être le même pour tous les axes (dimensions) du cube.

### Opérateurs liés à la granularité

**Forage vers le haut (*Roll-up*) :** *Roll-up*(C, D, $n$, $f_{ag}$) représente les données du cube C à un niveau $n$ de granularité supérieur selon une dimension D et conformément à sa hiérarchie. Le but est de passer d'un niveau de granularité à un autre plus général ; par exemple, d'une représentation multidimensionnelle des données suivant les villes à une autre suivant les pays. Ce passage nécessite l'association d'une fonction d'agrégation $f_{ag}$ (somme, moyenne, etc.) à l'opération de groupement pour indiquer comment sont calculées les valeurs du niveau supérieur à partir de celles du niveau inférieur. Techniquement, un *forage vers le haut* consiste à regrouper les différentes tranches du cube en une seule tout en suivant la direction de l'axe représentant la dimension concernée. Chaque cellule du résultat final contient le résultat de l'agrégation des mesures correspondantes.

**Forage vers le bas (*Drill-down*) :** Défini comme l'opérateur réciproque du *Roll-up*, *Drill-down*(C, D, $n$) permet d'obtenir des détails sur la signification d'un résultat en affinant une dimension D du cube C à un niveau d'agrégation inférieur $n$ ; par exemple, passer d'une représentation multidimensionnelle des données suivant les pays à une autre représentation suivant les villes. D'un point de vue technique, si le forage vers le haut permet de compresser un cube, le *forage vers le bas* le développe en détaillant les données suivant une ou plusieurs dimensions.

**Construction de cube (*Cube*) :** *Cube*(C, D, $f_{ag}$) peut être défini comme une opération d'agrégation généralisant le *regroupement*, le *forage vers le haut* et la *jointure* sur un cube de données afin de produire un cube plus général [GBLP96]. En premier lieu, un nombre de forages vers le haut égal au nombre de dimensions du cube est réalisé sur le cube initial C, pour chacune de ses dimensions $D_i \in \mathbb{D}$, où $\mathbb{D}$ est l'ensemble des dimensions de C. Dans une deuxième étape, afin de joindre les différents résultats de forage dans leurs dimensions communes, un calcul des mesures du forage est exigé pour chaque attribut de chaque axe, selon une fonction d'agrégation $f_{ag}$ (somme, moyenne, etc.). Pour compléter le cube, il faut finalement grouper toutes les mesures du cube en une seule cellule en utilisant la même fonction d'agrégation. L'association de cette cellule avec les deux résultats précédents permet d'obtenir le cube de données agrégé.

### 3.2.2 Algèbres OLAP

Malgré le développement des systèmes décisionnels, les bases de données multidimensionnelles et la technologie OLAP souffrent d'un manque de normalisation. Nous nous intéressons dans ce chapitre aux travaux qui proposent des modèles pour les données multidimensionnelles supportant une algèbre OLAP.

Il existe plusieurs déclinaisons de l'OLAP qui sont liées à la façon de représenter et de stocker les données. Nous distinguons trois déclinaisons ici : ROLAP (*Relational OLAP*), MOLAP (*Multidimensional OLAP*) et HOLAP (*Hybride de ROLAP et MOLAP*). La différence majeure entre ROLAP et MOLAP se résume dans la façon de représentation et de stockage de données multidimensionnelles. Les schémas d'entrepôt utilisés dans les approches ROLAP sont les schémas en étoile et ses dérivations (les schémas en flocon de neige ou en constellation). Dans les

approches MOLAP, des structures spécialisées (inspirées des cubes de données) sont définies pour représenter et stocker les données multidimensionnelles.

Dans ce qui suit, nous classifions les algèbres OLAP dans deux familles : (1) approches ROLAP et (2) approches MOLAP.

### Approches ROLAP

Li et Wang formalisent un cube de données basé sur le modèle relationnel [LW96]. Ils proposent ainsi deux algèbres pour manipuler les données multidimensionnelles représentées dans leur modèle. La première algèbre, appelée algèbre de groupement (*grouping algebra*), représente une extension de l'algèbre relationnelle par une fonction de groupement. Cet algèbre concerne les relations de groupement, et elle est composée de trois opérateurs : *renommer*, *groupement avec ordre* (*roll*) et *agrégation*. Selon les auteurs, les relations de groupement permettent de grouper les données dans un cube. Plus simplement, une relation de groupement est équivalente au résultat de l'opération SQL "group by" sur une table relationnelle. *Renommer* permet de changer le nom d'un attribut de dimension. L'opérateur de *groupement avec ordre* (*roll*) permet de grouper, suivant un ordre ascendant (*asc*) ou descendant (*desc*), les instances d'un ou plusieurs attributs d'une relation de groupement. Enfin, l'opérateur d'*agrégation* permet de réaliser des opérations qui reflètent le nom de l'opérateur (somme, etc.) sur les attributs -déjà groupés- d'une relation de groupement.

Dans une deuxième étape, les auteurs définissent les cubes multidimensionnels comme l'association de deux ensembles disjoints : les dimensions et leurs attributs. Une dimension est représenté par une relation de groupement. Sur la base de ce modèle, ils définissent une deuxième algèbre multidimensionnelle, composée de six opérateurs. Le premier opérateur, *ajout d'une dimension*, permet de générer un nouveau cube C1 à partir d'un cube existant C0, en insérant une dimension dans ce dernier. Le *transfert* change l'emplacement d'un attribut dans une dimension, pour mieux visualiser les relations entre les attributs. L'*union* de deux cubes, selon une ou plusieurs dimensions de jointure, est possible s'ils disposent de la même structure. Plusieurs opérations d'agrégation sont possibles sans passer par des opérations de groupement. L'*agrégation de cubes* permet de réduire la taille d'un cube en compressant les mesures relatives à ses dimensions. L'opérateur *rc-*

*join* permet de joindre une relation avec une dimension d'un cube. Finalement, la *construction* génère un cube à partir d'une relation. Les dimensions et les mesures concernées sont spécifiées en entrée de l'opérateur.

Dans un second travail, et avec pour objectif l'analyse en ligne OLAP, Gyssens et Lakshmanan modélisent les bases de données multidimensionnelles en tables relationnelles [GL97]. Cette représentation conserve les noms des tables de faits et de leurs attributs, pour les attributs dimensionnels comme pour les mesures.

Très proche de l'algèbre relationnelle, cette algèbre se compose de deux ensembles d'opérateurs. Le premier ensemble repose sur des *opérateurs classiques* repris de l'algèbre relationnelle (*union*, *intersection*, *différence* et *produit cartésien*) et les *opérateurs unaires* de *sélection*, de *projection* et de *renommage*. Chacun de ces opérateurs est défini de la même façon que dans le cas relationnel. Le second ensemble d'opérateurs propose deux *opérateurs de restructuration*, *unfold* et *fold*, qui permettent de remanier la structure d'un cube. *Unfold* ajoute une nouvelle dimension dans le schéma relationnel d'un cube. Réciproquement, l'élagage d'une dimension d'un cube est assuré par *fold*.

### Approches MOLAP

Gyssens *et al.* proposent une algèbre pour les données multidimensionnelles modélisées de façon tabulaire [GLS96]. Rangées dans des tableaux bidimensionnels (ou matrices), les données peuvent être des noms, des valeurs numériques ou nulles. En l'absence d'une structure figée, les tables peuvent prendre plusieurs formes tout en respectant l'aspect multidimensionnel des données. Les auteurs définissent ensuite une algèbre tabulaire et établissent une catégorisation des opérateurs.

Les opérateurs traditionnels proposés sont l'*union*, la *différence*, le *produit cartésien*, le *renommage*, la *projection* et la *sélection*. Ces derniers fonctionnent de la même façon que leurs équivalents en algèbre relationnelle. D'autres opérateurs de restructuration sont proposés : le *regroupement* (*grouping*), la *fusion* (*merging*), l'*éclatement* (*splitting*) et la *compression* (*collapsing*). Le *regroupement* et la *fusion* (respectivement l'*éclatement* et la *compression*) peuvent être vus comme l'inverse l'un de l'autre. Le *regroupement* permet de réunir, dans un tableau, les informations selon un certain attribut. La fusion effectue l'opération réciproque. L'*éclatement* permet de transformer une table en plusieurs. Par contre, la *com-*

*pression* fusionne un ensemble de tables en une seule. Il est clair que *regroupement* et *fusion* sont deux opérateurs qui manipulent essentiellement les données, alors qu'*éclatement* et *compression* manipulent leur structure. Grâce à l'opérateur de *transposition* (*transposing*), il est possible d'exprimer, pour toute opération sur les lignes d'une matrice, une opération similaire sur ses colonnes. Finalement, afin d'offrir une meilleure représentation des données, la *suppression des redondances* (*clean-up*) élimine dans une table les valeurs nulles.

Agrawal *et al.* proposent un modèle pour les bases de données multidimensionnelles et les opérateurs algébriques liés [AGS97]. Ils modélisent les cubes multidimensionnels sous forme de repères à trois dimensions et proposent sur cette base une algèbre riche d'opérateurs inspirés essentiellement du relationnel, dont la sortie représente un nouveau cube. Conçus pour être traduits en SQL, ces opérateurs peuvent être mis en œuvre via un SGBD relationnel ou via un autre moteur supportant les données multidimensionnelles.

L'opérateur *push* transforme une dimension en mesure. L'opération réciproque est assurée par *pull*. L'opérateur de *destruction d'une dimension* supprime une dimension d'un cube, à condition que cette dernière ne dispose que d'une seule valeur. La *restriction* élimine d'une dimension les attributs qui ne satisfont pas un prédicat prédéfini. Le principe relationnel de *jointure* est transcrit en reliant deux cubes via leurs dimensions communes, tout en respectant une condition de jointure. L'opérateur *fusion* (*merge*) permet de réunir deux dimensions d'un cube en une seule et de calculer les mesures résultantes par l'intermédiaire d'une fonction d'agrégation.

Les auteurs définissent ensuite des opérateurs OLAP basés sur cette algèbre. Une *projection* sur un cube s'effectue en fusionnant les dimensions non concernées par cette opération en une seule, puis en la détruisant. L'*union* est assurée en joignant les valeurs non nulles de deux cubes ayant le même nombre de dimensions et utilisant les mêmes types de données. L'*intersection* s'effectue en joignant deux cubes dans les valeurs communes des dimensions. La *différence* entre deux cubes C1 et C2 est assurée par une intersection entre eux suivie d'une union du résultat avec le premier cube C1.

Cabibbo et Torlone proposent un modèle logique des systèmes OLAP pour la conception des bases de données multidimensionnelles [CT97, CT98]. Dans ce modèle, les dimensions sont représentées en hiérarchies, suivant un ordre prédéfini

exprimé par une relation de forage vers le haut. Une famille de fonctions, appelées *fonctions de forage vers le haut* (*roll up functions*), complète la définition d'une dimension. Le schéma d'un modèle multidimensionnel consiste alors en un ensemble fini de dimensions et un ensemble fini de tables de faits. Malgré l'intérêt de leur approche, Cabibbo et Torlone se limitent à la formalisation de cette famille de *fonctions de forage vers le haut* et ne proposent pas d'autre opérateur.

Datta et Thomas proposent un modèle de cube de données supportant une algèbre OLAP [DT99]. Ce modèle de cube respecte la symétrie entre les dimensions et les mesures, comme la symétrie entre les agrégations de types *forage vers le haut* (*roll up*) et *valeurs numériques* (*somme*, *moyenne*, etc.), ou encore entre les *transformations* (*convertir une dimension en mesure* et, vice versa, *la conversion d'une mesure en dimension*). Cette symétrie est à l'origine des opérateurs OLAP proposés par les auteurs. Ces derniers définissent un cube comme un quadruplet <D, M, A, $f$>. D et M désignent les dimensions et les mesures, respectivement. Chaque D ou M est définie par un nom relatif à un domaine prédéfini. Les ensembles des noms des dimensions et des noms d'attributs sont disjoints. A représente un groupe d'attributs, chacun étant défini par son nom, extrait du domaine de noms d'attributs. Enfin, $f$ représente une fonction de D vers A qui permet d'établir la correspondance entre les dimensions et les attributs. Les auteurs modélisent aussi un cube matérialisé dans l'ensemble <D, M, A, f, V, $g$>. V représente l'ensemble des valeurs utilisées dans le cube, tandis que $g$ symbolise la combinaison des domaines des noms des dimensions associées à ces valeurs.

Sur ce modèle, l'algèbre des auteurs propose neuf opérateurs dédiés à la manipulation OLAP. Le premier opérateur, *restriction*, permet de limiter un ensemble des valeurs V pour obtenir en sortie un ensemble V0 inclut ou égal à V. L'opérateur *agrégation* permet d'agréger une mesure en groupant les attributs de la dimension correspondante. Le *produit cartésien* relie deux cubes en unifiant les dimensions d'une part et les mesures d'autre part. La *jointure* est un cas particulier du produit cartésien, conçue dans le but de relier deux cubes ayant une ou plusieurs dimensions communes. Les deux opérateurs suivants nécessitent que les deux cubes en question soient union-compatibles. L'*union* permet d'unir deux cubes. C'est une opération différente du produit cartésien puisqu'il s'agit d'unifier les ensembles de valeurs de deux cubes en une seule valeur V0 dans le cube résultant. La *différence* consiste à soustraire l'ensemble de valeurs correspondant au deuxième cube du pre-

mier. La *partition* classe les cellules d'un cube dans des groupes significatifs. Cette tâche concerne certains types d'agrégation. Par exemple, il est possible de grouper toutes les moyennes selon un prédicat prédéfini. Les deux derniers opérateurs, *pull* et *push*, peuvent être classés comme des opérateurs de transformation. *Pull* permet de convertir les mesures en dimensions selon une fonction de transformation. *Push* permet de réaliser l'opération inverse en utilisant aussi une autre fonction de transformation dédiée.

Un second travail des mêmes auteurs utilise ce modèle et propose une autre algèbre de neuf opérateurs [TD01]. Sept d'entre eux ne sont pas modifiés. La *restriction*, l'*agrégation*, l'*union* et la *différence* ont conservé les mêmes noms et principes. Le *produit cubique*, les *extractions des dimensions* et *des mesures* remplacent le *produit cartésien*, *pull* et *push*, respectivement. Deux nouveaux opérateurs sont proposés : la *projection métrique* et le *renommage*. La *projection métrique* réduit le nombre d'attributs correspondant aux mesures prises en compte dans la représentation des données. Le *renommage* permet de rebaptiser les éléments d'un ensemble d'attributs ou de caractéristiques (mesures ou dimensions).

Ravat *et al.* proposent un modèle qui sert de support à une algèbre et à un langage graphique pour l'analyse et la visualisation des données [RTZ06a, RTZ06b]. Leur algèbre intègre les opérateurs OLAP classiques complétés par des opérateurs avancés afin de faciliter l'expression d'analyses décisionnelles complexes. Le modèle repose sur une représentation multi-faits où chaque fait est analysé en fonction d'axes d'analyse (dimensions) multi-vues (multi-hiérarchisées). Les auteurs modélisent un cube de données par une structure de visualisation proche des arbres d'attributs, sous la forme d'un tableau à double entrée hiérarchisé appelé table multidimensionnelle (TM) [RTZ01].

L'algèbre proposée repose sur un *opérateur de construction* (*display*) qui produit une TM à partir d'une base de données multidimensionnelle et sur un ensemble de *onze opérateurs fondamentaux* portant sur les TM et facilitant la manipulation OLAP. Le premier opérateur, la *rotation*, permet de remplacer un axe d'analyse par un autre, ou de changer la hiérarchie sur un même axe au sein d'une TM. Les opérateurs *forages vers le bas* et *vers le haut* (*drill down* ou *roll up*) autorisent la modification des différents niveaux de granularité utilisés pour visualiser les données au sein d'une TM. Les dimensions de la TM peuvent être enrichies

par des attributs supplémentaires en employant l'opérateur d'*imbrication* (*nest*). La *sélection* restreint l'ensemble des valeurs affichées des attributs dimensionnels et des mesures correspondantes. L'opérateur d'*organisation* (*switch*) intervertit deux valeurs d'un attribut d'une dimension pour permettre l'ordonnancement des valeurs affichées. Le *calcul d'agrégats* permet d'ajouter dans une TM des calculs agrégeant ses lignes et/ou ses colonnes. L'opérateur de *conversion d'un paramètre en mesure* (*push*) transforme un paramètre (dimension) en mesure au sein de la TM. La *conversion d'une mesure en paramètre* (*pull*) transforme une mesure en paramètre (dimension). Enfin, les opérateurs d'*ajout* (*addm*) et de *suppression* (*delm*) *de mesures* permettent de modifier l'ensemble des mesures calculées.

Ben Messaoud étend OLAP avec de nouveaux opérateurs de fouille de données en ligne [Ben06]. L'objectif de cette extension est d'établir un cadre formel général, non seulement pour la structuration et la navigation, mais aussi pour la description, la classification, l'explication et la prédiction dans les données multidimensionnelles. L'auteur améliore le modèle de Datta et Thomas [DT99] par un *espace de données multidimensionnelles*, c'est-à-dire par un espace de modalités et un ensemble de fonctions d'agrégation. Sur la base de ce modèle, deux familles d'opérateurs sont proposés :

- opérateurs de structuration : *construction d'un cube de données*, *ajout d'une caractéristique*, *suppression d'une caractéristique*, *imbrication d'un attribut*, *conversion d'une dimension* et *conversion d'une mesure* ;
- opérateurs de navigation : *forage vers le haut*, *forage vers le bas*, *permutation de modalités*, *sélection de modalités* et *calcul d'agrégats*.

*Construction d'un cube de données* (*ConstructCube*) permet de créer un cube à partir d'un espace de données multidimensionnelles. *Ajout d'une caractéristique* (*AddCharacteristic*) rajoute une nouvelle caractéristique dans un cube. L'opération inverse est assurée par *Suppression d'une caractéristique* (*DltCharacteristic*). *Imbrication d'un attribut* (*Nest*) permet d'intégrer un attribut dans une dimension. *Sélection de modalités* (*Select*) consiste à sélectionner, dans un cube, un certain nombre de modalités d'une caractéristique. *Calcul d'agrégat* (*Aggregate*) permet de construire de nouveaux agrégats dans le cube. Enfin, *Conversion d'une dimension* (*Push*), *conversion d'une mesure* (*Pull*), *forage vers le haut* (*RollUp*), *forage vers le bas* (*DrillDown*), *permutation de modalités* (*Switch*) correspondent à leurs homonymes dans les opérateurs OLAP classiques.

### Discussion

Dans le Tableau 3.2, nous évaluons les algèbres OLAP présentés ci-dessus selon cinq critères de comparaison.

| | Modèle de données | Modélisation des hiérarchies de dimensions | Fonctions d'agrégation | Noyeau min. fermé | Combinaisons complexes |
|---|---|---|---|---|---|
| [LW96] | ROLAP | Oui | Oui | Non | Non |
| [GL97] | ROLAP | Non | Non | Non | Non |
| [AGS97] | MOLAP | Oui | Oui | Non | Non |
| [GLS96] | MOLAP | Non | Non | Non | Non |
| [CT97, CT98] | MOLAP | Oui | Oui | Non | Non |
| [DT99, TD01] | MOLAP | Oui | Oui | Oui | Oui |
| [RTZ06a, RTZ06b] | MOLAP | Oui | Oui | Oui | Oui |
| [Ben06] | MOLAP | Oui | Oui | Oui | Oui |

TABLE 3.2 – Comparaison des algèbes OLAP

**Modèle de données.** Dans les travaux que nous venons de présenter, chaque algèbre est basée sur un modèle de données spécifique. Bien que tous ces modèles partent du principe d'analyser des faits OLAP selon plusieurs dimensions, leur conception diffère d'une algèbre à l'autre. Nous retrouvons ici les deux grandes familles de modèles supportées par les différentes algèbres. La première famille modélise les données multidimensionnelles dans un format relationnel classique [LW96, GL97]. Au contraire, dans la deuxième famille, les concepts relationnels sont abstraits de toute représentation logique ou physique des données [GLS96, AGS97, CT97, CT98, DT99, TD01, RTZ06a, RTZ06b, Ben06]. Par exemple, dans leur travail, Agrawal *et al.* modélisent les données en un espace de représentation multidimensionnel sous forme d'axes de dimensions, sans se référer à des tables. Les autres auteurs ont aussi laissé de côté la représentation tabulaire des données,

à l'exception de Cabibbo et Torlone, qui modélisent les données multidimensionnelles dans des tables (mais ces dernières sont loin d'être relationnelles). Les auteurs définissent, pour chaque dimension, un ensemble de niveaux organisés selon un ordre partiel. Plus simplement, Cabbibo et Torlone définissent dans ces tables un sens de lecture des données différent de celui des données relationnelles.

**Opérateurs.** Outre les opérateurs OLAP classiques et les opérateurs inspirés du relationnel, peu d'opérateurs ont été proposés pour enrichir l'analyse en ligne en général. Parmi ceux-ci, nous pouvons cependant citer la *suppression des redondances* dans l'algèbre de Gyssens *et al.* [GLS96] ou encore les *opérateurs de restructuration* (*unfold* et *fold*) de Gyssens et Lakshmanan [GL97]. L'inconvénient de ces nouveaux opérateurs est qu'ils sont totalement dépendants du modèle de données. Leur utilisation dans le cadre d'une autre algèbre OLAP nécessitait donc une adaptation. De plus, nous pouvons également noter que ces opérateurs ont des synonymes dans d'autres algèbres. Par exemple, *unfold* est équivalent à l'opérateur *add dimension* de Li et Wang [LW96], tandis que *fold* est équivalent à *destroy dimension* de Gyssens *et al.* [GLS96].

**Granularité et hiérarchies de dimension.** Comme le montre le Tableau 3.2, les travaux qui prennent en compte les hiérarchies de dimension dans leurs modèles ont aussi mis en place un certain nombre d'opérateurs algébriques afin de manipuler la granularité des données. Gyssens *et al.* sont les seuls auteurs qui ne considèrent pas du tout la granularité des données dans leurs différents opérateurs [GLS96, GL97].

En effet, dans ces travaux, la prise en compte de la granularité des données est toujours accompagnée par une implémentation des opérateurs qui prennent en compte les hiérarchies de dimensions dans leur fonctionnement (*forage vers le haut* et/ou *vers le bas* et parfois l'opérateur *cube*). Agrawal *et al.* expriment le *forage vers le haut* en associant une fonction d'agrégation, comme *somme*, à l'opérateur *fusion* de leur algèbre [AGS97]. Ils expriment aussi le *forage vers le bas* en effectuant l'opération réciproque du *forage vers le haut*, c'est-à-dire en "remontant en arrière" pour savoir comment sont réalisées l'agrégation et la fusion des données. Li et Wang présentent, dans leur algèbre de groupement, un opérateur de *forage vers le haut* (*roll*) qui se repose sur une opération de *groupement* [LW96]. Les *fonctions*

*de forage vers le haut* de Cabibbo et Torlone [CT97, CT98] permettent également d'atteindre ce but, mais les auteurs ne donnent malheureusement pas plus de détails sur leur fonctionnement. Bien que Datta et Thomas [DT99, TD01] ne présentent pas d'*opérateur de forage* ni d'*agrégation*, ils affirment que leur algèbre peut supporter ces opérations. Enfin, Ravat *et al.* développent des *opérateurs de forage vers le haut* et *vers le bas* [RTZ06a, RTZ06b] qui ne s'éloignent pas, dans leur fonctionnement, de leurs équivalents dans les autres algèbres.

Certains auteurs proposent des opérateurs d'agrégation et de forage vers le haut qui reposent sur des fonctions d'agrégation externes à leurs modèles respectifs. Agrawal *et al.* expliquent ce choix par le souci de conserver un nombre limite d'opérateurs pour des raisons de simplicité [AGS97]. D'autres auteurs ont choisi d'exprimer de tels opérateurs en combinant certains opérateurs de leur algèbre. Datta et Thomas, Ravat *et al.* et Ben Messaoud ont proposé des opérateurs complexes construits par combinaisons d'opérateurs de base, comme les opérateurs de forage [DT99, TD01, RTZ06a, RTZ06b, Ben06]. Ces combinaisons complexes font de ces travaux [DT99, TD01, RTZ06a, RTZ06b, Ben06] les seules approches qui présentent un noyau minimal fermé d'opérateurs. Par définition, le noyau minimal fermé d'une algèbre OLAP comporte un nombre minimal et suffisant d'opérateurs algébriques capables d'assurer toutes les manipulations possibles des données multidimensionnelles [Ben06].

### 3.2.3 Approches XOLAP

Dans cette Section, nous nous intéressons aux travaux qui abordent l'analyse OLAP sur des données XML, c'est-à-dire les approches qui ont défini un environnement XML-OLAP (ou XOLAP) et les opérateurs définis dans ces approches.

#### Opérateurs XOLAP inspirés de l'OLAP classique

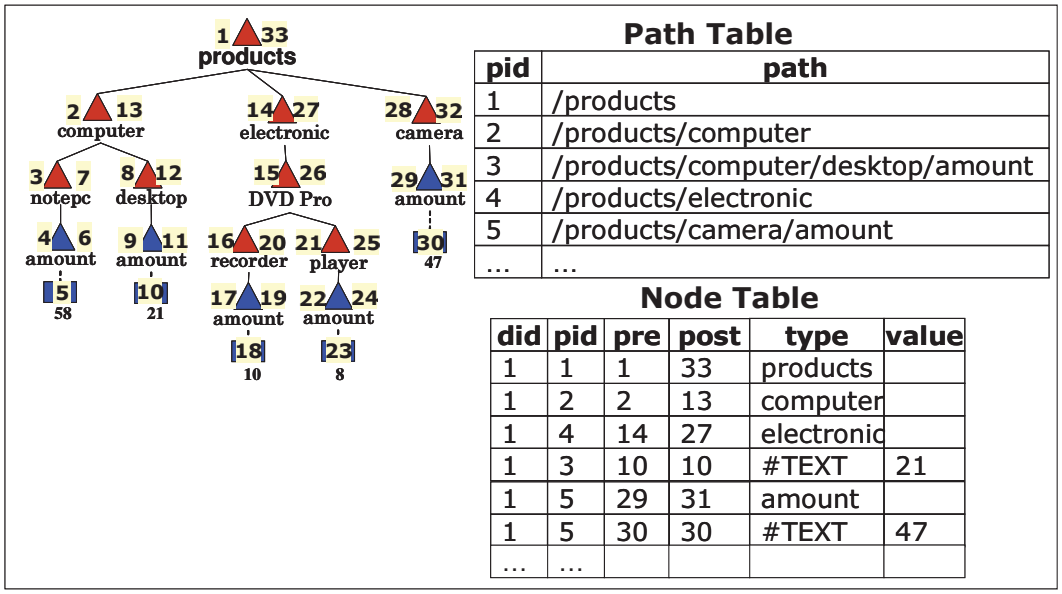
Dans cette Section, nous étudions les approches qui ont présenté un ensemble d'opérateurs XOLAP inspiré des opérateurs OLAP classiques présentés à la Section 3.2.1.

Dans un premier travail, nous avons choisi d'exprimer les opérateurs OLAP classiques à l'aide de l'algèbre XML TAX [HMD08]. Notre motivation principale provenait du fait que, à l'époque, il n'existait pas d'approche qui présente des

opérateurs OLAP classiques sur des données XML. Après une étude approfondie des différentes algèbres XML présentées dans la littérature [FSW00, JLST01, PAKJ+02, ZPR02, FHP02, BFLC03, ZY04, Pra06], nous avons conclu que TAX était l'algèbre XML la plus riche en terme d'opérateurs, surtout ceux qui pouvaient aider à exprimer les opérateurs OLAP classiques comme la *sélection*, le *groupement* ou aussi le *calcul d'agrégat*. Dans une étape suivante, nous avons modélisé les données multidimensionnelles à l'aide d'arbres XML TAX. Nous avons appelé ces arbres les arbres TAX multidimensionnels (ou aussi les arbres TAX XOLAP). La racine d'un arbre TAX multidimensionnel désigne les analyses à effectuer (*ventes*, par exemple). Chaque sous-arbre de l'arbre TAX multidimensionnel représente un fait (*vente*, par exemple). Les nœuds enfants d'un fait modélisent les dimensions et les mesures. En cas de dimension exprimée en hiérarchie, la hiérarchie en question est exprimée dans un arbre séparé ; et en cas de besoin, nous réalisons une jointure entre cet arbre et l'arbre TAX multidimensionnel. Sur la base de ce modèle, nous avons exprimé les opérateurs OLAP classiques à l'aide d'un ou plusieurs opérateurs TAX. Chacun de ces opérateurs XOLAP respecte la définition de base présentée dans la Section 3.2.1. Par exemple, nous exprimons l'opérateur *rotation* par une sélection TAX. *Rotation* sélectionne tous les faits en permutant les dimensions en question dans l'arbre de données autour d'une autre dimension. Nos opérateurs XOLAP sont implémentés avec le langage PHP derrière une interface Web qui permet à l'utilisateur de sélectionner les opérateurs et leurs paramètres. Par exemple, pour la *rotation*, l'utilisateur sélectionne les dimensions à permuter et la dimension axe de permutation. L'application génère finalement des requêtes XQuery.

Kit fait appel à l'approche de chemins (*path-approach* [YASU01]) pour le stockage relationnel de données XML [Kit10]. L'approche de chemins permet de stocker les données XML sous format relationnel et de les interroger à l'aide du langage SQL. Dans ce modèle, un nœud XML est transformé en deux n-uplets dans deux tables relationnelles : la table de chemins qui contient les expressions de chemins absolus pour tous les nœuds XML ; et la table de nœuds qui contient toutes les informations sur les nœuds XML. Soit l'exemple de la Figure 3.11. Cet exemple modélise le document "products.xml" (représenté en arbre de données) dans les deux tables de chemins (*Path Table*) et de nœuds (*Node Table*). Dans cet exemple, et afin d'assurer le stockage des données XML dans les tables relationnelles, chaque

nœud XML est étiqueté avec deux nombres : un pré-ordre et un post-ordre. Afin de réduire le coût élevé de cette technique, l'auteur utilise plusieurs techniques d'optimisation de stockage de données XML [YC08, MLC09, BC09].




**Path Table**

| pid | path |
|---|---|
| 1 | /products |
| 2 | /products/computer |
| 3 | /products/computer/desktop/amount |
| 4 | /products/electronic |
| 5 | /products/camera/amount |
| ... | ... |

**Node Table**

| did | pid | pre | post | type | value |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 33 | products | |
| 1 | 2 | 2 | 13 | computer | |
| 1 | 4 | 14 | 27 | electronic | |
| 1 | 3 | 10 | 10 | #TEXT | 21 |
| 1 | 5 | 29 | 31 | amount | |
| 1 | 5 | 30 | 30 | #TEXT | 47 |
| ... | ... | | | | |

FIGURE 3.11 – Exemple de storage de données XML sous format relationnel [Kit10]

Dans une seconde étape, Kit propose les opérateurs *rotation*, *découpage de cube en tranches*, *détachement d'un dé de cube*, *forage vers le haut* et *forage vers le haut topologique* (*topological rollup*) avec le langage XQuery. Par exemple, comme la *rotation* consiste à changer l'ordre suivant lequel sont représentées les dimensions, l'auteur exprime l'ordre final dans la clause RETURN de la requête XQuery. Afin de pouvoir exprimer les opérations de *forage*, l'auteur étend le langage XQuery avec une fonction de regroupement GROUP BY [KAK08, KAK09]. La spécificité du *forage vers le haut topologique* est qu'il permet de prendre en compte plusieurs niveaux d'agrégation dans la hiérarchie XML. Selon l'auteur, son seul inconvénient qu'il est coûteux à réaliser. À la base, l'opérateur *forage vers le haut topologique* a été introduit par Bordawekar et Lang, mais sans être implémenté [BL05]. Un exemple d'une opération de forage vers le haut topologique est illustré dans la Figure 3.12 sur les données de l'arbre "products.xml" de la Figure 3.11.

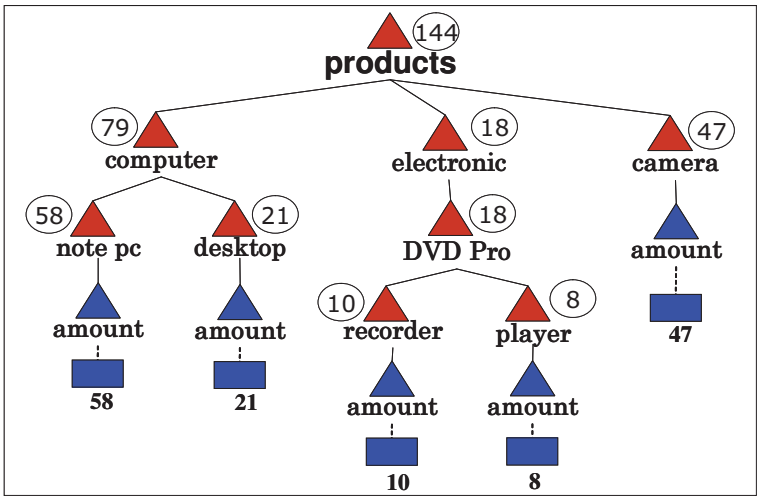



FIGURE 3.12 – Exemple de forage vers le haut topologique (*topological roll-lup*) [Kit10]

Egho présente un ensemble d'opérateurs XOLAP comme une solution pour exploiter des corpus linguistiques [Egh10]. À la base, Les données linguistiques sont stockées dans des documents XML. Cette approche se résume dans quatre étapes :

1. la modélisation relationnelle de données XML linguistiques : l'auteur prend en compte la nature de données et la façon dont lesquelles elles sont modélisées dans des documents XML en même temps. En fait, chaque document XML modélise un corpus. Un corpus inclut des enregistrements, des transcriptions, des blocs, des productions verbales et des évènements (mots, pauses, descriptions, chevauchements). Chaque composant du corpus est modélisé en une table relationnelle tout en respectant la hiérarchie des données. Ce modèle est implémenté à la fin dans une base de données MySQL [4].
2. la transformation des données : ici, l'auteur calcule le temps de chaque production verbale et de chaque événement. il calcule aussi la durée des pauses entre les mots, le nombre de mots, le nombre de descriptions et le nombre de chevauchement.
3. le chargement des données dans dans des cubes de données : afin d'atteindre

4. `http://www.mysql.fr`

cet objectif, l'auteur prend en compte les besoins de l'utilisateur, le laboratoire ICAR [5], qui sont de la forme :

- Combien de fois un mot donné est répété durant une période donnée ?
- Combien de fois l'interlocuteur prononce un mot donné ?
- Combien de pauses ont été réalisées dans une phrase donnée ? etc.

Par la suite, les valeurs numériques à calculer (le nombre, le minimum, le maximum, etc.) représente les faits à analyser. Les dimensions sont les mots, les pauses, les productions verbales, etc. Un exemple de cube de données qui permet de modéliser les nombres de mots par production verbale est illustré dans la Figure 3.13.

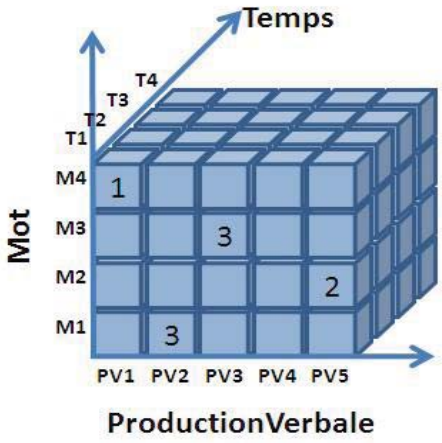



FIGURE 3.13 – Exemple de cube de données linguistiques [Egh10]

4. la définition d'un ensemble d'opérateurs pour cubes de données : au début, l'auteur étend le langage XQuery avec un opérateur de groupement *XGroup By*. Après, il exprime les opérateurs *rotation*, *découpage de cube en tranches*, *détachement d'un dé de cube*, *forage vers le haut* et *forage vers le bas* avec XQuery. Les opérateurs de *forage* sont basés sur la fonction *XGroup By*.

### Analyse multidimensionnelle de données XML

Jensen *et al.* proposent un travail qui se résume en trois points [JMP01]. Ils présentent tout d'abord une architecture d'intégration de données XML et

5. `http://icar.univ-lyon2.fr`

relationnelles. Ils proposent ensuite une modélisation multidimensionnelle des bases de données (XML/relationnelles) OLAP avec UML. Enfin, ils listent les considérations nécessaires à prendre en compte pour concevoir des bases de données XML supportant des analyses OLAP. En sortie du système d'intégration proposé, les données sont présentées en relationnel et évaluées avec le langage de requêtes objet OQL (*Object Query Language*). L'architecture de Jensen *et al.* est illustrée dans la Figure 3.14. Dans la même logique, Niemi *et al.* présentent une architecture d'intégration de données XML et un modèle de base de données OLAP exploités par MDX [6], le langage multidimensionnel de Microsoft [NNNT02]. Dans ces deux travaux, les opérateurs OLAP spécifiques à XML sont absents, puisque les données OLAP sont exploitées avec des langages de requêtes pour les données relationnelles. Les données XML sont traduites en relationnel avant leur utilisation au sein de ces architectures d'intégration.

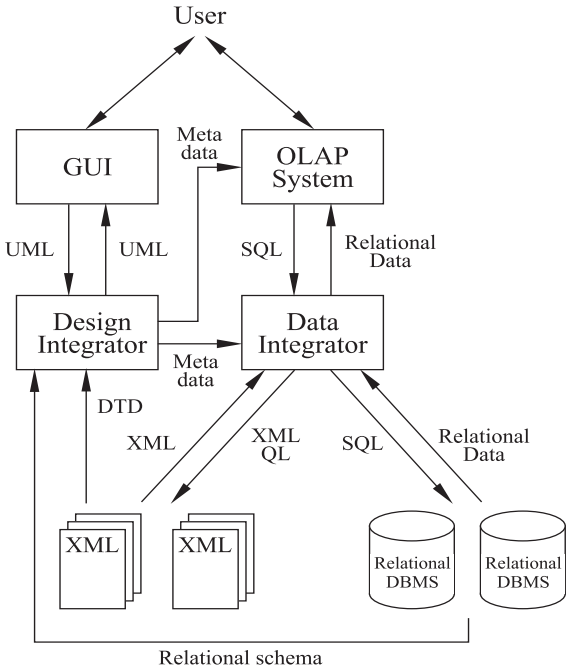



FIGURE 3.14 – Architecture d'intégration de données XML et relationnelles de Jensen *et al.* [JMP01]

6. `http://msdn.microsoft.com/en-us/library/ms145973.aspx`

Pedersen *et al.* intègrent des données XML et relationnelles au sein d'un même système [PRP02b, PP03, PPP04]. De plus, ils proposent une approche pour fédérer des sources de données XML avec des cubes OLAP existants. La fédération XML-OLAP se compose d'un cube, d'une collection de documents XML et de liens entre le cube et les documents. Un cube multidimensionnel est décrit par son nom et ses tables de dimensions et de faits. Chaque dimension est vue comme une hiérarchie de niveaux, où chaque niveau est associé à un groupe de valeurs dimensionnelles. La table de faits est une relation contenant un attribut pour chaque dimension et un attribut pour chaque mesure. Ce modèle de cube est associé à une algèbre composée de trois opérateurs. L'opérateur le plus fondamental dans une fédération OLAP-XML, la *décoration*, attache une nouvelle dimension à un cube en se basant sur les valeurs des éléments XML liés. La *sélection* sert à filtrer les mesures. L'opérateur de *projection généralisée* (*generalized projection*) a pour rôle l'agrégation des mesures. Cette opération est vue comme une opération SQL de *sélection* employant une clause de *groupement* (*group by*). Elle est réalisée en trois étapes. Au début, les dimensions qui ne font pas parti de l'opération sont supprimées. Ensuite, et pour les dimensions restantes, les valeurs sont agrégés au niveau supérieur indiqué (comme le cas du forage vers le haut). Finalement, la projection généralisée retourne un résultat. En cas de hiérarchies non-strictes ou non-couvrantes, les faits concernés seront perdus et le résultat retourné est égal à $\emptyset$. Ces opérateurs sont mis en œuvre sur les données OLAP-XML à l'aide une extension de $\text{SQL}_M$, nommée $\text{SQL}_{XM}$, qui permet d'associer des requêtes XPath [7] aux requêtes $\text{SQL}_M$. $\text{SQL}_M$ est lui-même une extension de SQL pour le traitement des données multidimensionnelles [PRP02a].

### Agrégation, groupement et XOLAP

Park *et al.* proposent un cadre pour l'analyse en ligne OLAP des données XML appelé XML-OLAP et introduisent la notion de cube XML (XQ-Cube) [PHS05]. La Figure 3.15 illustre sur le modèle de l'entrepôt de données XML de Park *et al.* Dans ce modèle multidimensionnel, chaque fait (et chaque dimension) est représenté par une (autre) collection de documents XML.

Afin d'interroger l'entrepôt de données XML, les auteurs proposent un langage

7. `http://www.w3.org/TR/xpath`

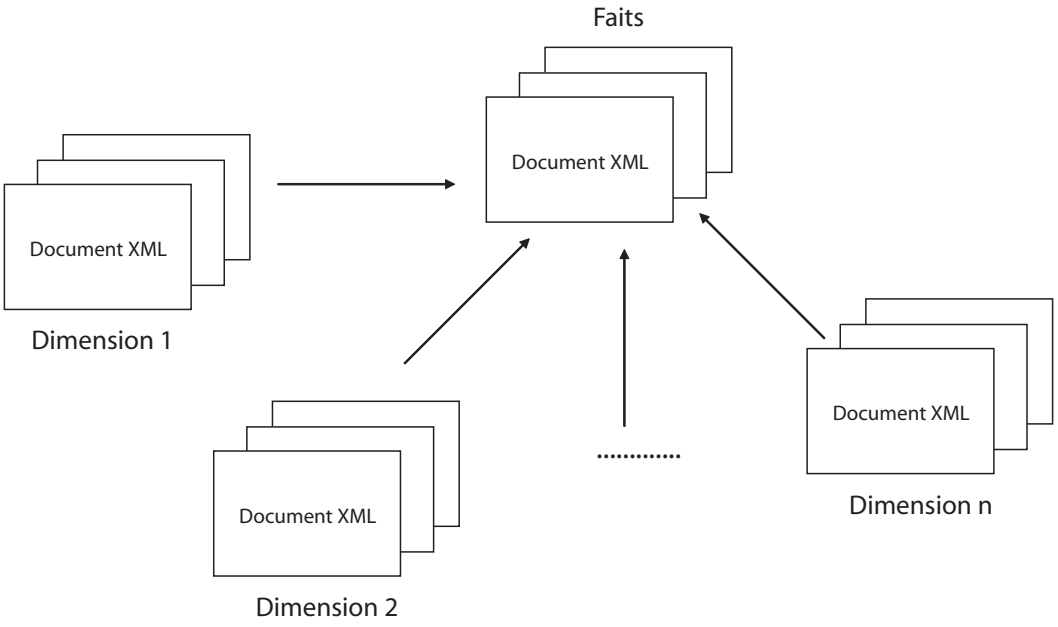



FIGURE 3.15 – Modèle multidimensionnel d'entrepôt XML-OLAP [PHS05]

pour les expressions multidimensionnelles, XML-MDX, inspiré du langage d'expressions multidimensionnelles MDX de Microsoft. Ils implémentent la création de cubes XML (CREATE XQ-CUBE) et leur interrogation (SELECT). En outre, ils proposent sept opérateurs d'agrégation : *ajout* (ADD), *listage* (LIST), *comptage* (COUNT), *résumé* (SUMMARY), *extraction du sujet* (TOPIC), *extraction du plus important mot-clé* (TOP KEYWORD) et *classification* (CLUSTER). Certains opérateurs sont inspirés du relationnel, d'autres sont conçus pour les données non-additives et exploitent des techniques de fouille de textes (*text mining*).

Dans le même contexte, Wang *et al.* présentent un deuxième environnement pour XOLAP [WLHG05]. Ils définissent un opérateur général d'agrégation pour XML, *GXaggregation*, qui forme la base de *XCube*, une extension de l'opérateur traditionnel cube pour les données XML. Mis en œuvre avec une extension de XQuery, *GXaggregation* permet l'extraction de propriétés à partir des dimensions et des mesures suivant leurs expressions de chemin XPath. Par conséquent, calculer des statistiques sur les données XML devient plus aisé. Enfin, ce processus est associé à des fonctions qui agrègent des données hétérogènes sur des hiérarchies.

Beyer *et al.* affirment que les requêtes analytiques exprimées en XQuery sont

difficiles à lire, à écrire et à manipuler d'une façon efficace [BCC+05]. Afin d'améliorer cela, ils étendent les expressions FLWOR de XQuery avec une syntaxe explicite de *regroupement* et de *comptage* des résultats. Face aux problèmes générés par le groupement avec XQuery, ils présentent également des solutions originales traitant l'aspect hétérogène et hiérarchique des données XML.

Ben Messaoud *et al.* proposent un opérateur d'agrégation OLAP basé sur une méthode automatique de classification (*clustering*) baptisé *OpAC* [BBLR06]. Cet opérateur permet de réaliser des analyses précises et fournit les agrégats sémantiques sur des données complexes représentées dans des documents XML.

Finalement, Wiwatwattana *et al.* affirment que les extensions du modèle relationnel ne peuvent pas prendre en compte toutes les spécificités de XML [WJLS07]. Ils proposent en conséquence un entrepôt de treillis de cubes XML, un opérateur cube intitulé X^3 et un mécanisme de spécification généralisé. Ils discutent également le mécanisme de construction de cube et comparent plusieurs algorithmes alternatifs. X^3 a été implémenté en C++ au sein du SGBD natif XML TIMBER [PAKC+03]. Ce travail est une extension de l'algèbre XML TAX [JLST01].

### Discussion

Dans le Tableau 3.3, nous évaluons les approches XOLAP selon quatre critères de comparaison : le modèle de données adopté, les opérateurs présentés dans l'approche, le langage de requêtes proposé pour formuler les opérations XOLAP et le type de hiérarchies (simples ou complexes) prises en compte dans l'approche.

**Modèle de données.** La majorité des approches XOLAP que nous avons étudié dans la Section 3.2.3 ont utilisé des systèmes d'intégration de données relationnels. Pour faciliter la formulation des requêtes OLAP sur les données du système d'intégration relationnel-XML, Jensen *et al.*, Pedersen *et al.* ainsi que Niemi *et al.* offrent à l'utilisateur la possibilité de formuler ces requêtes avec des langages de requête relationnels. Toutefois, l'approche de Pedersen *et al.* n'ignore pas complètement les données XML puisque ces dernières ne sont pas traduites en données relationnelles au sein de leur système. Lorsqu'une requête $\mathrm{SQL}_{XM}$ est destinée aux données XML, le système la traduit en requête XPath [PPP04]. Les travaux de Pedersen *et al.* constituent les premières véritables tentatives de définition d'un environnement XOLAP. Nous pensons que la force de leur ap-

| | Modèle de données | Opérateurs | Langage | Hiérarchies |
|---|---|---|---|---|
| [JMP01, NNNT02] | Intégration XML/rel. | aucun | OQL/MDX | simples |
| [PRP02b, PRP02a, PP03, PPP04] | Intégration XML/rel. | décoration (jointure), projection généralisée, sélection | $SQL_{XM}$ | complexes |
| [PHS05] | Cubes XML | cube, sélection, agrégation | XML-MDX | simples |
| [BBLR06] | Documents XML | agrégation | Non défini | simples |
| [WLHG05] | Documents XML | agrégation, cube | XQuery | simples |
| [BCC+05] | Documents XML | groupement, comptage | XQuery | complexes |
| [WJLS07] | Treillis de cubes XML | cube | XQuery | simples |
| [HMD08] | Documents XML | tous les opérateurs OLAP classiques | XQuery | simples |
| [Kit10] | Relationnel | rotation, découpage du cube en tranches, extraction d'un dé de cube, forage vers le haut, forage vers le haut topologique | XQuery | simples |
| [Egh10] | Intégration XML/rel. | rotation, découpage du cube en tranches, extraction d'un dé de cube, forages vers le haut et vers le bas | XQuery | simples |

TABLE 3.3 – Comparaison des approches XOLAP

proche est le couplage entre (1) la facilité de formulation de requêtes avec un langage familier pour l'utilisateur et (2) la prise en compte des données XML au sein du système. Les autres auteurs ont choisi de modéliser les données multidimensionnelles directement dans des documents XML [BCC+05, PHS05, WLHG05, BBLR06, WJLS07, HMD08]. La majorité de ces travaux utilise le langage XQuery

pour analyser les données dans les documents XML.

**Opérateurs et hiérarchies.** Même s'ils constituent les premières véritables tentatives de définition d'un environnement XOLAP, les travaux de Pedersen *et al.* présentent seulement deux opérateurs exprimés en SQL. Les travaux de la Section 3.2.3 ont adopté des structures multidimensionnelles de données représentées dans des documents ou des cubes XML. À l'exception du travail de Beyer *et al.*, la prise en compte de la granularité des données dans les opérateurs proposés est claire grâce à la présence d'opérateurs comme *agrégation* ou *cube*. L'ensemble d'opérateurs disponibles reste cependant très restreint. Les approches les plus récentes ont proposé de nouveaux opérateurs inspirés de l'OLAP classique afin de permettre à l'utilisateur de réaliser des opérations similaires à l'OLAP classique [HMD08, Kit10, Egh10].

Enfin, nous remarquons que la majorité des opérateurs XOLAP existants ne couvrent que les hiérarchies simples. Les travaux de Pedersen *et al.* et de Beyer *et al.* sont les seuls travaux qui traitent les hiérarchies complexes [PRP02a, PPP04, BCC+05]. L'opérateur d'agrégation (*projection généralisée*) de Pedersen *et al.* détecte en effet les hiérarchies non-strictes et non-couvrantes [PRP02a, PPP04]. Dans les deux cas, il retourne $c = \emptyset$. Dans leur travail, Pedersen *et al.* distinguent entre trois types de données : les données qui peuvent être agrégatives et additives, les données qui peuvent être agrégatives mais pas additives, et les données qui ne peuvent être ni agrégatives ni additives. Le troisième type correspond aux données multiples dus aux hiérarchies non-strictes. Dans ce cas, la fonction d'agrégation retourne $c = \emptyset$. Beyer *et al.* formulent des requêtes de forage vers le haut et de construction de cube sur des hiérarchies complexes (*raggaed hierarchies*) à l'aide de leur extension du langage XQuery avec une fonction de groupement [BCC+05]. Parmi tous les travaux sur XOLAP alors, ces deux approches [PRP02a, PPP04, BCC+05] sont les seuls qui détectent les hiérarchies complexes (qui sont à l'origine de problèmes d'additivité). Par contre, ces approches ne présentent aucune solution aux problèmes d'additivité à part signaler la présence des hiérarchies complexes en retournant un résultat vide. C'est pour cela que, dans cette thèse, nous nous sommes intéressés à la définition d'un environnement (modèle de données et un ou plusieurs opérateurs) XOLAP dans le but de résoudre les problèmes d'additivité.

## 3.3 Problèmes d'additivité dans les modèles multidimensionnels

Dans cette Section, nous étudions les problèmes d'additivité dans les modèles multidimensionnels. Nous commençons par définir la notion d'additivité (Section 3.3.1). Ensuite, nous présentons les différentes contraintes qui permettent d'assurer l'additivité dans un modèle multidimensionnel (Section 3.3.2). La Section 3.3.3 est consacrée aux approches qui proposent des modèles multidimensionnels pour supporter les hiérarchies de dimension complexes et qui présentent une solution pour les problèmes d'additivité à travers ces modèles. Nous présentons dans les Sections 3.3.3, 3.3.4, 3.3.5 et 3.3.6 les solutions présentées dans la littérature pour les problèmes d'additivité. Nous concluons et discutons l'ensemble des travaux sur l'additivité dans la Section 3.4.3.

### 3.3.1 Définition de l'additivité

La notion d'additivité (*summarizability*) a été introduite par Rafanelli et Shoshani dans le contexte des bases de données statistiques [RS90] et analysée par Lenz et Shoshani dans le contexte des bases de données multidimensionnelles [LS97]. Soit une dimension $D_i$, appartenant à l'ensemble $\mathbb{D}$ des dimensions d'un modèle multidimensionnel et exprimée en hiérarchie. L'additivité désigne le *calcul exact* des valeurs agrégées à un niveau d'agrégation donné ($H_{ij} \in D_i$) à partir des valeurs agrégées à un niveau d'agrégation inférieur ($H_{ik} \in D_i \ / \ k < j$) [RS90]. Ce calcul n'est autre que l'opération OLAP de forage vers le haut ou *roll-up* (Section 3.2.1).

Notons que le terme additivité que nous utilisons dans ce mémoire est la traduction imparfaite du terme anglo-saxon *summarizability* [MLT09]. Dans la littérature, le terme anglo-saxon *additivity* désigne un cas particulier de *summarizability* lorsque la fonction d'agrégation utilisée est la somme (*sum*) [HSC04]. Dans ce mémoire, nous étudions les problèmes d'additivité (*summarizability problems* [MLT09]) qui concernent toutes les fonctions d'agrégation, y compris la somme.

### 3.3.2 Contraintes d'additivité

Dans le contexte des bases de données statistiques, Rafanelli et Shoshani introduisent trois contraintes nécessaires à la garantie de l'additivité.

1. Les cardinalités de type plusieurs à plusieurs ne doivent pas être utilisées, c'est-à-dire que la cardinalité maximale associée au niveau supérieur (dans une association entre un niveau inférieur et un niveau supérieur d'une hiérarchie) ne peut être que 1.
2. Toutes les associations entre les niveaux hiérarchiques doivent être de type un à plusieurs.
3. Dans une hiérarchie de dimension, les relations de type zéro à plusieurs ne doivent pas être utilisées : toute instance du niveau inférieur doit être liée, au moins, à une instance au niveau supérieur.

Lenz et Shoshani concluent que l'additivité est un sujet très sensible dans les bases de données multidimensionnelles et les analyses OLAP [LS97].

Négliger ce problème peut facilement provoquer des erreurs d'analyse qui par la suite influencent la prise de décision. Lenz et Shoshani définissent donc trois conditions nécessaires pour garantir l'additivité dans le contexte des bases de données multidimensionnelles. La première contrainte est la disjonction (*disjointness*) et correspond à la contrainte (1) de Rafanelli et Shoshani. La deuxième contrainte, la complétude (*completeness*) correspond à la contrainte (3) de Rafanelli et Shoshani. Enfin, les auteurs introduisent une troisième et nouvelle contrainte, la compatibilité de type (*type compatibility*), qui signifie que la fonction d'agrégation choisie doit être compatible avec le type de données de la mesure.

Dans la littérature, plusieurs travaux se sont inspirés des travaux de Rafanelli et Shoshani [RS90] et de Lenz et Shoshani [LS97] pour définir des contraintes d'additivité [LAW98, NNT01, NNT03, LV03]. L'ensemble de ces contraintes confirme que les hiérarchies non-strictes et/ou les hiérarchies non-couvrantes provoquent des problèmes d'additivité. Lehner *et al.* définissent des dépendances fonctionnelles, inspirées de la théorie des bases de données relationnelles, comme contraintes d'additivité [LAW98]. Ils définissent des contraintes fortes pour désigner les hiérarchies non-strictes et des contraintes faibles pour désigner les hiérarchies non-couvrantes. Niemi *et al.* enrichissent les dépendances fonctionnelles de Lehner *et al.* par des nouvelles contraintes, les dépendances booléennes [NNT01, NNT03].

Par exemple, soit la hiérarchie *ville-région-pays*. Au cas où une ville n'appartient à aucune région, la dépendance booléenne permet de choisir entre *région* ou *pays* comme niveau supérieur de *ville*. La contrainte booléenne s'écrit alors : *ville* $\rightarrow$ *région OR pays*. Lechtenbörger et Vossen s'inspirent aussi du travail de Lehner *et al.*, ainsi que des bases de données relationnelles, pour introduire trois formes normales multidimensionnelles comme contraintes d'additivité [LV03]. La première forme normale regroupe les deux premières contraintes de Rafanelli et Shoshani [RS90]. Les deuxième et troisième formes normales regroupent les contraintes de Lenz et Shoshani [LS97] et les troisième et quatrième contraintes de Rafanelli et Shoshani [RS90], respectivement.

Nous concluons donc que, pour garantir l'additivité dans un modèle multidimensionnel, deux contraintes majeures doivent être vérifiées : les associations entre les niveaux hiérarchiques d'une dimension doivent être à la fois strictes (contrainte de disjonction) et complètes (contrainte de complétude). La présence de hiérarchies complexes (non-strictes et non-couvrantes à la fois) dans un modèle multidimensionnel provoque des problèmes d'additivité. Dans ce qui suit, nous l'illustrons à partir de cas réels.

**Hiérarchies de dimension non-strictes et problèmes d'additivité**

Soit l'association entre les deux niveaux hiérarchiques *semaine* et *mois* de la dimension *temps* de la Figure 2.5. Dans la vie de tous les jours, nous disons couramment qu'un mois se compose de quatre semaines. Or, en réalité, quatre semaines correspondent à vingt-huit jours, ce qui est strictement inférieur au nombre de jours d'un mois (généralement 30 ou 31 jours). De plus, le premier du mois ne correspond pas toujours au premier jour de la semaine (le lundi). Donc, un mois est composé de quatre semaines et une partie de la cinquième semaine, elle-même première semaine du mois suivant. En conclusion, une semaine peut appartenir à plusieurs mois et un mois contient plusieurs semaine. Donc, l'association entre *semaine* et *mois* est de type plusieurs à plusieurs, ce qui signifie que la hiérarchie est non-stricte.

Supposons maintenant que nous voulions calculer le total des ventes d'un produit donné pour les mois de juin et de juillet 2011 (Tableau 3.4(b)) à partir des ventes par semaine (Tableau 3.4(a)). Le Tableau 3.4(b) montre que si nous comp-

tons la cinquième semaine deux fois (une fois pour chaque mois), nous obtenons un total incorrect (celui de la Table 3.4(a)). Cet exemple, inspiré d'un cas réel, illustre comment les hiérarchies non-strictes posent des problèmes d'additivité.

| Semaine | Ventes |
|---|---|
| S1-06-2011 | 10 |
| S2-06-2011 | 10 |
| S3-06-2011 | 10 |
| S4-06-2011 | 10 |
| S5-06-2011 | 10 |
| S2-07-2011 | 10 |
| S3-07-2011 | 10 |
| S4-07-2011 | 10 |
| S5-07-2011 | 10 |
| Total | 90 |

(a) Ventes par semaine

| Mois | Ventes |
|---|---|
| 06-2011 | 50 |
| 07-2011 | 50 |
| Total | 100 |

(b) Ventes par mois

TABLE 3.4 – Hiérarchies non-strictes et problèmes d'additivité

**Hiérarchies de dimension non-couvrantes et problèmes d'additivité**

Soit l'association entre les deux niveaux hiérarchiques *région* et *pays* de la dimension *temps* de la Figure 2.5. Les cardinalités entre *région* et *pays* (zéro à plusieurs) reflètent le cas réel où certains pays du monde ne contiennent pas de région (comme Monaco, qui est à la fois une ville et un état). Cette relation est donc non-couvrante.

Les Tableaux 3.5(a) et 3.5(b) représentent les sommes des ventes d'un produit donné par région et par pays, respectivement. La somme des ventes par pays est différente de la somme des ventes par région car il n'existe pas de régions à Monaco. Cet exemple, inspiré d'un cas réel, démontre que les hiérarchies non-couvrantes provoquent les problèmes d'additivité.

| Région | Ventes |
|---|---|
| Ile-de-France | 200 |
| Rhône-Alpes | 200 |
| Bavière | 200 |
| Total | 600 |

(a) Ventes par région

| Pays | Ventes |
|---|---|
| France | 400 |
| Allemagne | 200 |
| Monaco | 200 |
| Total | 800 |

(b) Ventes par pays

TABLE 3.5 – Hiérarchies non-couvrantes et problèmes d'additivité

**Discussion**

Toutes les contraintes d'additivité que nous avons étudiées jusqu'à maintenant concernent les différents niveaux d'une dimension organisée en hiérarchie. Ces contraintes sont appelées contraintes intradimensionnelles par Lehner *et al.* [LAW98]. Lehner *et al.* introduisent également un autre type de contraintes d'additivité : les contraintes sur les relations entre faits et dimensions dans un modèle multidimensionnel [LAW98]. Même si certains auteurs identifient les relations inter et intradimensionnelles comme des problèmes séparés [LAW98, MLT08, MLT09], ces relations sont de même nature et les solutions pour régler les problèmes d'additivité sont exactement les mêmes [PJD99].

Le Tableau 3.6 présente une classification des différentes associations qui peuvent exister entre deux niveaux hiérarchiques d'une dimension et leur implication dans les problèmes d'additivité. "Niveau $s$" et "Niveau $i$" désignent, respectivement, les niveaux supérieur et inférieur. Le terme "simple" désigne une hiérarchie à la fois stricte et couvrante, c'est-à-dire une hiérarchie qui ne pose aucun problème d'additivité. En cas de hiérarchie complexe, et suivant les informations du Tableau 3.6, la cardinalité minimale est égale à zéro et la cardinalité maximale est égale à plusieurs.

Le Tableau 3.7 présente une classification des différentes relations qui peuvent exister entre faits et dimensions et les origines des problèmes d'additivité. Les informations de ce tableau confirment notre avis sur le traitement des problèmes d'additivité dans les relations faits-dimensions puisque se sont toujours les cardinalités associées aux dimensions qui sont à l'origine des problèmes d'additivité.

Dans les associations fait-dimension, l'additivité est toujours assurée pour les

| | Cardinalité minimale | | Cardinalité maximale | |
|---|---|---|---|---|
| | zéro | un | un | plusieurs |
| Niveau $i$ | non-couvrante | simple | simple | simple |
| Niveau $s$ | non-couvrante | simple | simple | non-stricte |

TABLE 3.6 – Classification des associations entre niveaux hiérarchiques d'une dimension [MLT08]

faits. Nous désignons alors les associations fait-dimension de "simples" dans le Tableau 3.7 pour tout type de cardinalité côté fait. Par exemple, le fait qu'un produit donné n'est associé à aucune vente (cardinalité minimale 0) n'influence pas le calcul de la somme totale des ventes [MLT09].

| | Cardinalité minimale | | Cardinalité maximale | |
|---|---|---|---|---|
| | zéro | un | un | plusieurs |
| Fait | simple | simple | simple | simple |
| Dimension | non-couvrante | simple | simple | non-stricte |

TABLE 3.7 – Classification des associations fait-dimension [MLT08]

Dans les sections suivantes, nous étudions les solutions présentées dans la littérature pour les problèmes d'additivité dans les hiérarchies de dimension et les associations fait-dimension.

### 3.3.3 Modélisation des hiérarchies de dimensions complexes

Dans ce qui suit, nous étudions les solutions de modélisation multidimensionnelle des hiérarchies de dimension complexes. Le but de cette étude est de démontrer comment l'additivité est garantie à l'étape de modélisation.

Dans un premier travail, Malinowski et Zimányi présentent une classification des différents types de hiérarchies incluant les hiérarchies non-couvrantes et non-strictes [MZ04]. Le but de cette classification est de guider la conception du modèle multidimensionnel. Dans un article plus récent, les mêmes auteurs étendent le modèle entité-association (E/A) et définissent un nouveau modèle conceptuel,

*MultiDimER*, pour modéliser les différents types de hiérarchies [MZ06]. La Figure 3.16 représente les différentes notations utilisées dans *MultiDimER* : (a) les dimensions, (b) les faits et les mesures, (c) les hiérarchies, (d) les cardinalités et (e) les critères d'analyse. Nous nous intéressons, dans ce qui suit, à la façon avec laquelle les hiérarchies non-couvrantes et non-strictes ont été modélisées dans *MultiDimER*.

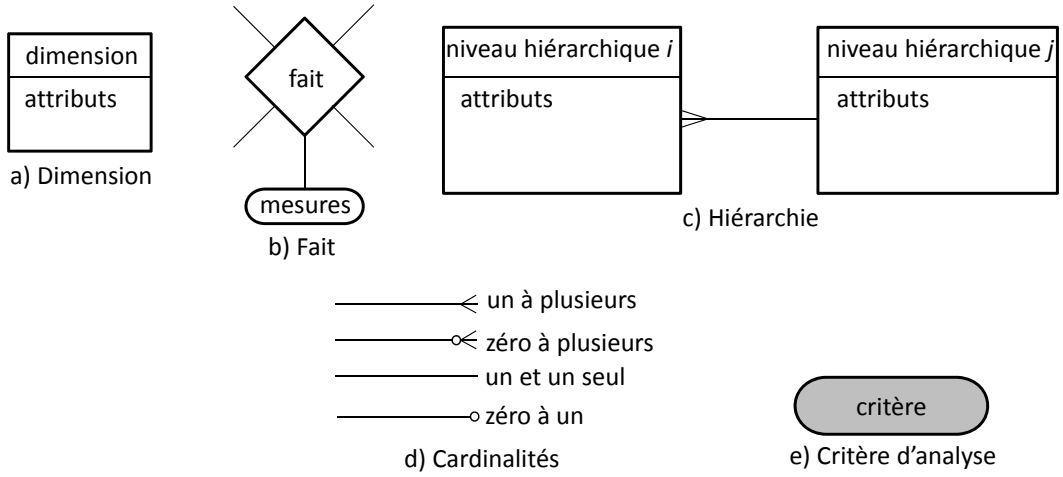



FIGURE 3.16 – Notations utilisées par Malinowski et Zimányi [MZ06]

Au niveau logique, les auteurs modélisent les hiérarchies non-onto de la manière suivante : ils remplacent les valeurs manquantes dans ces hiérarchies par des valeurs de substitution [PJD99] ou par des valeurs nulles (NULL) [CT00], comme le montre la Figure 2.4. Par exemple, soit la hiérarchie non-onto de la Figure 2.4. Les valeurs manquantes de cette hiérarchie sont remplacées par les valeurs "NULL" comme le montre la Figure 3.17. Afin d'éviter les problèmes d'additivité provoqués par la présence des hiérarchies non-strictes, Malinowski et Zimányi recommandent d'utiliser, au niveau logique, les tables de jointure (Section 3.3.4). Dans la littérature, d'autres travaux sur la modélisation des hiérarchies complexes existent, mais ignorent les problèmes d'additivité [ASS06, LMTS06].

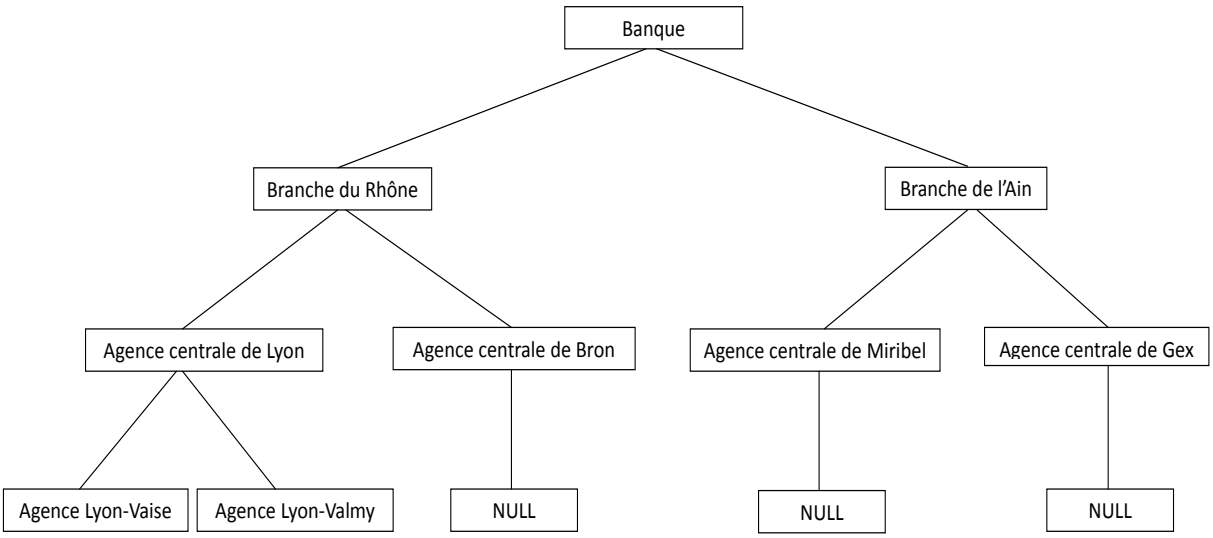



FIGURE 3.17 – Solution au niveau logique pour les hiérarchies non-onto [MZ06]

### 3.3.4 Normalisation des modèles multidimensionnels

La normalisation d'un modèle multidimensionnel est définie comme la transformation des hiérarchies de dimension (ou les associations fait-dimension) qui sont à l'origine des problèmes d'additivité en hiérarchies simples. Dans la littérature, deux stratégies sont utilisées dans le processus de normalisation.

Dans la première stratégie, le but est de définir des contraintes et des règles de transformation du modèle multidimensionnel. Hurtado *et al.* présentent un modèle, basé sur un graphe direct, pour modéliser les hiérarchies de dimension non-onto (dites hétérogènes dans ce travail) [HGM05]. Les auteurs définissent un graphe direct G par une paire d'ensembles (V, E). V regroupe l'ensemble des nœuds du graphe. E regroupe l'ensemble des arêtes directes qui relient entre ces nœuds. Hurtado *et al.* enrichissent ce schéma avec deux types de contraintes : *les contraintes de chemins* (*Path atoms*) et *les contraintes d'égalité* (*Equality atoms*). Les contraintes de chemins sont de la forme $[c, c_1,..., c_n]$ tel que $[c, c_1,..., c_n]$ est un chemin dans la hiérarchie de dimension. Les contraintes d'égalité sont de la forme $[c_1 = k]$ tel que $k \in$ M (M est un ensemble d'instances d'un membre d'une dimension). Les signes suivants sont autorisés dans la définition des contraintes : $\neg$ (négation), $\wedge$ (contraintes optionnelles), $\vee$ (différent de), $\Rightarrow$ (signifie que),

$\Leftrightarrow$ (équivaut à), $\oplus$ (association de contraintes), $\perp$ (faux), $\top$ (vrai) et $\bigodot_A$. $\bigodot_A$ signifie qu'il existe une seule vraie contrainte dans l'ensemble de contraintes $A$.

La Figure 3.18(a) illustre un graphe de *dimension géographique* avec un ensemble de quatre contraintes d'intégrité (a, b, c et d). Cet exemple suppose que des magasins se situent dans trois pays : la France, les États-Unis et le Canada. La contrainte (a) signifie que tous les magasins doivent être situés dans des villes. La contrainte (b) indique que si la ville est "Paris", le pays est "France". La contrainte (c) indique que la hiérarchie *ville-...-pays = "France"* est différente de la hiérarchie *ville-...-pays = "États-Unis"*. La contrainte (d) indique que si un magasin est situé dans une localité géographique de type *province-...-pays*, c'est qu'il est situé au Canada.

Ensuite, Hurtado *et al.* introduisent la notion de dimensions couvrantes minimales (*frozen dimensions*). Ces dimensions sont représentées avec les dimensions non-couvrantes. Le but de cette représentation est de rendre plus facile la compréhension des hiérarchies non-couvrantes. Par exemple, dans la Figure 3.18(b), la hiérarchie *magasin-ville-province-région-pays-All* est une hiérarchie minimale couvrante associée au Canada. Au Canada, nous n'avons pas d'état comme aux États-Unis. De plus, il n'existe pas de provinces-villes au Canada.

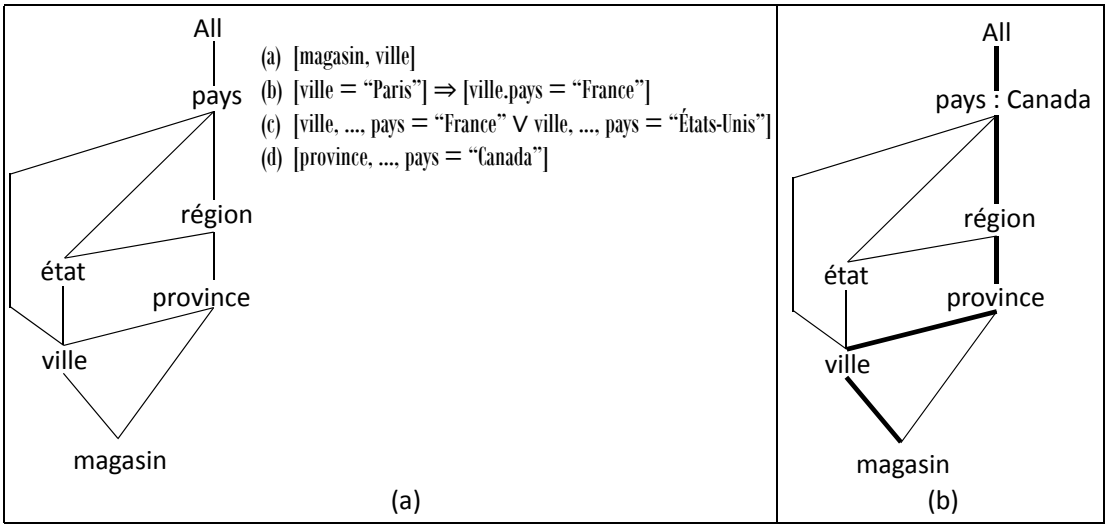



FIGURE 3.18 – Contraintes d'intégrité (a) et les dimensions minimales de Hurtado *et al.* (b) [HGM05]

La deuxième stratégie de normalisation consiste à ajouter de nouvelles structures au modèle logique dans le but de garantir l'additivité. À l'origine, cette technique est utilisée pour garantir l'additivité dans les associations faits-dimensions en stockant dans ces nouvelles structures (tables) les associations faits-dimensions non-strictes [SRME01, KR02]. De cette manière, les autres tables contiennent les données qui ne posent pas de problèmes d'additivité. Mazón *et al.* s'inspirent de ces travaux et présentent un modèle conceptuel pour les différentes associations (non-strictes et non-couvrantes) faits-dimensions afin de faciliter la représentation des situations du monde réel [MLT08]. Ils dérivent ensuite à partir de ce modèle un modèle conceptuel "normalisé" qui évite les problèmes d'additivité. La Figure 3.19(a) illustre un modèle conceptuel UML de Mazón *et al.* où les faits *ventes* sont exprimés selon les dimensions *date* et *client*.

Dans cet exemple, la relation entre *ventes* et *client* est non-couvrante puisque la cardinalité côté *client* est de type [0..1] (zéro à un). Afin de garantir l'additivité dans ce modèle, les auteurs séparent les ventes pour lesquelles nous ne connaissons pas de clients dans une nouvelle table de jointure *ventesSansClients* (Figure 3.19(b)) et l'association entre *ventes* et *client* est normalisée à [1..1] (un est un seul). Les faits *ventes* et *ventesSansClients* doivent avoir une structure similaire comme le montre la Figure 3.19(b). Par définition, une table de jointure est utilisée pour y séparer les relations faits-dimensions qui sont à l'origine des problèmes d'additivité.

Pour normaliser les associations faits-dimensions non-strictes, deux solutions sont proposées par Mazón *et al.* Dans la première solution, si nous connaissons la contribution de chaque instance de dimension pour chaque instance de fait, les valeurs des mesures sont partagées entre les différentes instances de dimensions en question. Dans ce cas, la multiplicité maximale côté dimension est transformée de $*$ (plusieurs) à 1 (un). Cette méthode ne nécessite pas la création d'une table de jointure, mais il suffit de regrouper les différentes instances de dimensions qui posent des problèmes d'additivité dans un nouvel attribut ajouté à la classe des faits. Dans la deuxième solution, si nous ne connaissons pas la contribution de quelques instances de dimension pour chaque instance de fait, les auteurs procèdent comme suit : (1) les instances connues seront traitées comme pour la première solution et (2) les instances inconnues sont séparées dans une nouvelle table de dimension (appelé aussi table séparée par Malinowski et Zimányi [MZ06]).

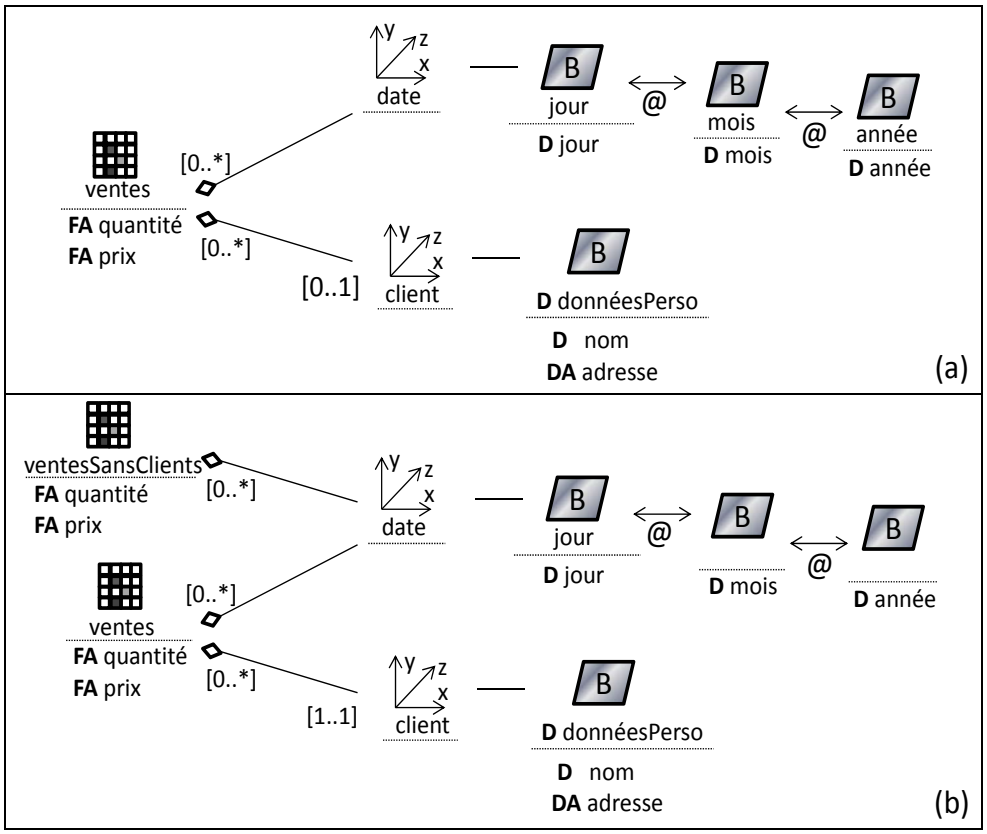



FIGURE 3.19 – Exemple de relation fait-dimension non-couvrante (a) et solution de Mazón *et al.* (b) [MLT08]

Dans un autre travail, Mansmann et Scholl présentent un outil OLAP visuel qui permet de normaliser, de parcourir et de visualiser les différents niveaux d'une hiérarchie [MS06, MS07]. Dans leur structure graphique, chaque niveau de la hiérarchie est modélisé par un répertoire. Explorer un niveau de la hiérarchie permet alors de visualiser la hiérarchie qui lui est associée (en commençant par le niveau supérieur). Une hiérarchie stricte est représentée par un dossier qui contient un seul sous-dossier. Les hiérarchies non-strictes sont représentées par des dossiers qui portent un nom composé de la fusion des différents niveaux hiérarchiques qui forment cette hiérarchie (le principe de fusion de Pedersen *et al.* [PJD99]). La normalisation des hiérarchies non-couvrantes n'est toutefois pas traitée dans ce travail.

### 3.3.5 Transformation de données multidimensionnelles

Les approches de transformation de données multidimensionnelles consistent à modifier les valeurs des instances de faits et de dimensions afin d'assurer l'additivité. Dans un premier travail, Pedersen *et al.* proposent une solution pour résoudre les problèmes d'additivité dans les hiérarchies de dimension complexes [PJD99]. Ils présentent deux algorithmes : (1) *MakeCovering* pour "remblayer" les valeurs manquantes dans les hiérarchies non-couvrantes et (2) *MakeStrict* pour "fusionner" les valeurs des instances multiples dans les hiérarchies non-strictes.

- *MakeCovering* : Soit la hiérarchie *adresse-ville-région-pays*. L'algorithme commence par insérer les valeurs manquantes à partir du niveau le plus détaillé de la hiérarchie (*adresse* dans cet exemple) jusqu'au niveau supérieur (*pays* dans cet exemple) en passant par les niveaux intermédiaires (*ville-région* dans cet exemple). *MakeCovering* cherche les valeurs manquantes dans les méta-données associées aux données multidimensionnelles et/ou fait appel à un expert. Par exemple, supposons que nous voulions compléter l'adresse d'une ruelle en Australie ou aux États-Unis, un avis d'expert peut être décisif dans ce cas en l'absence des métadonnées nécessaires.
- *MakeStrict* : Une fois les hiérarchies non-couvrantes transformées en hiérarchies couvrantes, l'algorithme *MakeStrict* est appelé afin de fusionner les valeurs des instances d'un même niveau hiérarchique ayant le même élément parent en une seule valeur. Les valeurs fusionnées sont ensuite insérées entre les deux niveaux hiérarchiques parent et enfant dans un objectif de calcul d'agrégations sans se soucier des problèmes d'additivité. Par exemple, soit la hiérarchie *livre-catégorie* de la Figure 3.20(a). Dans cet exemple, le livre "Da Vinci Code" est décrit dans deux catégories : "Thriller" et "Policier". Le livre "Hors de moi" est décrit dans une seule catégorie : "Thriller". Dans le cas du livre "Da Vinci Code", la hiérarchie *livre-catégorie* est non-stricte. *MakeStrict* créé alors un niveau intermédiaire qui fusionne les catégories "Policier" et "Thriller" en une seule, "Policier-Thriller", comme le montre la Figure 3.20(b)).
- Dans ce travail, un troisième algorithme (*MakeOnto*) est présenté pour transformer les hiérarchies non-onto en hiérarchies onto. Le principe de *MakeOnto* est le même que celui de *MakeCovering*. *MakeOnto* remplace les niveaux

manquantes par des niveaux artificiels (*placeholders*) au choix. *MakeOnto* est appelé après *MakeCovering* et avant *MakeStrict*.

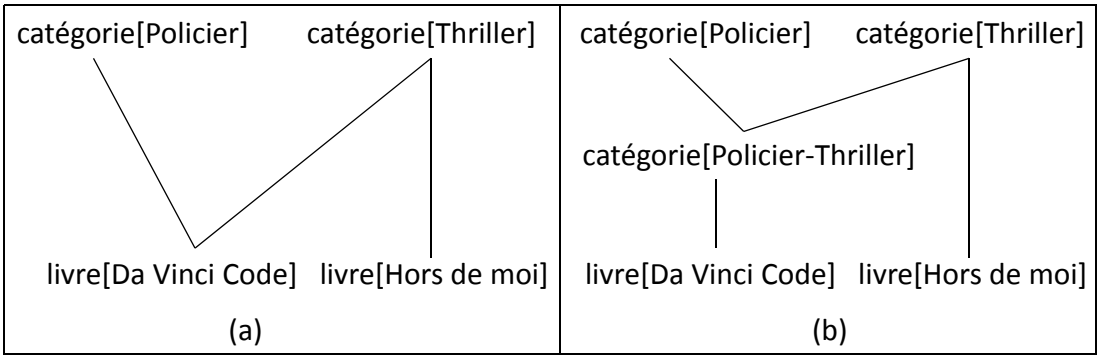



FIGURE 3.20 – Exemple de hiérarchie non-stricte (a) et solution de Pedersen *et al.* (b) [PJD99]

Pedersen *et al.* ne différencient pas les hiérarchies de dimension et les associations faits-dimensions. En fait, les auteurs utilisent de la même façon *MakeCovering* et *MakeStrict* pour résoudre les problèmes d'additivité dans les associations faits-dimensions [PJD99].

Dans un deuxième travail, Mansmann et Scholl affirment que la manipulation des données qui sont à l'origine des problèmes d'additivité dépend largement de leur sémantique [MS06, MS07]. Ils proposent donc de normaliser les instances, c'est-à-dire de transformer, les hiérarchies de dimension complexes tout en modifiant les techniques de transformation de données de Pedersen *et al.* [PJD99] comme suit.

1. Transformation des hiérarchies non-couvrantes : les auteurs regroupent toutes les instances d'un niveau hiérarchique donné pour lesquelles nous ne connaissons pas les valeur des instances de l'élément parent en dessous d'un niveau hiérarchique artificiel nommé "Others".
2. Transformation des hiérarchies non-strictes : en entrée de cette étape, nous trouvons la hiérarchie couvrante résultat de l'étape précédente. Pour normaliser une hiérarchie non-stricte, les auteurs construisent une nouvelle dimension entre les éléments parent et enfant de cette hiérarchie. Cette nouvelle dimension porte comme nom l'association des noms de tous les éléments parents d'une hiérarchie non-stricte.

3. Transformation des hiérarchies non-onto : est réalisée de la même façon que la transformation des hiérarchies non-couvrantes.

### 3.3.6 Solutions en temps réel

Dans ce qui suit, nous étudions un troisième type de solution pour les problèmes d'additivité : les solutions en temps réel [PRP02a, PPP04, HS05]. Le but de telles solutions est de détecter et de résoudre les problèmes d'additivité au moment du calcul des agrégations à un niveau hiérarchique donné à partir des valeurs agrégées à un niveau inférieur de la même hiérarchie.

L'opérateur *projection généralisée* de Pedersen *et al.* (Section 3.2.3) permet de calculer des agrégations [PRP02a, PPP04]. Au cas où il détecte des hiérarchies non-strictes ou non-couvrantes, cet opérateur retourne un résultat vide. Dans un deuxième travail, Horner et Song ouvrent des perspectives pour des nouvelles solutions en temps réel [HS05]. Pour résoudre les problèmes d'additivité dans les hiérarchies non-strictes, les auteurs proposent d'exécuter des scripts pour (1) déterminer combien de fois les mesures ont été agrégées dans ces hiérarchies et (2) identifier le nombre total des valeurs multiples. Ensuite, Horner et Song distinguent entre trois types de valeurs manquantes : *orphelines*, *omises* et *non-applicables*. Soit l'exemple de la hiérarchie *étudiant-département* de la Figure 3.21. Dans cet exemple, nous modélisons les trois types de valeurs manquantes définis par Horner et Song.

Une valeur est dite *manquante-orpheline* lorsque, en réalité, il existe un parent (une instance du niveau hiérarchique supérieur) pour cette valeur, mais qui n'est pas associé à cette valeur dans la base de données. Par exemple, dans la Figure 3.21, il est mentionné que l'étudiant E1 n'est inscrit dans aucun département. En réalité, l'étudiant E1 est bien inscrit dans le département "informatique" comme son collègue E2. Horner et Song modélisent les relations manquantes-orphelines avec des flèches discontinues (Figure 3.21). Pour résoudre ce problème, les auteurs proposent d'exécuter, d'une façon régulière, des scripts qui permettent d'identifier les valeurs manquantes-orphelines et leurs mesures.

Les valeurs *manquantes-omises* représentent les valeurs pour lesquelles les valeurs des parents n'ont pas été stockées dans la base de données. Par exemple, le département de l'étudiant E3 ne figure pas dans la base de données (Figure 3.21).

Pour résoudre ce problème, les auteurs proposent de : (1) déterminer les raisons pour lesquelles le concepteur de la base de données n'a pas enregistré ces valeurs dans la base de données et (2) calculer l'impact de cet oubli sur le résultat final.

Enfin, les valeurs *manquantes-non-applicables* représentent les valeurs qui ne sont associées à aucun élément parent en réalité. Par exemple, l'étudiant E4 de la Figure 3.21 n'appartient à aucun département, puisqu'il assiste aux cours sans être inscrit à l'université. Dans ce cas, les auteurs proposent de changer ces valeurs de "null" à "manquantes-non-applicables" dans la base de données.

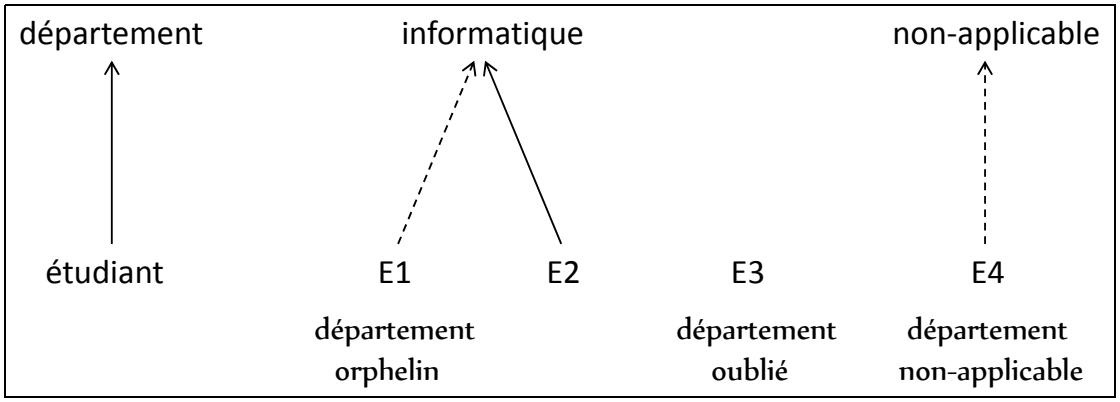



FIGURE 3.21 – Trois cas de hiérarchies non-couvrantes de Horner et Song [HS05]

### 3.3.7 Discussion

Dans ce qui suit, nous discutons les approches que nous avons présentées dans les Sections 3.3.3, 3.3.4, 3.3.5 et 3.3.6. Le Tableau 3.8 donne une comparaison de l'ensemble de ces approches selon les critères suivants :

- le modèle conceptuel utilisé dans l'approche : UML ou autre ;
- les hiérarchies traitées dans l'approche : non-couvrantes, non-strictes ou complexes ;
- la partie du modèle multidimensionnel considérée dans le modèle : associations fait-dimension, hiérarchies de dimension ou les deux ;
- l'implémentation (ou pas) de l'approche ;
- la nature de la solution présentée pour assurer l'additivité : *a priori* (transformation ou normalisation) ou en temps réel ;

- la complexité des algorithmes présentés.

| | Modélisation | Hiérarchies | H. dim. | Rel. F-D | Imp. | Nat. de sol. |
|---|---|---|---|---|---|---|
| [LMTS06] | UML | complexes | oui | oui | - | - |
| [ASS06] | UML | non-strictes | oui | oui | - | - |
| [MZ04, MZ06] | EA | complexes | oui | non | oui | directives |
| [PJD99] | Formalisme spécifique | complexes | oui | oui | oui | *a priori* |
| [MS06, MS07] | EA | complexes | oui | oui | oui | *a priori* |
| [HGM05] | Formalisme spécifique | non-couv. | oui | oui | oui | *a priori* |
| [MLT08] | UML | complexes | non | oui | oui | *a priori* |
| [PRP02a, PPP04] | XML | complexes | oui | non | oui | temps réel |
| [HS05] | Formalisme spécifique | complexes | oui | non | non | temps réel |

TABLE 3.8 – Comparaison des solutions pour les problèmes d'additivité

**Modèle de données et hiérarchies traitées**

La majorité des approches étudiées ont utilisé les techniques de modélisation traditionnelles pour modéliser, au niveau conceptuel, les situations les plus complexes du monde réel. Malinowski et Zimányi ainsi que Mansmann et Scholl étendent le modèle entité-association (EA) par de nouvelles notations spécifiques à la modélisation multidimensionnelle [MZ04, MZ06, MS06, MS07]. D'autres auteurs utilisent UML [ASS06, LMTS06, MLT08] ou un formalisme spécifique [PJD99, PRP02a, PPP04, HGM05, HS05]. Quelle que soit la nature du modèle conceptuel choisi, tous ces auteurs ont réussi à modéliser des situations complexes du monde réel. La majorité de ces modèles supportent les hiérarchies complexes, à l'excep-

tion des modèles de Hurtado *et al.* et d'Abelló *et al.*, comme le montre le Tableau 3.8.

Selon Mazón *et al.* [MLT09], dans un modèle multidimensionnel, le concepteur doit garantir :

1. la représentation adéquate des associations faits-dimensions [SRME01], et
2. la représentation adéquate des différents niveaux d'une dimension exprimée en hiérarchie [JLS99].

Or, certains auteurs ont choisi de prendre en compte une seule des deux types de hiérarchies : non-strictes ou non-couvrantes [PRP02a, PPP04, MZ04, MZ06, HGM05, HS05, MLT08].

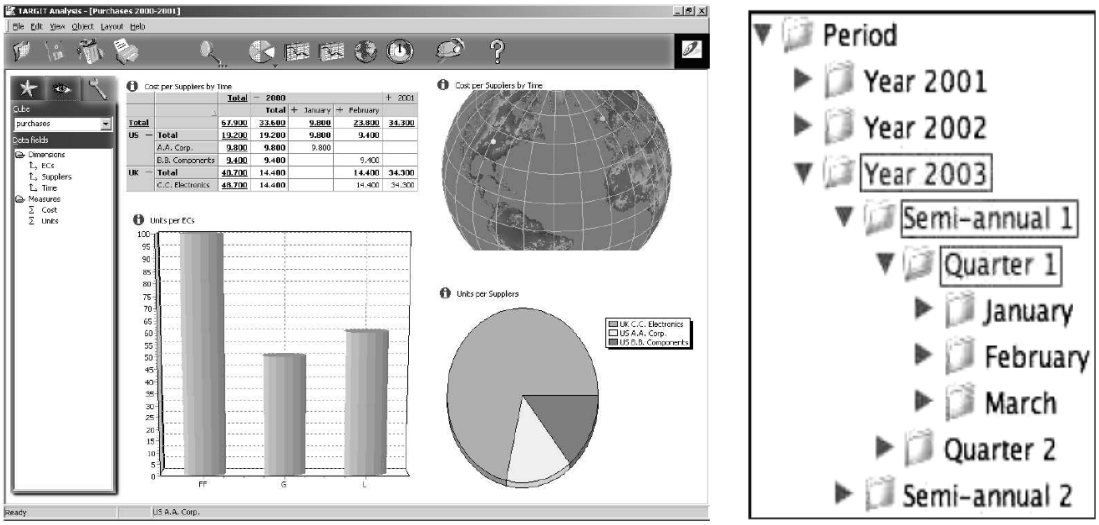
### Algorithmes et implémentations

Hurtado *et al.* affirment que le problème de la détection des problèmes d'additivité dans leur modèle est coNP-complet [HGM05]. Soient $D$ une dimension exprimée en hiérarchie, $c$ un élément de $D$ et $S$ un sous-ensemble de $D$. L'additivité est vérifiée pour $c$ de $S$ dans $D$ si et seulement si pour chaque élément inférieur $c_b$ de $D$, $c_b < c$. De plus, toutes les instances des éléments supérieurs à $c_b$ et inférieurs à $c$ doivent satisfaire l'additivité (pas d'éléments intermédiaires manquantes). En suivant ce raisonnement, les auteurs démontrent que le temps du test d'additivité est polynomial.

Pedersen *et al.* affirment que leurs algorithmes se caractérisent par une faible complexité de calcul et peuvent être appliqués sur de larges volumes de données [PJD99]. Toutes les étapes de *MakeCovering* sont exprimées à l'aide des opérateurs standards de l'algèbre relationnelle. La complexité de la jointure la plus élevée est de l'ordre de $O(n^2)$, $n$ étant la taille des données en entrée. Comme *MakeCovering* traite des hiérarchies, la complexité de la jointure est plutôt de l'ordre de $O(n\ log\ n)$ en pratique. Soit $k$ le nombre d'éléments d'une hiérarchie, la boucle interne de *MakeCovering* est évaluée au plus une fois pour chaque couple des $k$ éléments, c'est-à-dire $k^2/2$ fois. Dans ce cas, la complexité maximale de l'algorithme est de l'ordre de $O(k^2 n\ log\ n)$. Comme une hiérarchie non-onto est une hiérarchie non-couvrante, la complexité maximale de l'algorithme *MakeOnto* est également de l'ordre de $O(k^2 n\ log\ n)$.

Si nous suivons le même raisonnement utilisé pour le calcul de la complexité de *MakeCovering*, la complexité maximale de l'algorithme *MakeStrict* est de l'ordre de $O(pnk\ log\ n\ log\ k)$. Le paramètre $p$ désigne le nombre de parents immédiats et ses éléments enfants dans la hiérarchie de dimension. Le paramètre $k$ désigne le nombre maximum de valeurs "fusionnées". Dans la majorité des cas, $p$ et $k$ ont des valeurs faibles. Dans ce cas, la complexité de *MakeStrict* est plutôt de l'ordre $O(n\ log\ n)$.

Comme le montre le Tableau 3.8, la majorité des auteurs présentent une implémentation de leurs approches pour la détection et la résolution des problèmes d'additivité. Mansmann et Scholl présentent un outil OLAP visuel exceptionnel qui permet de modéliser et de transformer les hiérarchies complexes [MS06, MS07]. L'opérateur *projection généralisée* de Pedersen *et al.* est implémenté au sein de l'outil OLAP TARGIT [PRP02a, PPP04]. TARGIT est différent de l'outil de Mansmann et Scholl [MS06, MS07] puisqu'il permet de visualiser les résultats des analyses. La Figure 3.22 fournit deux captures d'écran de ces outils. Pedersen *et al.*, Hurtado *et al.* et Mazón *et al.* affirment qu'ils ont implémenté leurs approche, mais ne donnent pas plus de détails sur ce sujet [PJD99, HGM05, MLT08].




(a) (b)

FIGURE 3.22 – Exemples de (a) l'outil OLAP TARGIT [PRP02a, PPP04] et de (b) l'outil OLAP visuel de Mansmann et Scholl [MS06, MS07]

### Transformation ou normalisation

Toutes les approches de transformation [PJD99, MS06, MS07] se basent sur les deux algorithmes *MakeStrict* et *MakeCovering*. En fait, les hiérarchies non-strictes existent dans plusieurs situations du monde réel [PJD99] et leur transformation à l'aide du processus de fusion nécessite le respect de la sémantique des données exprimées dans ces hiérarchies [MS06, MS07]. De ce fait, nous pensons que l'algorithme *MakeStrict* respecte la sémantique des données réelles dans son fonctionnement. Dans l'exemple de la Figure 3.20, la fusion des deux catégories "Policier" et "Thriller" dans une seule catégorie "Policier-Thriller" ne contredit pas la sémantique de la description du livre "Da Vinci Code".

Li *et al.* démontrent toutefois que l'algorithme *MakeCovering* n'est pas adapté à certains cas réels [LSZY05]. Soit l'exemple de la *dimension géographique* liée à la Chine : *district-ville-province-pays*. Certaines provinces contiennent un ou plusieurs districts, mais ces districts n'appartiennent à aucune ville. En d'autres termes, certaines provinces sont elles même des villes, comme Pékin [8].

*MakeCovering* ne permet pas de résoudre ce problème puisqu'il remplace pour de tels cas les valeurs manquantes par des vraies valeurs, mais ne présente aucune solution spéciale. Li *et al.* affirment que le problème devient de plus en plus complexe en cas de la hiérarchie *entreprise-localité géographique*, puisque les entreprises sont organisées géographiquement d'une façon particulière en Chine [LSZY05]. Pour cela, Li *et al.* distinguent quatre types de hiérarchies non-couvrantes (A, B, C et D) et proposent plusieurs versions de *MakeCovering* afin de pouvoir traiter tous les cas possibles liés à la hiérarchie géographique en Chine. L'inconvénient de ce travail est qu'il traite un cas particulier, mais nous pensons que le même raisonnement est utile pour d'autres pays comme Monaco.

Même si l'approche de transformation de Pedersen *et al.* [PJD99] a été la première solution de la littérature pour résoudre les problèmes d'additivité dans les hiérarchies complexes, elle souffre de quelques problèmes de performance. En effet, à chaque fois que l'entrepôt est mis à jour, l'additivité doit être vérifiée. Pour cela, le choix de la transformation comme solution pour les problèmes d'additivité a été abandonné par les auteurs des approches de normalisation. Deux directions ont été suivies dans les approches de normalisation. La première technique se

8. `http://fr.wikipedia.org/wiki/Pekin`

résume dans les contraintes et les hiérarchies de dimension non-couvrantes minimales (*frozen dimensions*) de Hurtado *et al.* [HGM05]. Nous remarquons que les hiérarchies de dimension non-strictes n'ont pas été traitées dans ce travail. Nous pensons d'ailleurs que les dimensions non-couvrantes minimales peuvent être associées à l'algorithme *MakeCovering* de Pedersen *et al.* pour résoudre les problèmes présentés dans l'article de Li *et al.* [LSZY05].

Au lieu de définir plusieurs extensions de *MakeCovering*, nous pensons qu'il serait possible de déduire toutes les hiérarchies non-couvrantes minimales correspondantes à *district-ville-province-pays* en entrée de l'algorithme *MakeCovering*. Par exemple, nous pouvons associer la hiérarchie non-couvrante minimale *district-province-pays* à toutes les provinces-villes (comme Pékin). Dans ce cas, *MakeCovering* pourra traiter les hiérarchies non-couvrantes minimales sans qu'il soit nécessaire de le modifier. Enfin, cette proposition (théorique) nécessite l'intervention d'un expert ce qui ne contredit pas le principe de *MakeCovering*.

La deuxième technique de normalisation consiste à ajouter de nouvelles structures au modèle multidimensionnel : les tables de jointures (*bridge tables*). Ces techniques sont généralement utilisées au niveau logique [SRME01, KR02, MZ06]. Seuls Mazón *et al.* emploient les tables de jointure au niveau conceptuel. Leur motivation derrière ce choix est que l'utilisation de ces techniques au niveau logique nécessite une grande expertise pour modéliser les situations du monde réel les plus complexes.

Selon Malinowski et Zimányi, les avantages des tables de jointure se résument dans leur très faible taille et leur utilité lorsque les informations reliées aux mesures ne changent pas dans le temps [MZ06]. Par contre, côté performance, des efforts supplémentaires sont nécessaires pour agréger toutes les mesures, distribuées entre les faits originaux et les tables de jointure. De plus, ces tableaux sont plus utiles dans le cas des hiérarchies non-couvrantes et sont peu employés pour la normalisation des hiérarchies non-strictes. En pratique, seuls Mazón *et al.* implémentent leur approche basée sur les tables de jointure [MLT08].

### Vers des solutions en temps réel

Comme le montre le Tableau 3.8, la majorité des solutions proposées dans la littérature s'appliquent *a priori*. Ces solutions regroupent les approches de nor-

malisation (Section 3.3.4) et de transformation (Section 3.3.5). Même si elles sont différentes, la transformation et la normalisation partagent le même objectif : s'assurer de ne pas avoir des problèmes d'additivité au moment de l'exécution des requêtes OLAP qui prennent en compte les hiérarchies, comme les *opérateurs de forage* par exemple.

Nous avons présenté dans la Section 3.3.6 les seuls travaux qui proposent des solutions en temps réel pour les problèmes d'additivité. L'opérateur *projection généralisée* [PRP02a, PPP04], proposé dans le cadre d'une approche XOLAP, permet de signaler la présence des hiérarchies non-strictes ou non-couvrantes en retournant un résultat vide. Nous pensons que le fait de retourner un résultat vide ne représente pas une solution pour les problèmes d'additivité à la hauteur des solutions de transformation ou de normalisation. Les directives de Horner et Song [HS05] présentent plus de choix en terme de solutions pour les problèmes d'additivité, pour hiérarchies non-strictes et non-couvrantes (*orphelines*, *oubliées* et *non-applicables*), que l'opérateur *projection généralisée* de Pedersen *et al.* Toutefois, Horner et Song ne présentent pas d'algorithme ni de programme pour implémenter ces directives.

Si nous associons les travaux de Pedersen *et al.* [PRP02a, PPP04] aux travaux de Beyer *et al.* [BCC+05], nous concluons que XML est un langage très adapté à la modélisation des hiérarchies complexes. Pedersen *et al.* et Beyer *et al.* ne trouvent aucune difficulté pour modéliser les hiérarchies complexes en XML. Si nous associons aux avantages de la modélisation XML les directives de Horner et Song, nous pouvons déduire que de nouvelles solutions en temps réel -plus riches de ce qui a été présenté- peuvent être proposées pour résoudre les problèmes d'additivité dans les hiérarchies complexes modélisées en XML. D'un côté, modéliser les hiérarchies complexes en XML permet de préserver la sémantique des situations du monde réel (contrairement à la normalisation). D'un autre côté, et comme l'indiquent Beyer *et al.* dans les perspectives de leur travail, il serait intéressant d'étendre les requêtes XML avec des fonctions de gestion des hiérarchies complexes.

### Synthèse générale

Dans ce qui suit, nous récapitulons les limitations des différentes familles de solutions pour les problèmes d'additivité : modélisation, normalisation, transfor-

mation et solutions en temps réels.

- ⊖ Besoin d'un expert de la modélisation multidimensionnelle et de la technique des tables de jointure (modélisation, normalisation).
- ⊖ Modification de la sémantique des données réelles (modélisation, normalisation, transformation).
- ⊖ Besoin de métadonnées et/ou d'un expert du domaine et des données entreposées (transformation).
- ⊖ Non prise en compte de toutes les situations complexes du monde réel (transformation) [LSZY05].
- ⊖ Nécessité de mettre à jour toutes les données transformées, à chaque fois que l'entrepôt est mis à jour (transformation).
- ⊖ Détection des problèmes d'additivité, mais pas de traitement (solutions en temps réels).

## 3.4 Modèles d'arbres XML

Dans cette section, nous étudions les modèles d'arbres XML [HD12]. Au début, nous donnes les définitions nécessaires reliées aux modèles d'arbres XML (arbres de données, sous-arbres de données, modèles d'arbres...). Après, nous présentons le processus de *matching* qui consiste à apparier un modèle d'arbre avec un arbre de données. Nous verrons aussi comment les modèles d'arbres étaient cibles de plusieurs algorithmes de minimisation dans un but d'optimisation du *matching*. Après, nous présentons les approches holistiques qui ont également présenter une optimisation du processus de *matching* en limitant l'accès à la mémoire. Enfin, nous verrons comment les modèles d'arbres XML ont été utilisés dans des domaines variés comme la fouille de données.

### 3.4.1 Définitions et modèle d'arbre de base

Dans cette section, nous commençons par formaliser les concepts liés aux modèles d'arbres. Nous donnons aussi un exemple très simple de modèle d'arbre pour illustrer, de manière schématique, leur fonctionnement.

### Document XML

XML est connu comme un langage simple et très flexible de représentation et d'échange de données textuelles. Le contenu d'un document XML est encapsulé dans des éléments définis dans des balises, appelées *tags* en Anglais. Une balise a un nom défini par l'utilisateur et elle est représentée entre deux chevrons < > (exemple : <nom>). Les éléments d'un document XML sont généralement vus comme une hiérarchie organisée de manière arborescente.

### Fragment XML

Un document XML est un ensemble ordonné d'éléments. Tout élément XML peut contenir des sous-éléments. Tout sous-élément peut également contenir aussi des sous-éléments, et ainsi de suite. La racine d'un document XML est un élément particulier puisqu'il contient tous les éléments du document XML. Chaque élément XML (en plus de son contenu) différent de la racine est appelé un fragment XML. Tout fragment XML peut être représenté sous la forme d'un arbre enraciné, étiqueté et ordonné.

### Arbre de données

Un document XML peut être représenté sous la forme d'un arbre de données $t = (r, N, E)$, où $N$ est l'ensemble des nœuds de $t$, $r \in N$ est l'élément racine de $t$ et $E$ est l'ensemble d'arêtes reliant les paires de nœuds $(\mathrm{n}_i, \mathrm{n}_j) \in N \times N$ ($i$ et $j$ sont des entiers).

### Collection d'arbres de données

Un document XML est un ensemble de fragments XML. Il peut être représenté par une collection d'arbres de données $C$. Chaque arbre de $C$ modélise un fragment XML distinct. Une collection d'arbres de données est elle-même un arbre de données. Une collection d'arbres de données est appelée *forêt* dans l'algèbre XML TAX [JLST01].

### Sous-arbre de données

Soit un arbre de données $t = (r, N, E)$. $t^{'} = (r^{'}, N^{'}, E^{'})$ est un sous-arbre de $t$ si et seulement si :

1. $N^{'} \subseteq N$ ;
2. soit $e^{'} \in E^{'}$ une arête reliant deux nœuds $\mathrm{n}_i^{'}$ et $\mathrm{n}_j^{'}$ de $t^{'}$ ; il existe une arête $e \in E$ reliant deux nœuds $\mathrm{n}_i$ et $\mathrm{n}_j$ de $t$ tel que $\mathrm{n}_i = \mathrm{n}_i^{'}$ et $\mathrm{n}_j = \mathrm{n}_j^{'}$.

### Modèle d'arbre

Dans la littérature, un modèle d'arbre (*tree pattern*) est le fruit de la modélisation d'une requête utilisateur (ou d'une expression de chemin XPath) dans un format arborescent [BT99, JLST01, FFM03]. Théoriquement, un modèle d'arbre ne peut représenter qu'une seule requête XPath. La transformation d'une requête XML en un modèle d'arbre n'est pas une opération simple. En fait, certaines requêtes XQuery sont écrites dans des combinaisons complexes d'expressions XPath et FLWOR, ce qui nécessite plus d'un modèle d'arbre pour les représenter. Plus la requête XML est complexe, plus sa traduction en un (ou plusieurs) modèle d'arbre est difficile [MMS07]. Certaines approches ont donc choisi d'exprimer les requêtes utilisateur à l'aide des modèles d'arbres en premier lieu. Ensuite, ils modélisent ces modèles d'arbre dans un langage de requêtes XML [BT99].

Formellement, un modèle d'arbre $p = (t, F)$ est défini par :

1. $t = (r, N, E)$ est un arbre de données ;
2. $F$ est une formule qui spécifie des contraintes sur les nœuds à sélectionner en résultat. Les contraintes sont reliées d'une façon booléenne à l'aide des signes & (pour dire *et en même temps*). Dans le reste de ce mémoire, nous étiquetons les nœuds d'un modèle d'arbre par $\$i$ (avec $i$ un entier quelconque supérieur à 0). Par convention, nous commençons toujours par \$1 pour désigner le nœud racine.

### *Matching* des modèles d'arbre

Le *matching* (appariement) entre un modèle d'arbre $p$ et un arbre de données $t$ est défini par une fonction $f : p \rightarrow t$ qui permet d'apparier les nœuds de $p$ avec leur équivalents dans $t$, sachant que :

- les relations structurelles entre les différentes nœuds de $p$ doivent être respectées dans $t$. Nous distinguons deux types de relations structurelles : les relations *parent-enfant* et les relations *ancêtre-descendant*. Une relation parent-enfant (PE) est représentée par une arête simple (/) dans XPath. Une relation ancêtre-descendant (AD) est représentée par une arête double (//). Si deux nœuds quelconques de $p$ sont reliés par une relation PE ou AD, leurs équivalents dans $t$ doivent être reliés par une relation du même type.
- toutes les contraintes de la formule $F$ doivent être vérifiées dans $t$.

Dans le reste de ce mémoire, nous utilisons le terme *matching* sans le traduire.

**Exemple de modèle d'arbre**

Dans l'arbre de données de la Figure 3.23(a), nous modélisons des *ventes* de livres caractérisées par leur *titre*, leur(s) *auteur(s)*, le lieu de vente (*pays*), la date de vente (*année*), le *prix* et l'*éditeur* du livre. Les sous-arbres *vente* n'ont pas forcément la même structure. Par exemple, nous connaissons l'*année* d'édition du premier livre, "Verte", mais nous ne la connaissons pas pour le second livre, "La vie sauve". En revanche, nous connaissons l'*éditeur* de "La vie sauve" mais pas celui de "Verte".

Un exemple de modèle d'arbre est présenté dans la Figure 3.23(b). Ce modèle d'arbre sélectionne les ventes de livres par titre, auteur et année en respectant les contraintes de la formule $F$, située à droite du modèle. Les relations entre les nœuds de ce modèle d'arbre sont de type PE (/) ou AD (//). Ici, les relations AD signifient que notre modèle d'arbre sélectionne tous les livres, peu importe le niveau dans lequel sont modélisés les auteurs dans l'arbre de données (cas des livres écrits par plusieurs auteurs). La contrainte `$3.value != "L. Violet"` de $F$ indique que l'auteur (ou un des auteurs) des livres à sélectionner doit être différent de "L. Violet". Les termes *tag* et *value* de F désignent, respectivement, le nom et la valeur d'un nœud.

L'arbre résultat du *matching* (ou *witness tree* dans TAX [JLST01]) entre ce modèle d'arbre et l'arbre de données de la Figure 3.23(a) est illustré dans la Figure 3.23(c). Nous remarquons que "La vie sauve" n'est pas été sélectionné dans le résultat pour les raisons suivantes : (1) on ne connait pas l'année de son édition

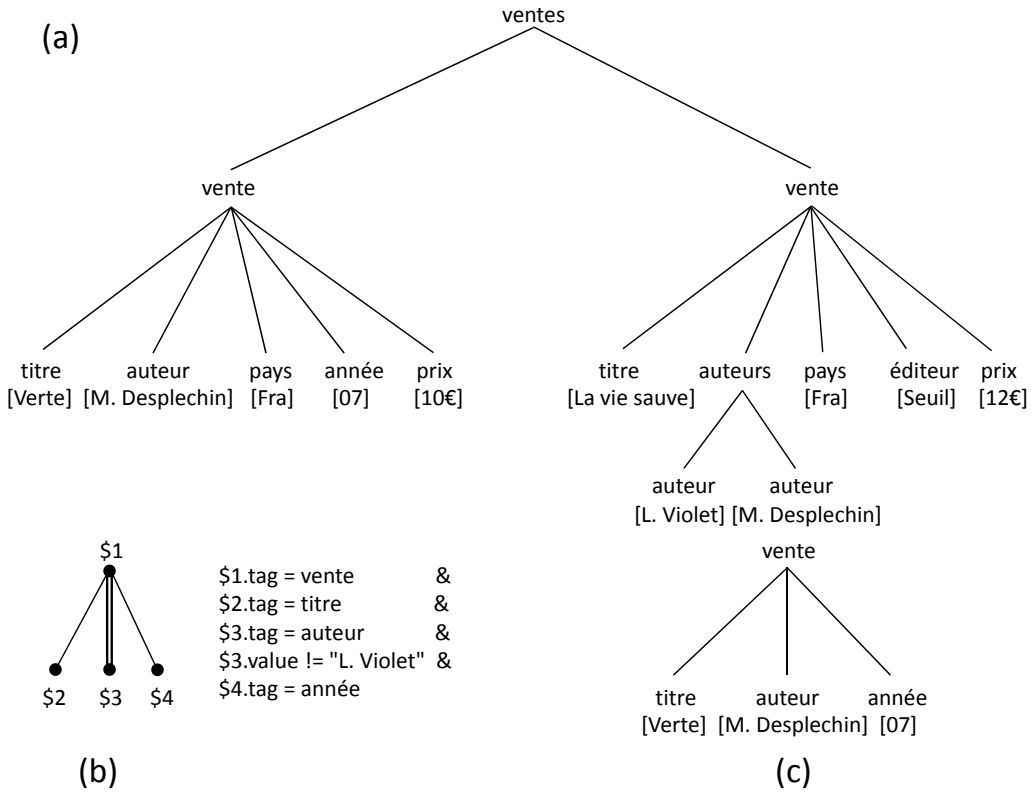



FIGURE 3.23 – Arbre de données (a), modèle d'arbre (b) et arbre résultat (c)

et (2) un de ses auteurs est "L. Violet", ce qui est contradictoire avec la formule $F$. Il est important de mentionner que la relation AD "vente//auteur" du modèle d'arbre (Figure 3.23(b)) a été correctement prise en compte dans le résultat. Toutefois, "La vie sauve" n'était pas sélectionné en résultat car un de ces auteurs est "L. Violet". Supposons maintenant que nous modifiions cette contrainte à `$3.value != "P. Coelho"` et que, en même temps, nous éliminons la contrainte `$4.tag = année` de $F$. Alors, "La vie sauve" sera sélectionné en résultat du *matching*.

### 3.4.2 Modèles d'arbres étendus

Le modèle d'arbre de la Figure 3.23(b) représente le modèle d'arbre le plus simple que nous pouvons trouver dans la littérature. Il correspond d'ailleurs au modèle d'arbre utilisé dans TAX [JLST01]. Depuis, plusieurs auteurs ont cherché à

étendre le processus de *matching* en enrichissant les nœuds et les arêtes du modèle d'arbre TAX avec des options de *matching* (des annotations, par exemple).

### Modèle d'arbre généralisé

Un modèle d'arbre généralisé (*generalized tree pattern*) associe des options aux arêtes du modèle d'arbre TAX (Figure 3.23(b)) afin d'améliorer le processus de *matching* en offrant la possibilité de sélectionner plus d'arbres en résultat que le modèle d'arbre TAX [CJLP03]. Chen *et al.* associent les mentions *obligatoire* ou *optionnel* aux arêtes du modèle d'arbre TAX. L'idée derrière ce nouveau modèle d'arbre.

La Figure 3.24(a) représente un modèle d'arbre généralisé qui sélectionne les *ventes* des livres décrits par leurs *titres* et leurs *années* d'édition.

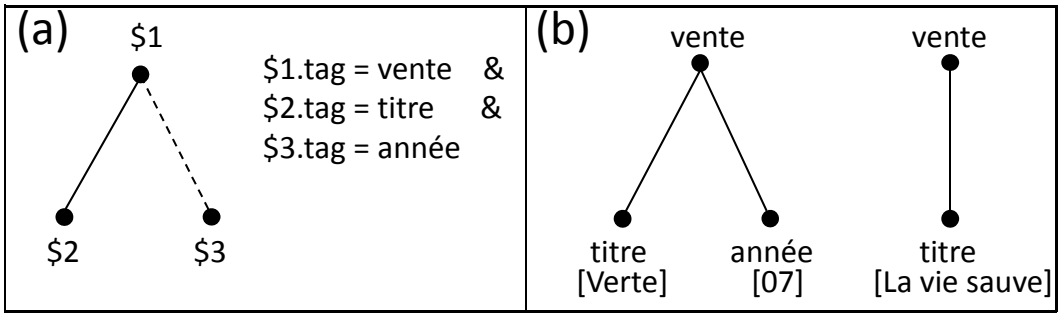



FIGURE 3.24 – Modèle d'arbre généralisé (a) et arbres résultats (b)

Dans ce modèle, les arêtes continues représentent les relations obligatoires dans le résultat. Le *matching* doit retourner les homologues de tous les couples de nœuds reliés par des relations obligatoires, sinon il ne retourne rien (exemple de la relation vente/titre de la Figure 3.24(a)). Les arêtes discontinues dans le modèle d'arbre de la Figure 3.24 représentent les relations optionnelles dans le résultat. Deux nœuds reliés par une relation optionnelle dans un modèle d'arbre généralisé signifient que le *matching* retourne les homologues de ces deux nœuds dans l'arbre de données s'ils existent. Sinon, seules les relations obligatoires apparaissent dans l'arbre résultat.

Le modèle d'arbre de la Figure 3.24(a) sélectionne donc toutes les *ventes* de livres (de la Figure 3.23(a)) décrits par leurs *titres* et leurs *années* d'édition. Au

cas où le nœud *année* est inconnu pour certains livres ("La vie sauve" dans cet exemple), ce livre est toue de même sélectionné dans le résultat comme l'illustre la Figure 3.24(b). L'avantage du modèle d'arbre généralisé par rapport à celui de TAX est qu'il permet de sélectionner, en résultat, plus d'arbres résultats même s'ils ne contiennent pas les nœuds marqués comme optionnels du modèle d'arbre.

### Modèle d'arbre annoté

Soient $p$ un modèle d'arbre TAX et $x$ un nœud de $p$. Le *matching* entre $p$ et un arbre de données quelconque retourne tous les homologues de $x$, s'ils existent. Par exemple, le modèle d'arbre de la Figure 3.23(b) sélectionne, pour chaque *vente*, le (ou les) *auteurs* du livre en question.

Afin de maîtriser le nombre d'homologues de nœuds à sélectionner dans le résultat, Paparizos *et al.* [PWLJ04] proposent d'annoter les arêtes du modèle d'arbre TAX comme suit :

- ? : zéro à un *matching* est autorisé ;
- $*$ : zéro à plusieurs *matchings* sont autorisés ;
- - : un et un seul *matching* est autorisé ;
- $+$ : un à plusieurs *matchings* sont autorisés.

La Figure 3.25(a) illustre un modèle d'arbre annoté. Dans ce modèle d'arbre, l'arête annoté avec un "-" signifie que ce modèle ne sélectionne que les livres écrits par un et un seul auteur. Le résultat du *matching* entre la Figure 3.25(a) et l'arbre de données de la Figure 3.23(a) est illustré dans la Figure 3.25(b).

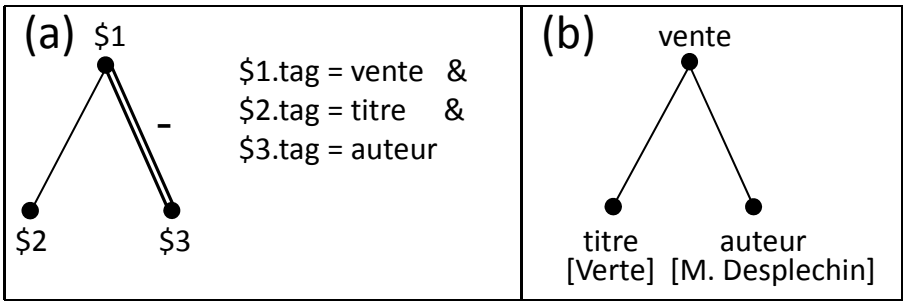



FIGURE 3.25 – Modèle d'arbre annoté (a) et arbre résultat (b)

Seul “Verte” est sélectionné puisqu'il est écrit par un seul auteur. Enfin, le modèle d'arbre annoté constitue le moteur des opérateurs de l'algèbre XML TLC (*Tree Logical Classes*) [PWLJ04], l'extension de TAX par ces annotations.

### Modèle d'arbre annoté ordonné

Le *matching* entre un modèle d'arbre TAX et un arbre de données quelconque préserve l'ordre des nœuds de l'arbre de données dans l'arbre résultat, peu importe leur ordre dans le modèle d'arbre. C'est pourquoi Paparizos et Jagadish enrichissent les opérateurs TLC *sélection* et *jointure* par une spécification d'ordre (*O-Spec*), et proposent par la suite le modèle d'arbre annoté ordonné [PJ05]. Dans ce travail, quatre spécifications d'ordre sont proposées :

- aucun ordre n'est spécifié ;
- l'ordre des nœuds dans l'arbre de données est maintenu dans *O-Spec* ;
- les nœuds dans l'arbre résultat sont entièrement triés suivant l'ordre indiqué dans *O-Spec* ;
- les nœuds dans l'arbre résultat sont partiellement triés suivant l'ordre indiqué dans *O-Spec*.

Soit le modèle d'arbre annoté ordonné de la Figure 3.26(a). Nous associons à ce modèle la spécification d'ordre suivante : `O-Spec = [auteur, titre]`.

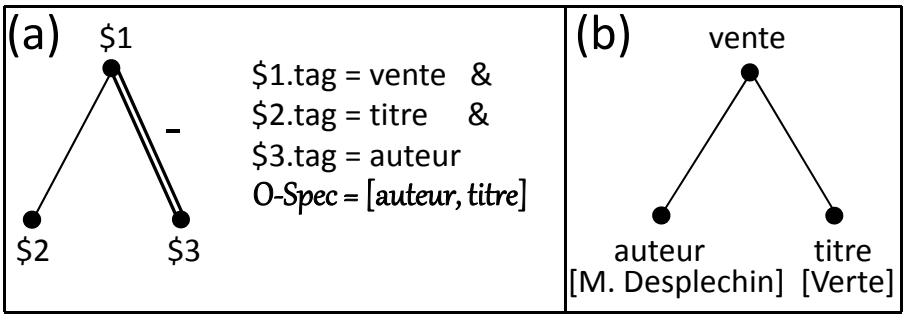



FIGURE 3.26 – Modèle d'arbre annoté ordonné (a) et arbre résultat (b)

Le *matching* de ce modèle d'arbre annoté ordonné avec l'arbre de données de la Figure 3.23(a) donne en résultat l'arbre de données de la Figure 3.26(d). Dans

ce résultat, nous remarquons que l'ordre spécifié dans *O-Spec* est respecté, puisque le nœud titre est situé à droite du nœud *auteur*.

### Modèle d'arbre global

Par définition, un modèle d'arbre global $g$ est conçu à partir de tous les sous-modèles d'arbres ordonnés possibles pour la même requête [Che04].

Nous résumons les étapes de construction d'un modèle d'arbre global $g$, à partir de $i$ sous-modèles d'arbres $p$ ; dans ce qui suit :

1. au début, la racine de $g$ est créée ;
2. après, chaque modèle d'arbre $p_i$ est fusionné dans $g$ comme suit :
   2-1. la racine de $p_i$ est fusionnée avec la racine de $g$ ;
   2-2. les nœuds de $p_i$ sont fusionnés avec leurs homologues dans $g$ tout en respectant leur ordre et les relations PE et AD.

Les deux modèles d'arbre $p_1$ et $p_2$, de la Figure 3.27, sont à l'origine du modèle d'arbre global $g$ qui sélectionne les *ventes* des livres décrits par leurs *titres* et leurs *années* d'édition.

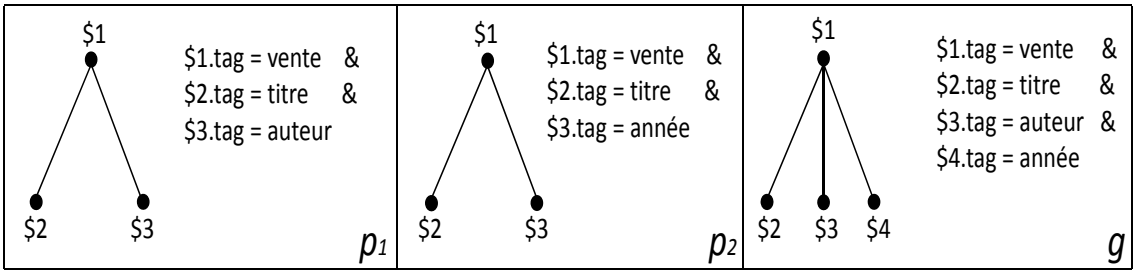



FIGURE 3.27 – Modèle d'arbre global

### Modèle de graphe

Kimelfeld et Sagiv introduisent les modèles de graphes, qui sont tout simplement des modèles d'arbres pour graphes directs et enracinés [KS06]. Il est important de mentionner que, dans cet article, les auteurs utilisent le terme Anglais *twig*

*pattern* pour désigner les modèles de graphes. Or, *twig pattern* a aussi été utilisé dans la littérature pour désigner les modèles d'arbres [LCL04].

Dans ce contexte, les documents XML sont modélisés en graphes, et non en arbres. Les arêtes des graphes sont représentées avec des flèches partant du nœud parent vers le nœud enfant. Toutefois, les nœuds de deux sous-graphes différents sont reliés avec des flèches discontinues (appelés références).

Dans le processus de *matching*, comme dans les modèles d'arbre, les modèles de graphes respectent les relations PE/AD. Les contraintes liées aux de nœuds sont en revanche spécifiées directement dans le schéma du modèle de graphe, comme le montre la Figure 3.28. Ce modèle de graphe est équivalent au modèle d'arbre de la Figure 3.23(b).

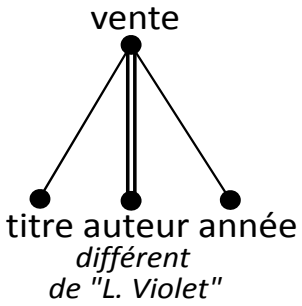



FIGURE 3.28 – Modèle de graphe

Kimelfeld et Sagiv proposent une extension du modèle de graphe : le modèle de graphe à distance limitée (*distance-bounding twig* ou *DBTwig*) [KS06]. Un modèle de graphe à distance limitée limite le grand nombre de graphes en sortie du *matching* entre un modèle de graphe et un graphe XML.

Pour atteindre cela, les auteurs utilisent la technique de *filtrage+classement* (*filtering+ranking*). Le principe de cette technique est de filtrer les graphes résultats en indiquant la longueur des chemins correspondants aux nœuds descendants. Par exemple, considérons un graphe $G$ ayant la même structure que l'arbre de données de la Figure 3.23(a). Considérons ensuite un modèle de graphe à distance limitée $g$ ayant la même structure que le modèle d'arbre de la Figure 3.23(b). Si nous associons la distance limitée *auteur = 1* à $g$ pour spécifier que ce modèle de graphe ne sélectionne que les livres écrits par un seul auteur, nous n'obtenons en résultat que le livre "Verte".

### Nœuds-opérateurs logiques

Izadi *et al.* enrichissent la notion de modèle d'arbre en proposant les nœuds-opérateurs logiques [IHH09]. Trois familles d'opérateurs sont proposés :

1. les *opérateurs binaires* : $\wedge$ (et), $\vee$ (ou) et $\oplus$ ;
2. les *opérateurs logiques* : AND, OR et XOR ;
3. l'*opérateur unaire* NOT ($\neg$).

La différence entre les opérateurs d'Izadi *et al.* et les annotations de Paparizos *et al.* est que ces dernières sont associés aux arêtes (relations entre nœuds) et pas aux nœuds. L'exemple de la Figure 3.29 sélectionne les ventes des livres identifiés par leur titres et un des deux éléments : année *ou* éditeur. Ce modèle sélectionne alors les deux ventes puisque nous connaissons l'année d'édition de "Verte" et l'éditeur de "La vie sauve".

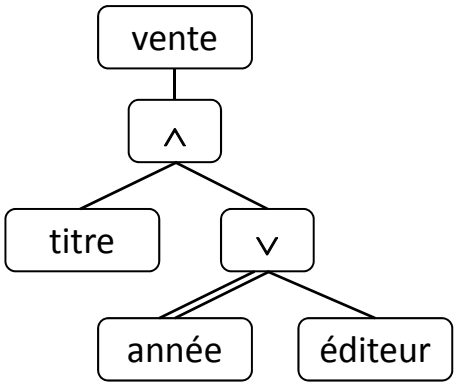



FIGURE 3.29 – Nœuds-opérateurs logiques

### Nœud de sortie et degrés de nœuds

Miklau et Sucio modélisent les expressions de chemins XPath en modèles d'arbres [MS02]. Dans ces modèles d'arbres, le nœud de sortie est étiqueté avec le symbole $*$. Supposons que, dans le résultat du *matching* entre le modèle d'arbre et l'arbre de données des Figures 3.23(a) et 3.23(b), nous ne voulons afficher que les *titres* de livres. Dans ce cas, il suffit de marquer le nœud *titre* du modèle d'arbre en question avec le symbole $*$. De plus, les auteurs définissent le degré d'un nœud $x$ (noté $degree(x)$) comme le nombre de ses nœuds enfants [MS02].

### Formule étendue

Lakshmanan *et al.* [LRWZ04] enrichissent la définition formelle de la formule $F$ par les contraintes suivantes :

- contraintes sur les noms des nœuds (*tag constraints*) : de la forme *nœud.tag = "titre"* ;
- contraintes sur les attributs et valeurs des nœuds (*value-based constraints*) : spécifient des conditions de sélection et de jointure à l'aide des opérateurs $\neq$, $\leq$, $\geq$, $<$ et $>$ ;
- contraintes sur les identificateurs de nœuds (*node identity constraints*) : précisent si deux nœuds sont équivalents ($\approx$) ou pas.

Dans l'exemple de la Figure 3.30, nous étendons le modèle d'arbre de la Figure 3.23(b) avec la contrainte suivante : *titre* (*$2.type = "roman"*), pour dire que le livre à sélectionner doit être un roman. Si nous supposons que, dans la Figure 3.23(a), "Verte" est de type "nouvelles" et "La vie sauve" est de type "roman", seul ce dernier livre sera sélectionné en résultat.

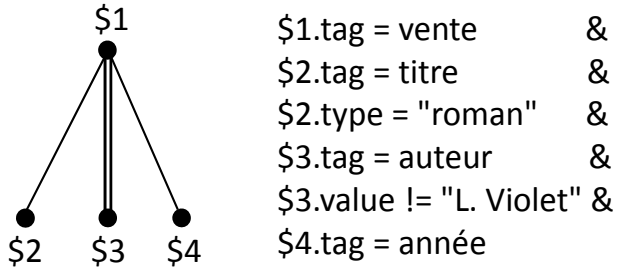



FIGURE 3.30 – Modèle d'arbre avec formule étendue

### Modèles d'arbre à nœuds-jokers et spécification d'ordre

Lu *et al.* étendent le modèle d'arbre de base par une fonction de négation, des nœuds-jokers (étiquetés du symbole $*$) et une spécification d'ordre [LLBW11]. Par exemple, le modèle d'arbre de la Figure 3.31 sélectionne les livres dont nous connaissons l'éditeur, mais pas l'année d'édition. Ici, un nœud-joker $*$ ne désigne pas un nœud de sortie, mais n'importe quel nœud dans l'arbre de données. Dans ce modèle, les nœuds de sortie sont soulignés. La spécification d'ordre, notée par

le symbole <, indique que les nœuds enfants de *vente* dans l'arbre résultat doivent respecter l'ordre indiqué dans le modèle d'arbre (*titre* avant les nœuds-jokers, de gauche à droite). Dans ce cas, "La vie sauve" est sélectionné, puisque nous connaissons l'éditeur (mais pas l'année d'édition) de ce livre dans l'arbre de données de la Figure 3.23(a).

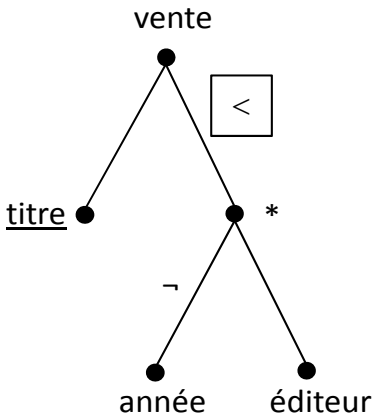



FIGURE 3.31 – Modèle d'arbre de Lu *et al.*

### 3.4.3 Discussion

Dans ce qui suit, nous discutons et comparons les modèles d'arbre présentés ci-dessus selon trois critères.

- *Efficacité du matching* : le *matching* traite deux parties du modèle d'arbre : sa structure (les arêtes et nœuds du modèle) et son contenu (la prise en compte des contraintes de la formule $F$). Nous pensons qu'un *matching* efficace implique un modèle d'arbre enrichi avec le maximum d'options de *matching* (annotations d'arêtes, opérateurs logiques, fonction d'ordre...). Utiliser de tels modèles d'arbres permet d'offrir plus de possibilités au processus de *matching* pour retourner plus d'arbres en résultat, ou de filtrer davantage le résultat. Tout dépend des besoin de l'utilisateur.
- *Possibilité de garantir l'ordre des nœuds* : dans une requête XML, il est peut être intéressant de définir l'ordre des nœuds en sortie d'une requête XML [PJ05]. Nous désignons par *possibilité de garantir l'ordre des nœuds*

la capacité d'un modèle d'arbre de modifier l'ordre des nœuds de l'arbre de données dans l'arbre résultat du *matching*. La spécification d'ordre peut être classée avec le premier point, *l'efficacité du matching*. Toutefois, vu son importance, nous préférons la tenir séparée.

- *Expressivité* : l'expressivité désigne comment un modèle d'arbre peut être implémenté dans un langage de requêtes XML (comme XQuery) ou au sein d'un SGBD natif XML offrant la possibilité de formuler des requêtes XML. Nous pensons qu'un modèle d'arbre qui ne peut pas être implémenté est généralement inutile. Ici, nous discutons des modèles d'arbre qui sont implémentés dans le cadre d'une approche d'interrogation de données XML.

**Efficacité du *matching***

Le modèle d'arbre TAX (Figure 3.23(b)) est le modèle d'arbre le plus simple qui existe dans la littérature. Aucune option de *matching* n'est associée aux arêtes et/ou aux nœuds de ce modèle d'arbre. Le *matching* est alors réalisé selon les contraintes de la formule $F$, tout en respectant les relations PE/AD entre les nœuds du modèle d'arbre. Cette simplicité provient du fait que, à l'époque de la publication de l'algèbre TAX, les recherches sur les modèles d'arbres (et XML en général) étaient dans leurs débuts. Les opérateurs TAX à base de modèles d'arbre ont en effet été proposés comme l'extension des opérateurs relationnels de sélection, de projection, de groupement, etc. [JLST01].

Associer l'option *obligatoire/optionnelle* aux arêtes du modèle d'arbre généralisé [CJLP03] permet d'augmenter le nombre d'arbres appariés dans le résultat. Cette idée provient du fait qu'il suffit qu'une seule arête du modèle d'arbre TAX soit introuvable dans l'arbre de données pour que le *matching* donne un résultat vide. Lorsqu'une telle arête est marquée comme *optionnelle* dans un modèle d'arbre généralisé, le *matching* donne en résultat tous les sous-arbres de données qui contiennent, ou pas, cette arête optionnelle. Les modèles d'arbres annotés [PWLJ04] permettent de filtrer le résultat du *matching* plutôt que de l'enrichir. Par exemple, l'annotation "-" dans le modèle d'arbre annoté de la Figure 3.25(a) permet de ne sélectionner que les livres écrits par un seul auteur.

Dans les modèles d'arbres annotés, deux cardinalités maximales sont autorisées : un ou plusieurs. Toutefois, aucune précision sur le chiffre dans le cas

plusieurs n'est présentée dans ce travail. Les modèles de graphe à distance limitée de Kimelfeld et Sagiv [KS06] permettent de résoudre ce problème puisqu'ils permettent de choisir le nombre exact de nœuds en résultat. Le degré de nœuds (nombre de nœuds enfants) [MS02] ou la fonction de négation [LLBW11] peuvent également être utilisés comme une solution à ce problème. Les nœuds-opérateurs logiques d'Izadi *et al.* [IHH09] rendent le processus de *matching* plus flexible. Cette flexibilité est surtout observée en combinant plusieurs opérateurs logiques au sein d'un seul modèle d'arbre.

Peu de différences existent entre les différentes formules associées aux modèles d'arbres. Au niveau graphique, Kimelfeld et Sagiv associent les contraintes de la formule $F$ au modèles de graphes directement [KS06], à la différence de TAX et de ses extensions, mais en pratique, cette différence reste un choix esthétique et ne change rien dans la définition de la formule. Seules les contraintes de Lakshmanan *et al.* [LRWZ04] représentent une véritable extension de la définition de base de la formule $F$.

#### Possibilité de garantir l'ordre des nœuds

Les arbres de données XML peuvent être ordonnés ou non. Dans ce dernier cas, il paraît essentiel d'associer une fonction d'ordre aux modèles d'arbres, comme c'est le cas dans XQuery [PJ05]. Paparizos et Jagadish affirment qu'il est important que les arbres de données soient munis d'une clause de type *order by* [PJ05], comme c'est le cas dans XQuery. Amer-Yahia *et al.* ajoutent que les modèles d'arbres "modernes" doivent être munis d'une fonction d'ordre [AYCS02].

Parmi tous les modèles d'arbre présentés ci-dessus, seuls le modèle d'arbre global [Che04], le modèle d'arbre annoté ordonné [PJ05] et le modèle d'arbre de Lu *et al.* [LLBW11] sont munis d'une fonction d'ordre. Les auteurs de l'algèbre TAX [JLST01] présentent également un opérateur *reordering* qui est basé sur un modèle d'arbre muni d'une spécification d'ordre.

#### Expressivité des modèles d'arbre

Le modèle d'arbre TAX et ses extensions [JLST01, CJLP03, PWLJ04, PJ05] sont implémentés au sein du SGBD natif XML TIMBER [PAKC+03]. TIMBER permet de stocker les données XML sous leur format natif et offre un outil de

requêtes supportant les fonctions du langage XQuery et les opérateurs TAX. À travers l'interface de requêtes TIMBER, les fonctions `FOR`, `LET`, `DISTINCT`, `WHERE`, `RETURN` et `ORDER BY` sont utilisées de la même façon que dans XQuery. De plus, l'algèbre TAX présente un opérateur *Group by* qui n'a pas d'équivalent en XQuery et qui peut être utilisé dans TIMBER. Lakshmanan *et al.* testent la satisfiabilité des modèles d'arbre [LRWZ04]. La satisfiabilité d'un modèle d'arbre signifie que le résultat du *matching* ne doit pas être vide. Dans ce cas, le modèle d'arbre est dit satisfiable [Hid04, LRWZ04, BFG08]. Les modèles d'arbre sont ensuite implémentés en XQuery au sein du système XQEngine [9].

L'expressivité du reste des modèles d'arbres n'est pas étudiée par leurs auteurs, car ces modèles sont implémentés dans des algorithmes d'optimisation comme les algorithmes de réécriture de requêtes, de test d'équivalence, etc.

### Synthèse générale

Dans le Tableau 3.9, nous résumons les caractéristiques de tous les modèles d'arbres présentés en fonction des trois critères utilisés dans notre discussion.

Il nous semble difficile de classer les options du *matching* du Tableau 3.9 en terme d'efficacité. Ces options diffèrent d'une approche à une autre. Chaque option est utile au *matching* en général, mais elle dépend essentiellement de l'approche en question.

Enfin, nous pensons que le fait de rassembler le maximum d'options de *matching* dans un seul modèle d'arbre offre à l'utilisateur la possibilité d'avoir un résultat plus filtré ou plus riche en arbres de données. La spécification d'ordre *O-Spec* de Paparizos et Jagadish [PJ05] est une solution simple et efficace pour spécifier l'ordre des nœuds dans l'arbre résultat du *matching*. Cette solution peut être associé à n'importe quel autre modèle d'arbre.

Enfin, l'expressivité d'un modèle d'arbre dans un langage de requête XML n'est pas une tâche facile, et elle devient de plus en plus compliquée lorsque le modèle d'arbre est très riche en options de *matching*. Toutefois, exprimer une requête utilisateur dans un modèle d'arbre est beaucoup plus facile que d'exprimer un modèle d'arbre dans un langage de requête XML. De plus, il est possible d'enrichir le modèle d'arbre avec autant d'options que l'utilisateur aura besoin, ou par des

9. `http://xqengine.sourceforge.net`

| **Modèle d'arbre** | **Options du *matching*** | **Ordre** | **Expressivité** |
|---|---|---|---|
| [JLST01] | modèle de base | non | TIMBER |
| [CJLP03] | arête indispensable/optionnelle | non | TIMBER |
| [PWLJ04] | cardinalités associées aux arêtes | non | TIMBER |
| [PJ05] | spécification d'ordre | oui | TIMBER |
| [Che04] | Ensemble de modèles d'arbres | oui | - |
| [KS06] | filtrage+classement | non | - |
| [IHH09] | nœuds-opérateurs logiques | non | - |
| [MS02] | degrés de nœuds + nœuds de sortie | non | - |
| [LRWZ04] | contraintes sur les étiquettes et les identificateurs des nœuds | non | XQuery |
| [LLBW11] | fonction de négation + nœuds-jokers | oui | - |

TABLE 3.9 – Caractéristiques des différentes modèles d'arbre : synthèse

fonctions absentes dans XQuery comme *Group by*. Ainsi, il est plus efficace de définir sa requête sous forme de modèle d'arbre enrichi avec toutes les options de *matching* nécessaires, en premier lieu, et d'implémenter ce modèle dans un système de gestion de bases de données natif XML, en second lieu, comme les modèles d'arbres TAX implémentés dans TIMBER.

### 3.4.4 Minimisation des modèles d'arbres XML

La minimisation d'un modèle d'arbre quelconque $p$ consiste à identifier et à éliminer le maximum de nœuds redondants dans $p$ [AYCLS01]. Cette idée est mo-

tivée par le fait que l'efficacité du *matching* dépend essentiellement de la taille du modèle d'arbre utilisé *matching* [AYCLS01, Ram02, FFM03, CC08]. Logiquement, plus le modèle d'arbre est réduit, plus le *matching* est effectué rapidement.

Amer-Yahia *et al.* introduisent et définissent la minimisation d'un modèle d'arbre $p$ XML comme suit [AYCLS01]. Minimiser $p$ consiste à déterminer, à partir de $p$, un deuxième modèle d'arbre $p_{min}$ tel que $p_{min}$ est inclus et équivalent à (donne le même résultat que) $p$, en éliminant les nœuds redondants dans $p$. Suite à l'élagage des nœuds redondants, la taille de $p_{min}$ (nombre de nœuds) devient inférieure à la taille de $p$.

Formellement, soient un arbre de données $T$ et un modèle d'arbre $p$ de taille (nombre de nœuds) $n$. Soit maintenant $S = \{p_i\}$ l'ensemble de modèles d'arbres $p_i$, de tailles $n_i$, inclus dans $p$ ($p_i \subseteq p$ et $n_i \leq n \; \forall \; i$). Minimiser $p$ consiste à trouver un modèle d'arbre $p_{min} \in S$ de taille $n_{min}$ tel que :

- $p_{min}$ est équivalent à $p$ (noté $p_{min} \equiv p$), c'est-à-dire que le résultat du *matching* de $p$ avec $T$ doit être identique au résultat du *matching* de $p_{min}$ avec $T$ ;
- $n_{min} \leq n_i \; \forall \; i$.

Dans certaines approches, la minimisation d'un modèle d'arbre s'effectue selon un certain nombre de contraintes d'intégrité. Il existe deux types de contraintes d'intégrité :

- chaque nœud de type $A$ ($A =$ *livre* par exemple) doit avoir un nœud enfant (respectivement, un nœud descendant) de type $B$ ($B =$ *auteur* par exemple). On écrit alors $A \rightarrow B$ ($A \Rightarrow B$, respectivement).
- chaque nœud de type $A$ ($A =$ *livre* par exemple) doit avoir un nœud descendant de type $C$ ($C =$ *nom* par exemple). Sachant que $C$ est un nœud descendant d'un troisième nœud $B$ ($B =$ *auteur* par exemple), la contrainte d'intégrité s'écrit $A \Rightarrow C$ sachant que $B \Rightarrow C$.

Par ailleurs, la minimisation des modèles d'arbres est basée sur deux tests : les tests d'inclusion et d'équivalence.

### Tests d'inclusion et d'équivalence

L'*inclusion* (*containment*) [AYCLS01, MS02] de deux modèles d'arbres $p_1$ et $p_2$ est défini par une fonction $h : p_1 \rightarrow p_2$ tels que :

- $h$ préserve les types des nœuds, c'est-à-dire $\forall x \in p_1$, $x$ et $h(x)$ doivent être du même type (*vente*, *auteur*, etc.). De plus, si $x$ est un nœud de sortie, $h(x)$ doit être également un nœud de sortie ;
- $h$ préserve les relations entre les nœuds, c'est-à-dire si deux nœuds $x$ et $y$ sont reliés par une relation de type PE (ou AD) dans $p_1$, $h(x)$ et $h(y)$ doivent être reliés par une relation du même type dans $p_2$.

Le test d'*équivalence* entre deux modèles d'arbres est défini comme un test d'inclusion à double sens [AYCLS01, FFM03]. Formellement, soient $p_1$ et $p_2$ deux modèles d'arbre, $p_1 \equiv p_2$ si et seulement si $p_1 \subseteq p_2$ et $p_2 \subseteq p_1$.

Miklau et Suciu affirment que le test d'inclusion est un problème difficile à traiter dans la majorité des cas [MS02]. Différentes méthodes ont été présentées dans la littérature pour tester l'inclusion, en utilisant des automates d'arbres (*tree automata*) [MS02] ou en extrayant les contraintes d'inclusion à partir des DTD [NS03, Woo03].

### Minimisation sans contraintes d'intégrité

Amer-Yahia *et al.* proposent CIM (*Constraint Independent Minimization*), l'algorithme de minimisation sans contraintes d'intégrité de référence [AYCLS01]. Dans ce travail, les auteurs définissent et exploitent la notion d'images de nœuds. Soit $x$ un nœud de $p$, un modèle d'arbre à minimiser. La liste d'images de $x$, notée $images(x)$, est composée de nœuds de $p$, différents de $x$ et de même type que $x$ (*auteur*, par exemple). CIM procède dans de façon ascendante (des nœuds feuilles vers le nœud racine). Soit un premier nœud feuille $y$ de $p$. CIM commence par chercher les nœuds du même type que $y$ dans l'ensemble des descendants d'$images(parent(y))$, avec $parent(y)$ le nœud parent de $y$ dans $p$. Au cas où cette recherche retourne un ou plusieurs nœuds de même type que $y$, $y$ est redondant et est supprimé de $p$. L'algorithme procède de la même façon pour le reste des nœuds de l'arbre.

Par exemple, soit le modèle d'arbre $p$ de la Figure 3.32. Soit \$5 le premier nœud

à tester. En appliquant CIM, les ensembles suivants sont construits : *parent*(\$5) = \$4, *images*(\$4) = {\$2} ; *descendants*(\$2) = {\$3}. \$5 est de même type que \$3 (*auteur*) et leurs nœuds parents sont également de même type (\$2.tag = \$4.tag = *livre*). Nous concluons donc que \$5 est redondant, il est par la suite élagué. En suivant le même raisonnement, le nœud \$4 est aussi un nœud redondant et doit être élagué. CIM détecte ensuite que le reste des nœuds (\$1, \$2 et \$3) ne sont pas redondants et retourne le modèle d'arbre minimal $p_{min}$. $p_{min}$ est bien équivalent à $p$ puisque la suppression de l'un de ses nœuds (\$1, \$2 ou \$3) signifie un résultat de *matching* différent. De plus, il est évident que $p_{min}$ est contenu dans $p$. Enfin, si nous avions choisi de commencer la minimisation par le nœud \$3 au lieu de \$5, nous aurions obtenu un deuxième arbre minimal équivalent à $p_{min}$ et constitué des nœuds \$1, \$4 et \$5.

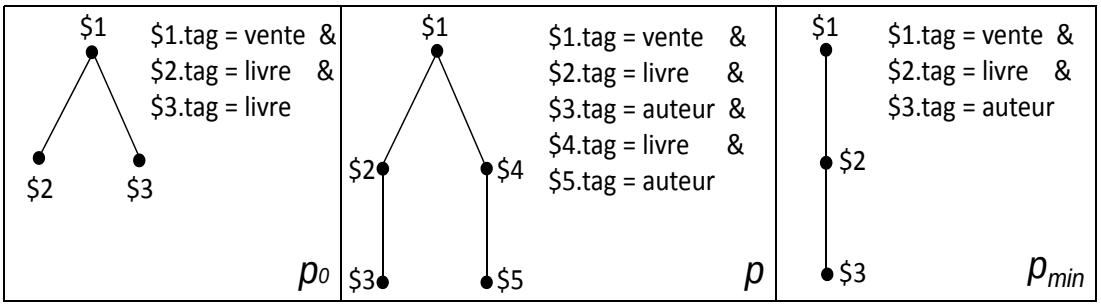



FIGURE 3.32 – Exemple de minimisation de modèle d'arbre à l'aide de l'algorithme CIM d'Amer-Yahia *et al.* [AYCLS01]

Ramanan remplace les images de nœuds de CIM par un autre type de relation, la *simulation* [Ram02]. Cette technique est également appelée *couverture* (*covering*) [CL05, CC06]. Soient $p$ un modèle d'arbre à minimiser et $x$ et $y$ deux nœuds de $p$. $y$ couvre $x$ ($x \angle y$) si et seulement si :

- $x$ et $y$ sont du même type ;
- si $x$ a un nœud enfant (ou descendant) $x'$, $y$ doit avoir un nœud enfant (ou descendant) $y'$ tel que $x' \angle y'$.

L'ensemble des nœuds qui couvrent $x$ est noté $cov(x)$. $cov(x) = x$ si $x$ est marqué comme un nœud de sortie dans $p$. Dans ces approches, la minimisation est testée, de façon descendante (en commençant pas la racine du modèle d'arbre). Soient *enfants*($x$) l'ensemble des nœuds enfants de $x$ et *descendants*($x$)

l'ensemble des nœuds descendants de $x$. $\forall\ x\ \in p$, $\forall\ x^{'} \in enfants(x)$ (respectivement $descendants(x)$). Si $\exists\ x^{''} \in$ *enfants(x)* (respectivement *descendants(x)*) tel que $x^{''} \in cov(x^{'})$, le nœud $x^{'}$ ainsi que le sous-arbre enraciné dans $x^{'}$ sont supprimés.

Chan *et al.* élaguent aussi les sous-arbres redondants d'un modèle d'arbre donné de façon récursive [CFF+02]. Un sous-arbre enraciné en $x$, noté $subtree(x)$, est minimisé dans deux étapes :

1. $\forall\ x^{'}, x^{''} \in enfants(x)$, si $subtree(x^{'}) \subseteq subtree(x^{''})$ alors $subtree(x^{'})$ est redondant. Il est donc supprimé ;
2. $\forall\ x^{'} \in enfants(x)$ (les nœuds enfants de $x$ qui n'ont pas été supprimés), $subtree(x^{'})$ est minimisé comme dans l'étape 1.

Flesca *el al.* procèdent aussi de la même façon, par suppression de sous-arbres redondants [FFM03]. La première étape est de chercher tout modèle d'arbre $p_i$ redondant par rapport à $s_p$, où $s_p$ est $p$ privé de sa racine. Formellement, l'algorithme vérifie si $p$ - $sp_i \subseteq p$ est vrai. Si c'est le cas, les sous-arbres redondants sont supprimés. Ensuite, l'algorithme est exécuté d'une façon récursive sur les sous-arbres $sp_i$ qui ne sont pas encore supprimés.

### Minimisation à base de contraintes d'intégrité

Amer-Yahia *et al.* enrichissent l'algorithme CIM par la *technique de poursuite* (*the chase algorithm*) [Ull88] et proposent l'algorithme ACIM (*Augmented* CIM) de minimisation à base de contraintes [AYCLS01]. ACIM est résumé en trois étapes.

1. Augmenter la taille du modèle d'arbre avec les nœuds et les arêtes qui correspondent aux contraintes d'intégrité. Cette opération est connue dans la littérature sous le nom de *technique de poursuite*. Par exemple, soit la contrainte d'intégrité suivante : tout *livre* doit avoir un *auteur* (*livre* $\rightarrow$ *auteur*). Dans ce cas, il faut associer à chaque nœud *livre* du modèle d'arbre à minimiser un nœud enfant *auteur* (au cas où il n'existe pas). La technique de chasse concerne seulement les nœuds originaux du modèle d'arbre, mais pas les nœuds qui ont été ajoutés au modèle d'arbre par cette technique.
2. Exécuter l'algorithme CIM [AYCLS01] sur le modèle d'arbre, sans prendre en compte les nœuds ajoutés à l'étape (1) ;

3. Supprimer dans l'arbre minimal les nœuds ajoutés par la technique de poursuite.

Si nous associons la contrainte *livre* $\rightarrow$ *auteur* au modèle d'arbre $p_0$ de la Figure 3.32, nous obtenons le modèle d'arbre $p$ de la Figure 3.32. Si nous minimisons $p$ à l'aide de l'algorithme CIM en premier lieu, et si nous supprimons les nœuds *auteur* ajoutés à l'aide de la technique de chasse ensuite, nous obtenons le modèle d'arbre minimal $p_{min}$.

L'inconvénient de la technique de chasse est que la taille du modèle d'arbre à minimiser peut augmenter de façon considérable avec le nombre de contraintes d'intégrité. Pour faire face à ce problème, Amer-Yahia *et al.* proposent l'algorithme CDM (Constraint Dependant Minimization) [AYCLS01]. Soient un modèle d'arbre $p$ et un nœud $x^{'}$ (de type $T^{'}$) redondant de $p$. CDM détecte et supprime $x^{'}$ de $p$ selon les contraintes d'intégrité suivantes.

- Soient $x$ et $x^{'}$ deux nœuds reliés avec une relation de type PE (AD, respectivement). Nous notons $parent(x^{'}) = x$ ($ancestor(x^{'}) = x$, respectivement). Si $x$ est de type $T$, alors $T \rightarrow T^{'}$ ($T \Rightarrow T^{'}$, respectivement).
- Soient $x$ et $x^{'}$ deux nœuds reliés avec une relation de type PE (AD, respectivement). Si $x$ est de type $T$, $x^{'}$ est de type $T^{'}$, et en même temps si $x$ est un nœud parent (ancêtre, respectivement) de $x^{''}$ de type $T^{''}$, alors $T^{'} = T^{''}$ ($T^{''} \Rightarrow T^{'}$ ou $T^{'} = T^{''}$, respectivement).

Ramanan étend la technique de *simulation* à la minimisation à base de contraintes d'intégrité et propose un troisième algorithme de minimisation à base de contraintes d'intégrité [Ram02]. il utilise la même technique qu'Amer-Yahia *et al.* dans cet algorithme (augmentation du modèle d'arbre, minimisation et élagage des nœuds ajoutés).

Chen et Chan jugent les contraintes d'Amer-Yahia *et al.* (contraintes FT : *Forward and subtypes constraints*) insuffisantes pour que le processus de minimisation soit totalement efficace [CC08]. Pour cela, ils proposent une nouvelle classe de contraintes plus riche, appelée *FBST-constraints*. Ces contraintes enrichissent les contraintes FT avec des contraintes dites *Backward* et *Sibling*.

- Les contraintes *Backward*, qui sont les contraintes opposées aux contraintes *Forward*, sont classifiées en deux types : parent-requis (PR) et ancêtre-requis (AR). Soient $x$ et $y$ deux nœuds du modèle d'arbre à minimiser. Une contrainte de type PR est notée $x \leftarrow y$, tandis qu'une contrainte de type

AR est notée $x \twoheadleftarrow y$. Ces contraintes signifient que pour chaque nœud de type $y$, il existe un nœud parent (ou un nœud ancêtre, respectivement) de type $x$.

- Les contraintes *Sibling* s'écrivent de la façon suivante $x \overset{z}{\dashrightarrow} y$ et signifient que pour chaque nœud de type $x$, si $x$ a un nœud enfant de type $z$, alors $x$ doit aussi avoir un nœud enfant de type $y$.

Chen et Chan proposent ensuite l'algorithme `ChaseMinimizeFBST` pour la minimisation d'un seul modèle d'arbre. Les auteurs utilisent la technique de poursuite dans leur algorithme. Soient le modèle d'arbre de la Figure 3.33(a) à minimiser et les contraintes FSBT à prendre en compte dans le processus de minimisation de la Figure 3.33(b). Ces contraintes sont ajoutées au modèle d'arbre de la Figure 3.33(a), comme le montre la Figure 3.33(c). `ChaseMinimizeFBST` détecte que le nœud $b$ est redondant, puisque le modèle d'arbre de la Figure 3.33(c) contient un deuxième nœud $b_1$ de même type. Le nœud $b$ est donc supprimé. Ensuite, `ChaseMinimizeFBST` détecte s'il n'existe pas d'autre nœud du même type que le nœud $c$ parmi les nœuds originaux du modèle d'arbre. Tous les nœuds originaux du modèle d'arbre sont testés, les nœuds ajoutés sont donc supprimés de l'arbre et nous obtenons l'arbre minimal de la Figure 3.33(d).

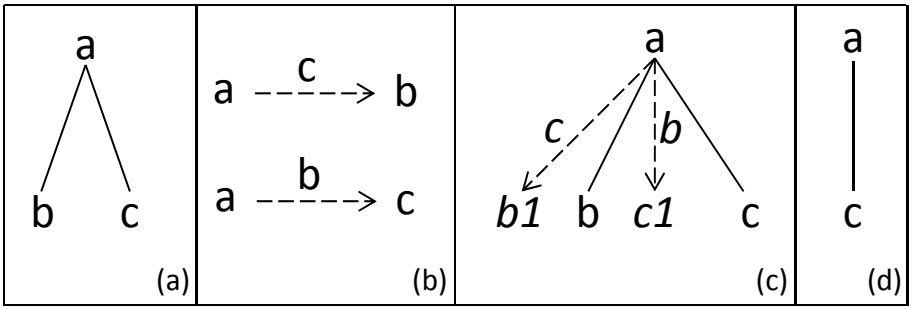



FIGURE 3.33 – Exemple de minimisation de modèle d'arbre à l'aide de l'algorithme *ChaseMinimizeFBST*

### 3.4.5 Approches holistiques

Au contraire de la minimisation, les approches holistiques ont pour but de limiter l'accès à la mémoire en contrôlant l'accès à l'arbre de données lors du processus

du *matching* [GB10, Lu10]. Ce contrôle d'accès est assuré avec les étiquettes que nous donnons aux nœuds de l'arbre de données. Selon Lu [Lu10], une approche holistique est basée sur deux étapes :

1. l'*étiquetage* : consiste à attribuer à chaque nœud $x$ d'un arbre de données $t$ une étiquette notée *label(x)*. La valeur de *label(x)* indique la position de $x$ dans l'arbre (niveau), ainsi que sa position par rapport à ces nœuds ancêtres et descendants ;
2. le *traitement* : consiste à utiliser ces étiquettes dans le processus de *matching* pour éviter de parcourir tout l'arbre de donnés.

Face au grand nombre d'approches holistiques qui existent dans la littérature, comme le montre les états de l'art de Grimsmo et Bjørklund [GB10] et de Lu [Lu10], nous choisissons de n'étudier que les approches les plus efficaces dans ce qui suit.

### Étape 1 : Étiquetage des arbres de données XML

Étiqueter les différents nœuds d'un arbre de données sert à déduire les relations entre eux (PE ou AD) à partir des étiquettes seulement [Lu10]. Deux principales familles d'approches d'étiquetage existent dans la littérature : (1) par région, aussi appelé inclusion (*containment*) et (2) Dewey.

D'autres méthodes d'étiquetage existent dans la littérature, mais elles sont peu exploitées dans les algorithmes holistiques, comme l'ordre de parcours de l'arbre (*tree-traversal order*) [LM01], l'étiquetage à base de nombres premiers [WLH04], l'étiquetage binaire [ALAA06] ou l'association d'index structurels et des listes inversées [KKNR04].

**Étiquetage par région.** Cette méthode consiste à étiqueter, dans un arbre de données, tous les nœuds avec des n-uplets (*début*, *fin*, *niveau*) [BKS02].

- *début* (*fin*, respectivement) est un compteur de la racine jusqu'au nœud *début* (le nœud *fin*, respectivement).
- *level* représente le niveau dans lequel chaque nœud $x$ est modélisé (en commençant par la racine, qui est le niveau 1).

Dans la Figure 3.34, les nœuds de l'arbre de données de la Figure 3.23(a) sont étiquetés avec un schéma d'étiquetage par région (en rouge).

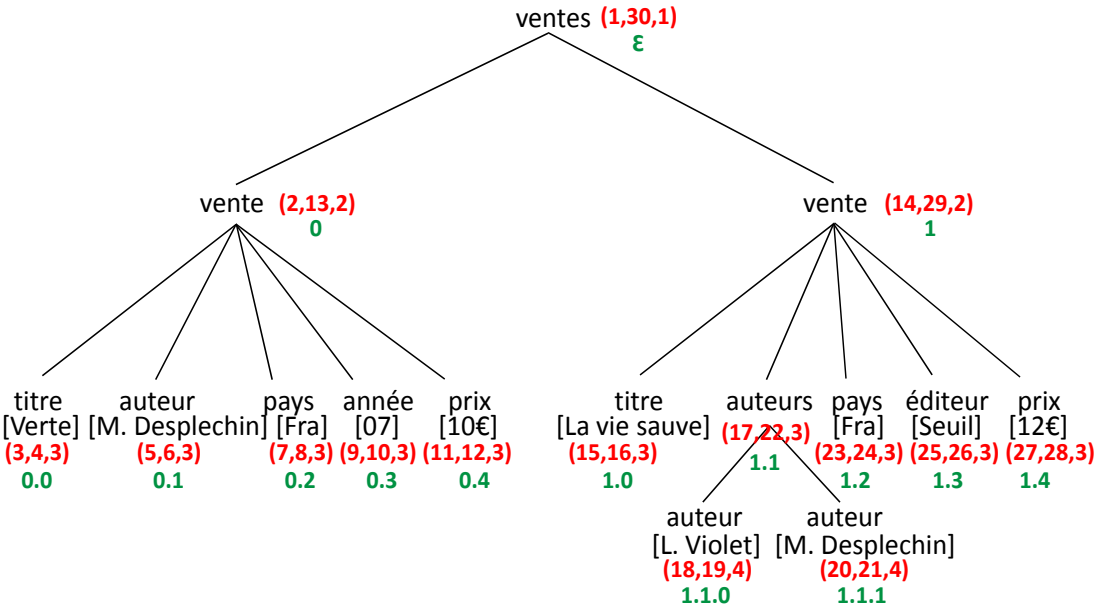



FIGURE 3.34 – Exemples d'étiquetage par région (en rouge) et Dewey (en vert)

Par exemple, le nœud *titre*[*Verte*] est étiqueté (3,4,3). Pour ce nœud, *début* est égal à 3 puisqu'il est le premier nœud fils du nœud *vente* (2,13,2) (en commençant de la gauche). Pour ce même nœud, *fin* est égal à 4 ((*début* = 3)+1) puisque ce nœud n'a pas de descendants. La valeur de *fin* pour le nœud `auteur`(17,22,3) n'est pas égale à *début + 1* puisqu'elle est déterminée après parcours de tous ses nœuds descendant. La valeur de *fin* du nœud racine est déterminée à la fin du parcours de tout l'arbre de données.

Formellement, soient $x$ et $y$ deux nœuds quelconques d'un arbre de données étiquetés par région. Soient ($D_x$, $F_x$, $N_x$) et ($D_y$, $F_y$, $N_y$) les étiquettes associées à $x$ et $y$, respectivement.

- $y$ est un nœud descendant de $x$ si et seulement si $D_x < D_y$ et $F_y < F_x$.
- $y$ est un nœud enfant de $x$ si et seulement si $D_x < D_y$, $F_y < F_x$ et N et $N_y = N_x + 1$.

Par exemple, dans la Figure 3.34, le nœud *auteur*[*L. Violet*] étiqueté avec (18,19,4) est un nœud descendant du nœud *vente* étiqueté avec (14,29,2) et, en même temps, un nœud enfant du nœud `auteurs`, étiqueté avec (17,22,3).

**Étiquetage Dewey.** Cette technique permet d'étiqueter les nœuds d'un arbre de données de façon séquentielle [TVB+02]. Le nœud racine est étiqueté $\epsilon$. Les nœuds enfants de la racine sont étiquetés 0, 1, 2, etc. (de gauche à droite). Dans le cas général, soit un nœud $x$ quelconque (différent de la racine) dans un arbre de données $t$. Soit $label(x)$ l'étiquette de $x$ dans $t$. Les nœuds enfants de $x$ sont étiquetés $label(x).0$, $label(x).1$, $label(x).2$, etc. (de gauche à droite). De cette façon, à partir de l'étiquette de n'importe quel nœud de $t$, nous pouvons déduire l'ensemble de ses ancêtres.

Dans la Figure 3.34, les nœuds de l'arbre de données de la Figure 3.23(a) sont étiquetés avec des étiquettes Dewey (en vert). Dans cet exemple, le nœud *auteur*[*L. Violet*] est étiqueté 1.1.0, ce qui signifie que ce nœud est le premier nœud enfant (de gauche à droite) de l'élément étiqueté 1.1 et le nœud descendant du nœud étiqueté 1.

À l'aide des informations extraites des schémas XML et des DTDs, Lu *et al.* étendent l'étiquetage Dewey pour prendre en compte les noms des nœuds en plus de leurs étiquettes [LLCC05]. Les auteurs associent tout simplement à chaque étiquette Dewey le nom du nœud correspondant. De cette façon, nous pouvons déduire le chemin entre la racine et le nœud en question. Par exemple, à partir du nœud étiqueté 1.1.0, nous pouvons déduire le chemin ventes/vente/auteurs/auteur.

L'inconvénient de cette méthode est que le réétiquetage des documents XML dynamiques (qui supportent l'insertion et la suppression de nœuds) est une opération coûteuse. Pour faire face à ce problème, Xu *et al.* proposent une nouvelle approche d'étiquetage d'arbres de données, appelée *étiquetage Dewey dynamique* [XLWB09]. Cette approche consiste à mettre à jour les étiquettes de l'arbre de données au moment de l'insertion ou de la suppression d'un nœud de l'arbre. Pour préserver l'ordre relatif des nœuds, Xu *et al.* définissent une opération d'addition pour les étiquettes de même longueur. Soient $A$ et $B$ deux étiquettes de même longueur $m$. Formellement, nous écrivons $A : a_1.a_2...a_m$ et $B : b_1.b_2...b_m$, $A + B = (a_1 + b_1).(a_2 + b_2)...(a_m + b_m)$. Pour pouvoir additionner les deux étiquettes des nœuds $A$ et $B$, $A$ et $B$ doivent avoir le même nœud parent.

Soit l'exemple de la Figure 3.35. Les nœuds marqués avec des lignes discontinues sont les nœuds que nous avons ajoutés à l'arbre de données, composé à la base des nœuds avec des lignes continues. Les nœuds sont insérés dans l'ordre alphabétique (A, B, C, ..., I).

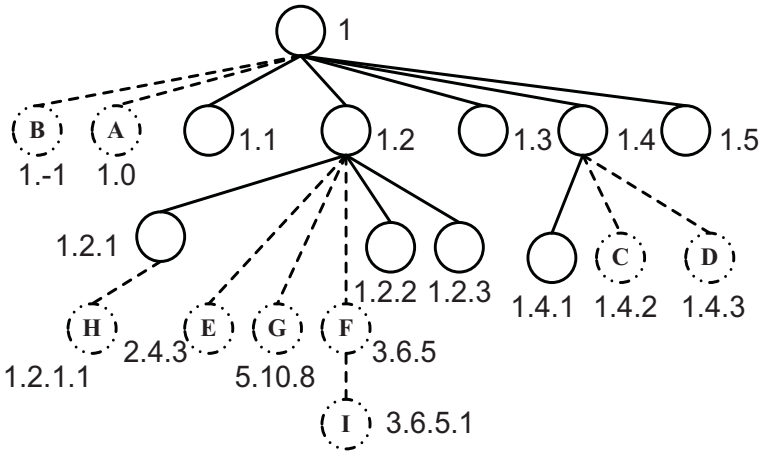



FIGURE 3.35 – Exemple d'étiquetage Dewey dynamique [XLWB09]

Le nœud A est inséré avant le premier nœud enfant de la racine (1.1), donc il acquiert l'étiquette 1.0, car A est devenu le premier nœud enfant de la racine (de gauche à droite). Lorsque le nœud B est inséré à gauche du nœud A, il devient le premier nœud enfant de la racine et il acquiert l'étiquette 1.-1. Les nœuds C et D sont insérés à droite du nœud étiqueté avec 1.4.1, et acquièrent les étiquettes 1.4.2 et 1.4.3, respectivement. Comme E est inséré entre les deux nœuds étiquetés avec 1.2.1 et 1.2.2, son étiquette est égale à 2.4.3 = 1.2.1 + 1.2.2. Les étiquettes de F et G sont calculés de la même façon. Comme les nœuds H et I sont les enfants respectifs des nœuds étiquetés 1.2.1 et 3.6.5 et qu'ils n'ont pas de nœud enfant, ils deviennent leurs premiers enfants et acquièrent les étiquettes 1.2.1.1 et 3.6.5.1, respectivement. Dans cette approche, les auteurs étiquettent avec *label(x).1* le premier enfant d'un nœud $x$ donné, et non avec *label(x).0*, comme dans l'étiquetage Dewey classique. De plus, ils étiquettent la racine 1 au lieu de $\epsilon$ pour permettre les calculs. En cas d'insertion de nouveaux nœuds, les étiquettes des premiers enfants peuvent avoir des valeurs négatives, comme l'étiquette du nœud B. Ce choix ne change rien dans la définition de base de l'étiquetage Dewey. Enfin, les auteurs ajoutent qu'en cas de suppression d'un nœud, l'ordre des nœuds est conservé.

**Synthèse générale.** Dans ce qui précède, nous avons pu vérifier, à l'aide d'exemples réels, que la différence majeure entre l'étiquetage par région et l'étiquetage

Dewey se résume dans la façon de déterminer les relations structurelles entre les nœuds de l'arbre de données. Avec l'étiquetage Dewey, nous pouvons déduire, à partir d'une seule étiquette, les étiquettes de tous ses nœuds ascendants jusqu'à la racine. Dans l'arbre de données de la Figure 3.34, nous pouvons déduire que le nœud étiqueté 0.2 est au troisième niveau de l'arbre, que son nœud parent est étiqueté 0, et qu'il existe au moins un autre nœud qui a le même parent que lui, le nœud étiqueté 0.1. Avec l'étiquetage par région, il faut tester les étiquettes de deux nœuds pour savoir s'ils sont reliés avec une relation de type PE (ou AD) ou non. Dans l'arbre de données de la Figure 3.34, à partir des deux étiquettes *(17,22,3)* et *(18,19,4)*, nous pouvons déduire que les deux nœuds en question sont reliés avec une relation de type PE (18 = 17 + 1). Enfin, Lu remarque qu'il est évident qu'une mise à jour des étiquettes Dewey est plus facile qu'une mise à jour des étiquettes région [Lu06].

### Étape 2 : Traitement

Une fois étiqueté, l'arbre de données est prêt pour la deuxième étape : le traitement. L'idée du traitement est d'utiliser les étiquettes pour accéder aux nœuds du modèle d'arbre au moment du *matching*, sans être obligé de parcourir la totalité de l'arbre de donnés comme dans le *matching* traditionnel. Plus simplement, le traitement consiste à construire, pour chaque nœud, une liste qui contient toutes les étiquettes des nœuds du même type. Cette liste est ensuite utilisée pendant le *matching*.

Soit l'exemple du modèle d'arbre de la Figure 3.36(a), qui sélectionne les ventes de livres de la Figure 3.34 par *titre* et par *auteur*. Dans la Figure 3.36(b), pour chaque type de nœud, nous construisons ses listes d'étiquettes dans l'arbre de données. Cet exemple est appelé l'*approche naïve* par Lu [Lu10]. Nous décrivons dans ce qui suit les algorithmes holistiques proposés pour la construction des résultats de *matching* à partir des listes d'étiquettes par région et Dewey.

***TwigStack.*** Le principe de *TwigStack* se résume dans la fusion de résultats intermédiaires construits à partir d'étiquettes afin de construire les arbres résultats [BKS02]. Un résultat intermédiaire est l'association de deux étiquettes qui correspondent à deux nœuds reliés dans le modèle d'arbre. Les relations PE/AD entre

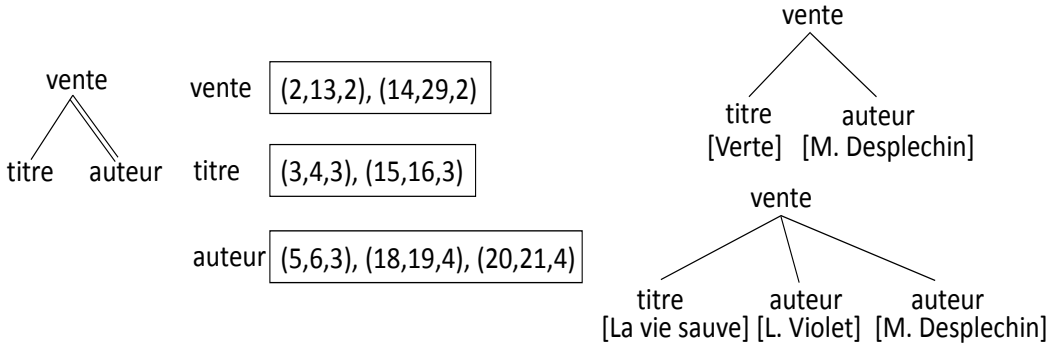



FIGURE 3.36 – Modèle d'arbre (a), listes d'étiquettes par région (b) et résultat (c)

les nœuds du modèle d'arbre sont respectées lors de la construction des résultats intermédiaires.

Le résultat du *matching* entre le modèle d'arbre de la Figure 3.36(a) et le modèle d'arbre de la Figure 3.34 est illustré dans la Figure 3.36(c). *TwigStack* construit ce résultat à partir des résultats intermédiaires suivants, déduits des listes d'étiquettes de la Figure 3.36(b).

- *vente/titre* : (2,13,2)(3,4,3) ; (14,29,2)(15,16,3)
- *vente//auteur* : (2,13,2)(5,6,3) ; (14,29,2)(18,19,4) ; (14,29,2)(20,21,4)

Après jointure des résultats intermédiaires, nous obtenons les résultats suivants, qui correspondent aux arbres résultats de la Figure 3.36(c).

- (2,13,2)(3,4,3)(5,6,3)
- (14,29,2)(15,16,3)(18,19,4)(20,21,4)

*TwigStack* est optimal pour les relations de type AD du modèle d'arbre [LCL04, Lu10]. En réalité, cet algorithme est conçu pour les requêtes qui ne contiennent que des relations AD. Et en cas de relations PE dans les modèles d'arbres, il peut générer de nombreux résultats intermédiaires inutiles (qui feront pas parties du résultat final). De plus, *TwigStack* ne prend pas en compte les prédicats d'ordre qui peuvent être associés aux modèles d'arbres, ni les nœuds marqués avec le signe *.

**Extensions de *TwigStack*.** *TwigStackList* permet de prendre en compte les résultats intermédiaires dans les relations PE [LCL04]. L'idée des auteurs est de

stocker les étiquettes correspondant aux nœuds parents des relations PE dans des listes. Une fois ces listes constituées, les résultats intermédiaires sont correctement pris en compte. En fait, *TwigStack* peut produire de faux résultats en cas d'utilisation de modèles d'arbres avec des relations PE. Par contre, seuls les relations AD ramifiées [10] sont prises en compte par *TwigStackList*, contrairement à *TwigStack* qui prend en compte toutes les relations AD d'un modèle d'arbre, comme le montre la Figure 3.37.

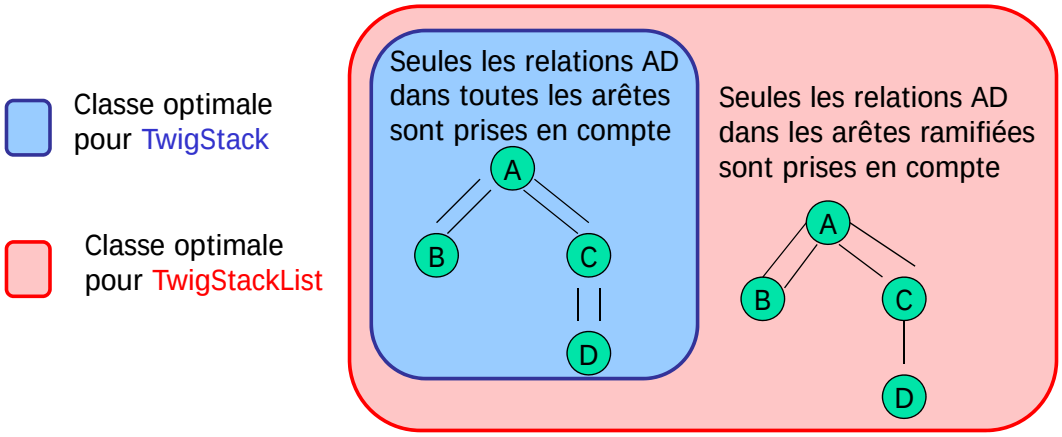



FIGURE 3.37 – Comparaison de *TwigStack* et *TwigStackList* [Lu10]

Dans le même contexte, *Twig²Stack* réduit la taille des résultats intermédiaires [CLT+06]. L'idée des auteurs est d'associer à chaque type de nœud du modèle d'arbre une pile hiérarchique. Pour chaque pile, les nœuds sont insérés en commençant des feuilles vers la racine. Une fois toutes les piles hiérarchiques sont construites, le *matching* est réalisé par la fusion des différentes piles hiérarchiques en respectant les relations hiérarchiques indiquées dans le modèle d'arbre. Finalement, Baca *et al.* fusionnent *TwigStack* et *Twig²Stack* en un seul algorithme pour réduire encore la taille des résultats intermédiaires [BKS08].

***OrderedTJ.*** Les algorithmes *TwigStack* et *TwigStackList* ne prennent pas en compte les modèles d'arbres munis d'une spécification d'ordre. *OrderedTJ* étend donc *TwigStackList* aux modèles d'arbres ordonnés [LLY+05].

10. Un nœud ramifié (*branching node* en Anglais) est un nœud dont le nombre d'enfants est supérieur à un. Une arête qui part d'un nœud ramifié est appelé une arête ramifiée [LCL04].

Supposons que nous ajoutions au modèle d'arbre de la Figure 3.36(a) la spécification d'ordre suivante : *titre* < *auteur* (le nœud *titre* est sélectionné avant, c'est-à-dire à gauche, du nœud *auteur*). Dans ce cas, *OrderedTJ* ne prend en compte, dans le résultat du *matching*, que les résultats intermédiaires *vente//auteur* qui respectent cette spécification d'ordre. Pour atteindre cet objectif, il suffit de comparer les valeurs des étiquettes. Par exemple, il est très facile de déduire que l'étiquette (3,4,3), qui correspond à *titre*, est située avant (à gauche) de l'étiquette (5,6,3), qui correspond à *auteur*, dans l'arbre de données de la Figure 3.34.

*OrderedTJ* est optimal pour les relations de type AD dans les modèles d'arbre munis d'une fonction d'ordre, mais à partir de la deuxième arête ramifiée [Lu10]. En d'autres termes, *OrderedTJ* est optimal pour les modèles d'arbres avec une relation de type PE dans la première arête.

***TJFast.*** Lu *et al.* partent de l'idée qu'il est plus facile de déduire les étiquettes et les noms de tous les nœuds ancêtres dans le cas de l'étiquetage Dewey. En fait, à partir d'une seule étiquette Dewey, nous pouvons déduire les étiquettes de tous les nœuds ascendants du nœud en question jusqu'à la racine. Cette opération nécessite plus qu'une étiquette dans le cas d'un étiquetage par région. Les auteurs proposent l'algorithme holistique *TJFast* sur cette base [LLCC05].

Le principe de *TJFast* est le suivant. Des listes $T_x$ qui contiennent toutes les étiquettes Dewey des éléments de type $x$ sont construites en premier lieu. Les étiquettes de chaque liste sont organisées selon l'ordre des éléments correspondants dans le modèle d'arbre. Comme dans les approches précédentes, une fois les résultats intermédiaires construits, il ne reste qu'à les joindre pour obtenir le résultat final. Soient le modèle d'arbre de la Figure 3.36(a) et l'arbre de données de la Figure 3.34, la jointure des fragments de l'arbre de données correspondants aux résultats intermédiaires suivants donne le résultat de la Figure 3.36(c).

- *vente/titre* : 0/0.0 ; 1/1.0
- *vente//auteur* : 0/0.1 ; 1/1.1.0 ; 1/1.2.0

Finalement, *TJFast+L* étend *TJFast* par la prise en compte des étiquettes et des noms de nœuds dans son processus [Lu06]. *TJFast+L* optimise *TJFast* en réduisant l'accès à la mémoire lors du processus de *matching* et en prenant en compte toutes les relations PE du modèle d'arbre en entrée, comme le montre la Figure 3.38.

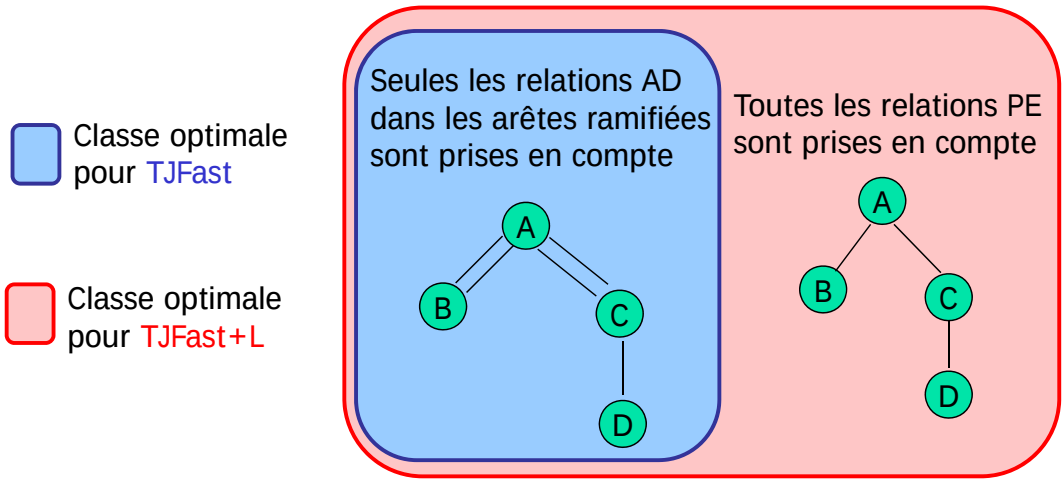



FIGURE 3.38 – Comparaison de *TJFast* et *TJFast+L* [Lu10]

***TreeMatch.*** Lu *et al.* démontrent de façon théorique que la raison intrinsèque de sous-optimalité des algorithmes holistiques (due à des résultats intermédiaires inutiles au calcul du résultat final) est causée par des nœuds médiateurs lors du processus de *matching* [LLBW11]. Soit l'exemple de la Figure 3.39.

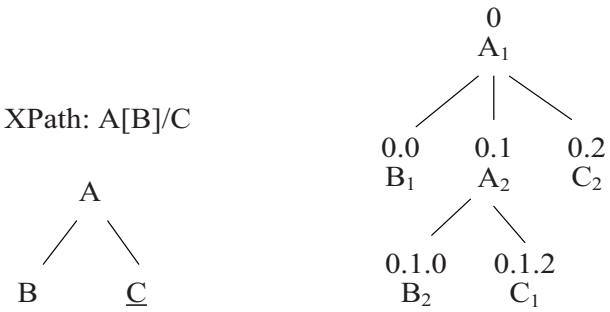



FIGURE 3.39 – Premier exemple de modèle d'arbre (à gauche) et d'arbre de données (à droite) utilisés dans *TreeMatch* [LLBW11]

Le *matching* traditionnel vérifie, et pour chaque sous-arbre de l'arbre de données de la Figure 3.39, toutes les relations de type $AB$, et puis il vérifie toutes les relations $AC$. À la fin, ces résultats sont associés ensemble pour constituer le résultat final. $A$ est un nœud médiateur dans cet exemple.

Dans ce qui suit, nous démontrons comment les nœuds médiateurs peuvent

être à l'origine de sous-optimalité des algorithmes holistiques. Dans la Figure 3.39, le nœud $C$ (souligné) du modèle d'arbre modélise le nœud de sortie. L'arbre de données est étiqueté avec un étiquetage Dewey. Quand la condition $A_1/B_1$ est vérifiée, l'algorithme holistique ne peut pas déterminer, à ce stade, si $A_1$ dispose d'un nœud de type $C$ ou non. Dans ce cas, l'algorithme holistique peut adopter deux stratégies :

- garder les résultats intermédiaires inutiles de type $A/B$ (dans le cas où $A$ n'a pas de nœud enfant de type $C$) ;
- ne pas garder les résultats intermédiaires de type $A/B$ et prendre le risque d'écarter une réponse utile.

Pour résoudre ce problème, Lu *et al.* proposent l'algorithme *TreeMatch* qui consiste à modifier la façon suivant laquelle les nœuds médiateurs sont pris en compte. Leur idée consiste à sélectionner les nœuds médiateurs après la sélection de tous les nœuds feuilles qui sont des réponses possibles à la requête. Par exemple, soit le modèle d'arbre de la Figure 3.40.

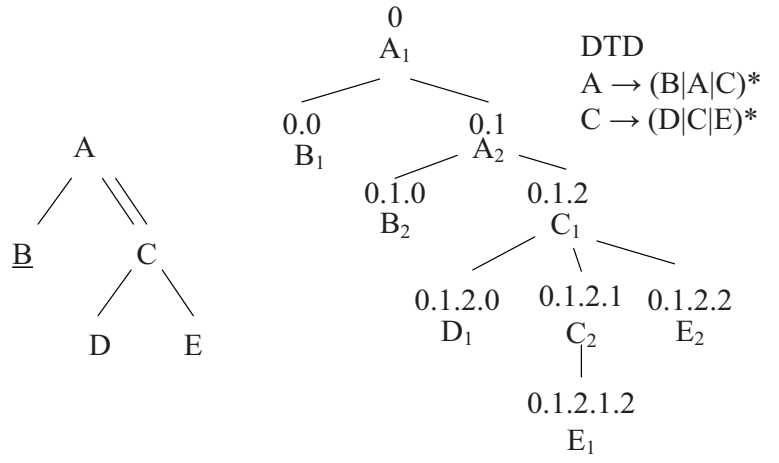



FIGURE 3.40 – Deuxième exemple de modèle d'arbre (à gauche) et d'arbre de données (à droite) utilisés dans *TreeMatch* [LLBW11]

Le nœud $B$ (souligné) est le nœud de sortie. *TreeMatch* commence par sélectionner les nœuds de types $B$, $D$ et $E$ ($B_1$, $B_2$, $D_1$, $E_1$ et $E_2$) de l'arbre de données puisque les nœuds de type $A$ et $C$ ($A_1$, $A_2$, $C_1$ et $C_2$) sont des nœuds médiateurs. Ensuite, les nœuds médiateurs sont traités dans l'ordre ascendant. *TreeMatch* détecte que

le nœud $C_2$ ne participe pas au résultat final puisqu'il n'a pas de nœud enfant de type $D$. De cette manière, il évite de sélectionner le résultat intermédiaire $C_2/E_1$, inutile pour le calcul de la réponse finale. Enfin, *TreeMatch* détecte que $A_1$ et $A_2$ participent au résultat final et déduit, par la suite, que les nœuds $B_1$ et $B_2$ sont les nœuds résultats du *matching* entre le modèle d'arbre et l'arbre de données de la Figure 3.40. Finalement, les auteurs étendent l'algorithme *TreeMatch* aux modèles d'arbres munis d'une fonction d'ordre et d'une fonction de négation.

*TreeMatch* est optimal pour les requêtes qui contiennent des arêtes AD ramifiées qui ne sont pas reliées à des nœuds de sortie, comme le montre la Figure 3.41.

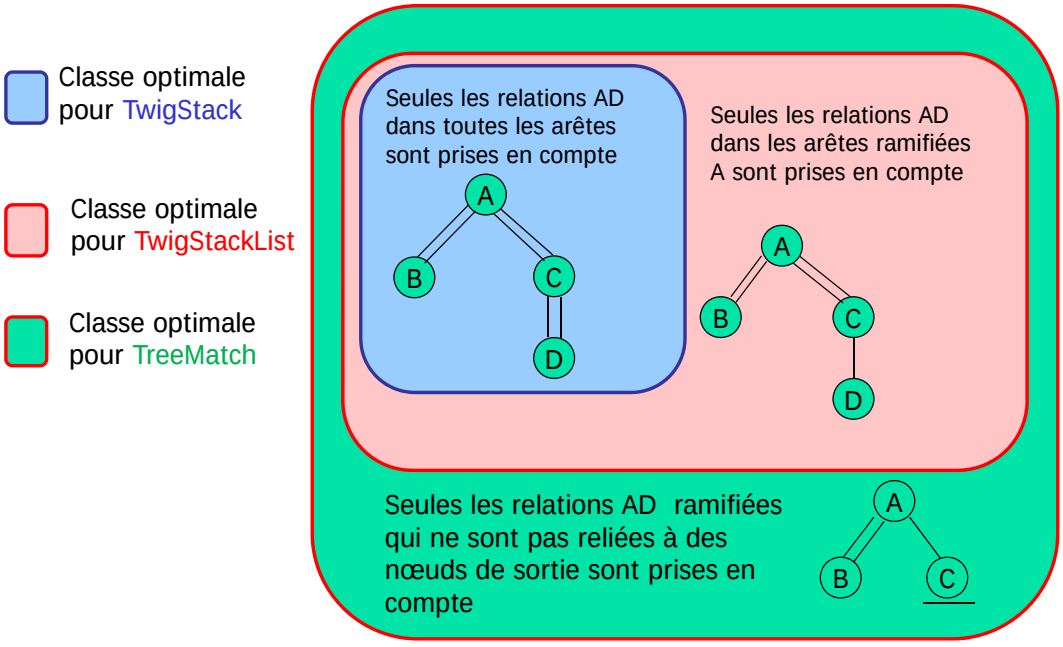



FIGURE 3.41 – Optimalité de *TreeMatch* [Lu10]

### 3.4.6 Autres utilisations

En plus des travaux d'optimisation du *matching*, les modèles d'arbres XML constituent le corps de plusieurs applications basées sur XML comme :

- le traitement des requêtes en mémoire cache (*query caching*) [YLH03] ;
- le routage dans les réseaux pair-à-pair [TH04] ;

- la conception des entrepôts de données XML [ZLBT03] ;
- le XML actif [MHZ08, ABM09] ;
- l'extraction de connaissances à partir des données (par exemple, l'identification de communautés d'utilisateurs [CFG07]).
  Nous étudions quelques exemples de ces approches dans ce qui suit.

**Fouille de modèles d'arbre**

La fouille de modèles d'arbres XML revient actuellement à la recherche des modèles d'arbres fréquents dans une collection de modèles d'arbres. Cette technique de fouille est utilisée, par exemple, dans la gestion des modèles d'arbres en mémoire dans le but d'améliorer le temps de réponse des requêtes [YLH03], pour construire des schémas d'entrepôts à partir des requêtes d'utilisateurs historisées [ZLBT03] ou pour l'administration de sites Web par la fouille des flux de données [HWC06].

- ***Formulation du problème :*** Soient $C = \{p_1, p_2, ..., p_n\}$ une collection de $n$ modèles d'arbres et $minsup \in ]0, 1]$ le support minimum. Le support de tout arbre de données $r$ est calculé par $sup(r) = freq(r)/n$, où $freq(r)$ est le nombre d'occurrences de $r$ dans $C$. Le problème de fouille de modèles d'arbres fréquents dans $C$ consiste à déterminer l'ensemble $F = \{r_1, r_2, ..., r_m\}$ des $m$ modèles d'arbres tel que $\forall\, i \in [1, m]$, $sup(r_m) \geq minsup$ [YLH03]. Par example, soient la collection de modèles d'arbres $C$ et le modèle d'arbre $r$ de la Figure 3.42. Puisque $r \subseteq p_1$ et $r \subseteq p_2$, $freq(r) = 2$ et $sup(r) = \frac{2}{3}$. Si $minsup = \frac{1}{2}$, alors $r$ est considéré comme fréquent. Notons que la recherche de modèles d'arbres (ou d'arbres en général) fréquents se résume à un test d'inclusion (Section 3.4.4).

- ***Algorithmes de recherche de modèles d'arbres fréquents :*** Yang *et al.* s'inspirent de l'algorithme de recherche de motifs fréquents *Apriori* [AS94] et proposent l'algorithme *XQPMiner* de recherche de modèles d'arbres XML fréquents [YLHA03]. *XQPMiner* initialise tout d'abord $F_1$ avec tous les modèles d'arbres 1-*arête* fréquents de $C$. Un modèle d'arbre 1-*arête* est un modèle d'arbre qui ne contient pas de nœud ramifié. Chaque étape $i$

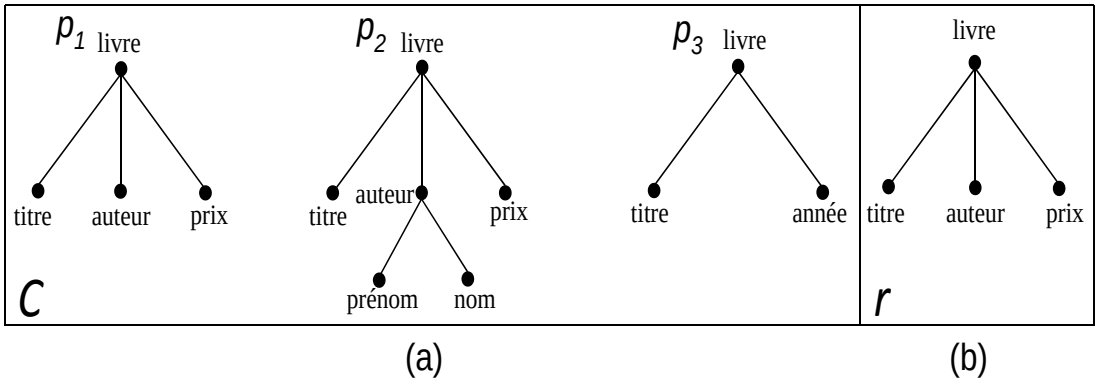



FIGURE 3.42 – Collection de modèles d'arbre $C$ (a) et modèle d'arbre fréquent $r$ (b)

suivante est constituée de deux sous-étapes :

1. les arbres *i-arête* candidats sont construits à partir des arbres $(i-1)$-*arête*. Ils sont ensuite filtrés selon la valeur du support minimum ;
2. l'inclusion des arbres *i-arête* est testée dans la collection $C$ pour calculer les valeurs de leur support.

*FastXMiner* optimise *XQPMiner* par la construction du modèle d'arbre global [YLH03]. Le modèle d'arbre global est construit suite à la fusion des différents modèles d'arbres correspondants à $C$ (Section 3.4.2). De cette façon, le problème de recherche de modèles d'arbres fréquents devient limité à l'identification de tous les sous-modèles d'arbres dans ce modèle d'arbre global.

### Réécriture de requêtes

Dans le cas général, réécrire une requête $Q$ sur une base de données quelconque $D$, consiste à déduire une requête $Q'$ sur une vue $V$ déduite de $D$, telle que $Q$ et $Q'$ retournent le même résultat. L'idée générale des approches de réécriture de modèles d'arbres consiste alors en un test d'inclusion pour identifier l'ensemble minimal de requêtes garantissant un *matching* réussi [BOB+04, LWZ06, WWL09].

La réécriture de modèles d'arbres a été utilisée dans les systèmes de réseaux pair-à-pair. Par exemple, dans le système Piazza [TH04], les données sont modélisées en XML et les requêtes en XQuery. Le problème d'optimisation de requêtes

revient donc à un problème d'optimisation du *matching*. Par exemple, une requête donnée peut être distribuée sur plusieurs chemins dans le système de réseau pair-à-pair, ce qui rend cette requête (et les requêtes similaires) redondantes. L'idée est alors d'éliminer toutes les requêtes redondantes par un simple test d'inclusion entre les requêtes existantes et la nouvelle requête.

#### Test de satisfiabilité des modèles d'arbres pour les documents XML actifs

Cette approche consiste à tester la satisfiabilié des modèles d'arbres sur les documents XML actifs [MHZ08, ABM09]. Un document XML actif contient, en plus des données XML traditionnelles, les références vers des services Web [ABM+02]. Lorsqu'un service Web est appelé, le résultat est inséré dans le document XML. Dans ce contexte, la satisfiabilité des modèles d'arbres est testée pour éviter les requêtes inutiles. La satisfiabilité est surtout utilisée pour ne déterminer que les services Web qui contribuent au résultat final.

#### Identification de communautés sémantiques d'individus

L'identification de communautés sémantiques d'individus consiste à définir les communautés d'individus qui partagent les mêmes centres d'intérêts. Ces centres d'intérêts sont modélisés par des modèles d'arbres. Dans ce contexte, Chand *et al.* remplacent la notion d'équivalence dans les modèles d'arbre par la notion de similarité [CFG07]. Les auteurs définissent la similarité comme suit. Soient $S$ un ensemble de modèles d'arbres un ensemble de modèles d'arbres et $D$ un ensemble d'arbres de données. Soient $p$ et $q$ deux modèles d'arbres de $S$. La similarité de $p$ et $q$ est définie par une fonction $sim : S^2 \rightarrow [0, 1]$ tel que $sim(p, q)$ est la probabilité que $p$ sélectionne le même fragment de $D$ que $q$.

### 3.4.7 Synthèse générale

Dans cette section, nous avons présenté une étude des modèles d'arbres XML, qui sont considérés aujourd'hui comme des outils importants dans l'optimisation des requêtes XML. Nous avons comparé les différentes structures que peuvent prendre les modèles d'arbres. Nous concluons après cette étude que le modèle

d'arbre le plus efficace est un modèle d'arbre enrichi avec le maximum d'options de *matching*. Un modèle d'arbre doit enfin pourvoir s'exprimer en XPath/XQuery.

Dans une seconde étape, nous avons étudié les deux grands types d'optimisation du *matching* : la minimisation des modèles d'arbres et les algorithmes holistiques. Nous avons étudié les algorithmes les plus importants de ces deux familles. Lu *et al.* proposent plusieurs algorithmes holistiques pour optimiser le *matching* en limitant l'utilisation de la mémoire et prendre en compte le maximum de relations entre nœuds (PC, AD, ramifiées ou pas), ce qui signifie la prise en compte de la classe la plus large de requêtes. Or, nous remarquons que *TreeMatch* [LLBW11], l'algorithme holistique le plus récent, optimise les algorithmes existants en élargissant les classes de requêtes prises en compte. Toutefois, il est loin d'être l'algorithme optimal pour tous types de requête. Nous concluons alors, et en dépit de la maturité des recherches sur les modèles d'arbres et l'optimisation du *matching*, que plusieurs efforts sont encore nécessaires pour trouver l'algorithme holistique optimal pour tous type de modèle d'arbre.

Nous pensons que l'association d'un algorithme de minimisation et d'un algorithme holistique peut être efficace pour optimiser le *matching*. Comme nous le savons déjà, la minimisation prend en compte les modèles d'arbres dans son fonctionnement, tandis que les algorithmes holistiques prennent en compte les données. Donc, ces deux techniques sont complémentaires. Tous les algorithmes holistiques, de *TwigStack* [BKS02] à *TreeMatch* [LLBW11], sont optimaux pour une partie du modèle d'arbre. Par exemple, *TreeMatch* est optimal pour les requêtes qui contiennent des arêtes AD ramifiées qui ne sont pas reliées à des nœuds de sortie. Notre idée d'associer un algorithme de minimisation à un algorithme holistique est décrite comme suit. Au début, l'algorithme de minimisation élimine les nœuds redondants du modèle d'arbre. Dans une étape suivante, l'algorithme holistique utilise ce modèle d'arbre minimal dans son fonctionnement. Dans ce cas, le *matching* serait plus efficace puisqu'un modèle d'arbre minimal limite l'utilisation de la mémoire d'un coté. D'un autre coté, le *matching* serait plus efficace si les nœuds élagués ne font pas partis des nœuds pour lesquels l'algorithme est optimal. Enfin, nous avons vu comment les modèles d'arbres XML ont été utilisés dans plusieurs approches basées sur XML, comme le XML actif, la fouille de données, etc. L'étude de tels travaux montre que les modèles d'arbres peuvent être utilisés dans différentes domaines basés sur XML.

En général, même si les recherches sur les modèles d'arbres XML, qui ont débuté il y a un peu plus d'une décennie, apparaissent assez matures, les modèles d'arbres sont toujours la cible de critiques et d'amélioration. Par exemple, certaines applications récentes nécessitent l'intégration et l'interrogation de flux de données hétérogènes. Dans ce genre d'applications, le langage XML est fortement recommandé pour modéliser les flux de données et les modèles d'arbres XML semblent être une solution efficace pour l'interrogation des données. Toutefois, la recherche par mots-clés ne peut pas être exprimée avec les modèles d'arbres existants [TW09]. Dans ce contexte, Theodoratos et Wu proposent des modèles d'arbres partiels qui permettent d'adapter les nœuds du modèle d'arbre à la nature hétérogène des données de flux [TW09]. Les modèles d'arbres partiels sont tout simplement des modèles d'arbres généralisés (Section 3.4.2).

Notre objectif derrière cette étude est de chercher un modèle d'arbre pour données multidimensionnelles qui puisse servir de base pour des opérateurs XOLAP. Or, la seule approche qui utilise des modèles d'arbres dans un contexte de données multidimensionnelles est l'approche de conception des entrepôts de données XML de Zhang *et al.* [ZLBT03]. Dans cette approche, les schémas des documents XML (à intégrer dans l'entrepôt de données XML) sont construits en utilisant les modèles d'arbres fréquents (Section 3.4.6). Les modèles d'arbres fréquents sont déduits à partir des requêtes historisés par l'utilisateur. L'intégration des données dans l'entrepôt de données est réalisée à l'aide de la technique de classification hiérarchique (*hierarchical clustering technique*).

Toutefois, les modèles d'arbres de Zhang *et al.* ne nous sommes pas utiles puisqu'ils sont conçus pour la conception d'entrepôts XML et ne pas pour l'interrogation des données multidimensionnelles. Notre étude des options de *matching* des différents modèles d'arbres a été donc fructueuse puisque certaines options de *matching* semblent utiles à nos objectifs. Par exemple, les arêtes optionnelles de Chen *et al.* [CJLP03] peuvent être utilisées dans l'interrogation des hiérarchies non-couvrantes en définissant, par exemple, les nœuds correspondants aux données manquantes comme optionnels dans le modèle d'arbre. De même, les annotations de Paparizos *et al.* [PWLJ04] sont aussi utiles pour le *matching* des hiérarchies non-strictes. Par exemple, l'annotation (-) qui autorise un et un seul *matching* peut être une solution efficace pour éviter d'interroger plusieurs nœuds de même type et de même contenu. D'un autre côté, nous remarquons que les options de

*matching* de tous les modèles d'arbres étudiés dépendent essentiellement des besoins de l'utilisateur. Nous déduisons donc que nous pouvons définir les options de *matching* nécessaires pour le traitement des hiérarchies complexes et la garantie de l'additivité.

À première vue, les approches d'étiquetage (Dewey et par région) semblent possibles à utiliser pour détecter et traiter l'additivité dans les hiérarchies complexes. Par exemple, si nous comparons les étiquettes Dewey de deux nœuds de même type, nous pouvons vite déterminer si les hiérarchies en question sont des hiérarchies non-couvrantes ou non. Plus explicitement, si un nœud $x$ est étiqueté avec une étiquette Dewey qui contient moins d'éléments qu'une deuxième étiquette Dewey d'un nœud de même type que $x$, alors la hiérarchie contenant le nœud $x$ est non-couvrante. Par exemple, si un nœud *ville* $v_1$ est étiqueté 1.1 et un autre nœud *ville* $v_2$ est étiqueté 2.1.1, dans une hiérarchie *pays-région-ville*, nous pouvons conclure que la hiérarchie reliée à $v_1$ est non-couvrante.

# Chapitre 4

# Traitement des problèmes d'additivité dans les hiérarchies complexes

Dans cette thèse, notre objectif est de présenter un opérateur XOLAP d'agrégation qui opère sur des données multidimensionnelles modélisées en XML. Notre opérateur d'agrégation prend en compte les problèmes d'additivité dus aux hiérarchies complexes afin de reproduire des résultats d'agrégation corrects. Cet opérateur est ensuite étendu à un opérateur de forage vers le haut (*roll-up*) qui prend aussi en compte les problèmes d'additivité dus aux hiérarchies complexes. Dans la Section 4.1, nous présentons la problématique et la motivation des solutions que nous présentons dans ce chapitre. Dans la Section 4.2, nous présentons les stratégies qui nous permettent de traiter les problèmes d'additivité dus aux hiérarchies complexes à l'aide d'un exemple. Dans la Section 4.3, nous définissons notre modèle de données : les arbres de données multidimensionnels. Dans cette définition, nous utilisons les définitions des faits, des dimensions, des mesures, et des hiérarchies, ainsi que les notations de la Section 2.1. Nous présentons ensuite un exemple récurrent d'arbre de données multidimensionnel que nous utilisons dans le reste du chapitre.

Dans la Section 4.4, nous présentons le modèle d'arbre que nous utilisons pour interroger les arbres de données multidimensionnels. Notre modèle d'arbre pour données multidimensionnelles permet de détecter et de corriger les problèmes

d'additivité. Pour atteindre cet objectif, nous associons au modèle d'arbre de base [JLST01] les options de *matching* nécessaires au parcours des hiérarchies complexes, au regroupement et au calcul d'agrégation. Dans la Section 4.5, nous présentons un algorithme de regroupement et d'agrégation. Cet algorithme prend en entrée un arbre de données multidimensionnel et un modèle d'arbre pour données multidimensionnelles. Pour garantir l'additivité, nous y appliquons nos stratégies de traitement des hiérarchies complexes. Ensuite, nous généralisons cet algorithme à un opérateur de forage vers le haut. Nous donnons également un exemple d'opération de forage vers le haut pour illustrer notre approche. Enfin, nous concluons et discutons l'approche présentée dans ce chapitre (Section 4.6).

## 4.1 Problématique et motivation

Certaines situations réelles donnent lieu à des modèles de hiérarchies complexes (hiérarchies géographiques, par exemple [LSZY05]). Ce type de hiérarchies, et les problèmes d'additivité qui en découlent, ne sont pas gérés par les outils OLAP actuels [RALT06]. D'ailleurs, plusieurs auteurs ignorent la modélisation des hiérarchies complexes et la gestion des problèmes d'additivité dans leurs modèles multidimensionnels et leurs opérateurs OLAP. Nous pensons en revanche que la performance d'un outil OLAP tient beaucoup à sa capacité à prendre en compte les situations réelles les plus complexes, à gérer les problèmes d'additivité et à permettre de les résoudre pour aider l'utilisateur à prendre les bonnes décisions.

La représentation des hiérarchies complexes dans les SGBD relationnels classiques s'avère être une opération très compliquée. C'est pourquoi nous nous inspirons du travail de Beyer *et al.* [BCC+05], qui modélisent les hiérarchies complexes en XML et proposons une nouvelle solution basée sur XML pour modéliser et interroger les hiérarchies complexes tout en prenant en compte les problèmes d'additivité dans nos analyses OLAP.

Dans la littérature, il existe trois types de solutions pour les problèmes d'additivité : la normalisation des modèles multidimensionnels (Section 3.3.4), la transformation des données multidimensionnelles (Section 3.3.5) et les solutions en temps réel (Section 3.3.6). Nous ne choisissons pas une solution de normalisation car cette dernière modifie la sémantique des données et leur modélisation initiale. Or, nous pensons que cette modélisation traduit une vision précise du concepteur

de l'entrepôt qu'il est potentiellement dangereux d'altérer, car cela peut impacter les analyses à venir. La transformation préserve, elle, la sémantique des données. Toutefois, cette approche souffre de problèmes de performance. Par exemple, à chaque fois que l'entrepôt de données est mis à jour, toutes les transformations doivent être refaites. De plus, Li *et al.* démontrent que l'algorithme de transformation *MakeCovering* de Pedersen *et al.* [PJD99] n'est pas adapté à certains cas réels liés à la Chine [LSZY05]. En effet, en Chine, certaines provinces contiennent un ou plusieurs districts, mais ces districts n'appartiennent à aucune ville. En d'autres termes, certaines provinces sont elles-même des villes, comme Pékin. *MakeCovering* remplace les valeurs manquantes dans les hiérarchies par des valeurs extraites des métadonnées, mais ne présente aucune solution spéciale pour les cas particuliers comme la hiérarchie géographique de la Chine.

Enfin, les solutions en temps réel existantes se limitent à la détection des problèmes d'additivité, mais ne les traitent pas [PRP02a, PPP04, HS05]. Nous proposons donc d'aller au bout du processus et de calculer les agrégations de manière à garantir l'additivité et à fournir des données en sortie qui puissent être réinterrogées par des requêtes ultérieures. De plus, notre solution ne nécessite aucune expertise de la part de l'utilisateur, contrairement à la normalisation qui nécessite un expert en modélisation multidimensionnelle et à la transformation qui nécessite un expert des données et du système d'information pour résoudre les problèmes d'additivité.

## 4.2 Stratégies utilisées pour garantir l'additivité

Dans cette section, nous présentons les stratégies que nous utilisons pour identifier les hiérarchies complexes au moment de l'analyse, de sorte que l'additivité soit assurée en temps réel. En premier lieu, nous présentons la stratégie que nous utilisons pour la résolution des problèmes d'additivité dans les hiérarchies non-strictes : *les groupes*. Nous présentons ensuite la stratégie que nous utilisons pour la résolution des problèmes d'additivité dans les hiérarchies non-couvrantes : la substitution des valeurs "manquantes" par une valeur artificielle "Autre". Pour résoudre les problèmes d'additivité dans les hiérarchies complexes, nous associons ces deux stratégies.

La Figure 4.1 représente une hiérarchie complexe de catégories de livres, que

nous allons utiliser pour illustrer ces approches. Cette hiérarchie est non-stricte puisqu'une catégorie peut regrouper plusieurs livres. Par exemple, *catégorie 2* regroupe *livre A* et *livre B*. Un livre peut également appartenir à plusieurs catégories, à l'exemple du *livre A* qui appartient aux *catégories 1* et *2*. Cette hiérarchie est aussi non-couvrante puisque *livre D* n'est décrit par aucune catégorie.

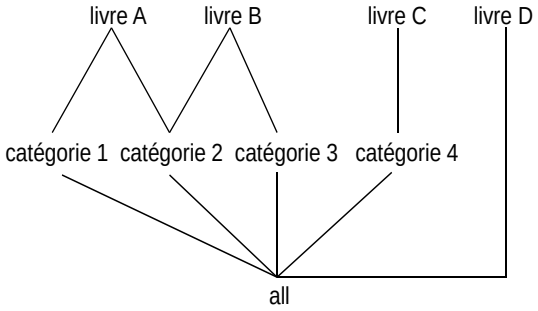



FIGURE 4.1 – Exemple de hiérarchie complexe *livre-catégorie*

### 4.2.1 Résolution des problèmes d'additivité dans les hiérarchies non-strictes

Soit une hiérarchie de dimension non-stricte quelconque. Lorsque nous calculons des valeurs agrégées à un niveau de hiérarchie donné $H_{ij+1}$ à partir des valeurs du niveau $H_{ij}$, nous devons éviter d'agréger plusieurs fois les mesures du niveau $H_{ij}$. Par définition, un groupe $G$ fusionne toutes les valeurs des instances de $H_{ij+1}$ au moment du calcul des valeurs agrégées depuis les valeurs du niveau $H_{ij}$. Formellement, $G = \bigcup_{l \in [1, h_{ij}]} \rho_{ijl}$, où les valeurs multivariées de $\rho_{ijl}$ sont considérées comme des ensembles de valeurs au lieu de valeurs uniques.

Par exemple, à partir de l'arbre de données de la Figure 4.1, supposons que nous voulions déterminer le nombre de livres par catégorie, pour les *livres A* et *B*. Dans une première étape, nous déterminons ce nombre sans prendre en compte le problème d'additivité. Nous avons : *catégorie 1* décrit un livre, *catégorie 2* décrit deux livres et *catégorie 3* décrit un livre. Au total, quatre livres sont décrits dans ces trois catégories. Ce résultat est faux puisque nous ne traitons ici que deux livres.

Effectuons maintenant la même opération en utilisant notre stratégie de construction de groupes. Pour les catégories liées au *livre A*, nous créons le groupe $G_{NS1}$ = [*catégorie 1-catégorie 2*]. Pour les catégories liées au *livre B*, nous créons le groupe $G_{NS2}$ = [*catégorie 2-catégorie 3*]. Dans ce cas, le nombre de livres par catégorie est égal à 2 puisque chaque livre est lié à un groupe de catégorie : [*catégorie 1-catégorie 2*] pour *livre A* et [*catégorie 2-catégorie 3*] pour *livre B*. Chaque groupe de catégorie est en fin de compte une catégorie hybride.

La solution de création de groupes est couramment utilisée dans les bases de données, notamment par Pedersen *et al.* dans leur approche de transformation de données multidimensionnelles [PJD99]. L'algorithme *MakeStrict* permet en effet de fusionner les valeurs des instances d'un même niveau hiérarchique ayant le même élément parent en une seule valeur. Les valeurs fusionnées sont ensuite insérées entre les deux niveaux hiérarchiques parent et enfant pour calculer les agrégations sans se soucier des problèmes d'additivité (Section 3.3.5). Le principe de construction des groupes est le même que celui dans l'algorithme *MakeStrict*, décrit ci-dessus. La différence entre Pedersen *et al.* et nous est dans la manière d'insérer les groupes dans l'arbre de données. Dans notre solution, nous remplaçons toutes les valeurs d'un niveau hiérarchique donné d'une hiérarchie non-stricte par le groupe $G$.

### 4.2.2 Résolution des problèmes d'additivité dans les hiérarchies non-couvrantes

Pour traiter les hiérarchies non-couvrantes, nous regroupons toutes les valeurs manquantes de $H_{ij+1}$ au moment du calcul des valeurs agrégées depuis les valeurs du niveau $H_{ij}$ dans une seule valeur artificielle "Autre". Formellement, $G = \bigcup_{l \in [1,h_{ij}]} \rho_{ijl} \cup \{Other\}$ such that $\exists l' / \rho_{ijl} = \sigma_{i(j+1)l'}$. Cette solution est déjà utilisée dans le domaine des bases de données en général, et dans la solution de Mansmann et Scholl pour les problèmes d'additivité en particulier [MS06, MS07]. Comme dans notre approche, Mansmann et Scholl regroupent toutes les valeurs manquantes d'un niveau hiérarchique donné dans une seule valeur artificielle "Others".

Supposons maintenant que nous voulions déterminer le nombre de livres par catégorie pour les livres $C$ et $D$ de la Figure 4.1. Le *livre D* n'appartient à aucune

catégorie. La hiérarchie *livre-catégorie* est donc non-couvrante. Nous créons alors dans cette hiérarchie une catégorie artificielle "Autre" et nous l'associons au *livre D*. Dans ce cas, le nombre de livres par catégorie est égal à deux, puisque nous avons une catégorie par livre : *catégorie 4* pour *livre C* et *catégorie "Autre"* pour *livre D*. Le total (deux livres) est bien correct. Si nous n'avions pas créé la catégorie artificielle "Autre", nous aurions obtenu un résultat total à un livre, puisque seule *catégorie 4* aurait été prise en compte.

### 4.2.3 Résolution des problèmes d'additivité dans les hiérarchies complexes

Dans le but de résoudre les problèmes d'additivité dans les hiérarchies complexes, nous associons les solutions de traitement des hiérarchies non-strictes et des hiérarchies non-couvrantes en une seule solution. L'idée est ici de regrouper toutes les valeurs d'un niveau hiérarchique donné dans un seul groupe, comme nous avons fait pour résoudre les problèmes d'additivité dans les hiérarchies non-strictes. Dans ce groupe, les instances "manquantes" des hiérarchies non-couvrantes sont toutes remplacées par la valeur artificielle "Autre" : $G = \bigcup_{l \in [1,h_{ij}]} \rho_{ijl} \cup \{Autre\}$ tel que $\exists L / \rho_{ijl} = \bigcup_{l' \in L} \sigma_{i(j+1)l'}$.

Supposons maintenant que nous voulions déterminer le nombre de livres par catégorie pour tous les livres de la Figure 4.1. Pour *livre A*, nous créons la catégorie [*catégorie 1-catégorie 2*]. Pour *livre B*, nous nous créons la catégorie [*catégorie 2-catégorie 3*]. Pour *livre C*, nous avons la catégorie *catégorie 4*. Et pour *livre D*, nous créons la catégorie artificielle "Autre". Nous obtenons donc un livre par catégorie et quatre livres au total, ce qui correspond à un résultat correct.

## 4.3 Arbres de données multidimensionnels

Au niveau logique, nous profitons de la simplicité des arbres de données et de leur capacité à représenter des situations complexes du monde réel pour modéliser les données multidimensionnelles et les hiérarchies complexes [HD10a, HD10b]. Nous baptisons ces arbres de données arbres de données multidimensionnels. Dans la Section 4.3.1, nous formalisons les arbres de données multidimensionnels. Au-

paravant, nous présentons nos trois motivations derrière ce choix.

En premier lieu, nous choisissons de représenter les données multidimensionnelles en XML au niveau physique. La majorité des auteurs qui proposent des algèbres XML modélisent les données XML, au niveau logique, à l'aide d'arbres de données [JLST01, PAKJ+02, BFLC03, ZY04, Pra06]. Les arbres de données sont aussi utilisés dans d'autres types d'approches XML, notamment les approches holistiques [BKS02, LCL04, LLY+05, LLCC05, CLT+06, Lu06, BKS08, LLBW11]. La motivation principale des auteurs de toutes ces approches est la facilité de modéliser des situations réelles à l'aide d'arbres de données. Deuxièmement, plusieurs auteurs ont également choisi de modéliser les données multidimensionnelles dans des documents XML [PHS05, WLHG05, BBLR06, WJLS07]. Beyer *et al.* profitent de la flexibilité de XML pour représenter et regrouper des données difficiles à modéliser et à regrouper en relationnel [BCC+05]. De plus, ils affirment que les hiérarchies complexes sont facilement représentées en XML. Par exemple, il est facile de représenter un livre écrit par deux auteurs (hiérarchie non-stricte) ou un livre dont on ne connait pas l'éditeur (hiérarchie non-couvrante) dans un seul fichier XML.

Notre troisième motivation est liée à certaines approches qui ont présenté des modèles spécifiques pour les hiérarchies complexes. Nous classons ces travaux en deux familles. Dans la première famille, les auteurs présentent des modèles conceptuels entité-association pour modéliser les hiérarchies complexes [MZ04, MZ06, ASS06]. Au niveau logique, les données sont modélisées en relationnel dans ces approches. Dans la deuxième famille, Hurtado *et al.* et Pedersen *et al.* présentent des formalismes spécifiques, inspirés des graphes, pour la modélisation des hiérarchies complexes [HGM05, PJD99]. À l'aide de ces formalismes spécifiques, les hiérarchies complexes sont modélisées de manière arborescente (Figures 3.18(a) et 3.20). Ce sont les travaux de la deuxième famille qui nous intéressent.

En effet, l'avantage du modèle logique de Hurtado *et al.* est qu'il permet d'intégrer, par exemple, plusieurs dimensions hétérogènes et de même type dans un graphe (des hiérarchies géographiques, par exemple). La représentation de différentes dimensions géographiques dans le même graphe permet de mettre en évidence, d'une façon visuelle et facile, leurs différences. Par exemple, le graphe de la Figure 3.18(a) permet de représenter des sites géographiques français, canadiens et américains (donc différents) dans le même schéma, de manière très simple. De

même, le modèle logique de Pedersen *et al.* modélise les hiérarchies complexes de manière très simple (Figure 3.20). Le deuxième avantage de ces modèles logiques est qu'ils permettent de mettre en œuvre les solutions de leurs auteurs pour les problèmes d'additivité.

### 4.3.1 Métamodèle

La Figure 4.2 illustre la façon dont nous modélisons, au niveau logique, les arbres de données multidimensionnels. Le symbole $*$ associé aux différentes arêtes de notre modèle autorise la modélisation des relations de type un à plusieurs. La racine $W$ modélise l'entrepôt de données. Les nœuds enfants $F$ de $W$ modélisent les faits. Chaque fait est décrit par une ou plusieurs dimensions $D$ et par une ou plusieurs mesures $M$. Une hiérarchie de dimension peut avoir un nombre quelconque de niveaux hiérarchiques $H$. Nous représentons les hiérarchies de dimensions $H$ de façon récursive pour que notre modèle de données autorise la modélisation des hiérarchies complexes. En fait, le symbole $*$ associé à l'arête qui relie $D$ et $H$ autorise les relations entre les faits et tout niveau hiérarchique, peu importe son niveau de granularité (relations entre faits et dimensions). Par ailleurs, l'arête associée à $H$ et représentée de façon récursive permet de relier un niveau hiérarchique $H_{ij}$ à un niveau $H_{ij'}$ en "sautant" un ou plusieurs niveaux intermédiaires (relations intradimensionelles).

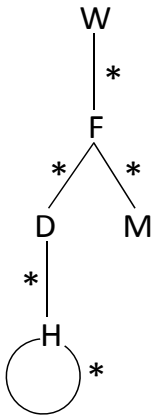



FIGURE 4.2 – Métamodèle d'arbres de données multidimensionnels

### 4.3.2 Exemple d'instanciation du métamodèle

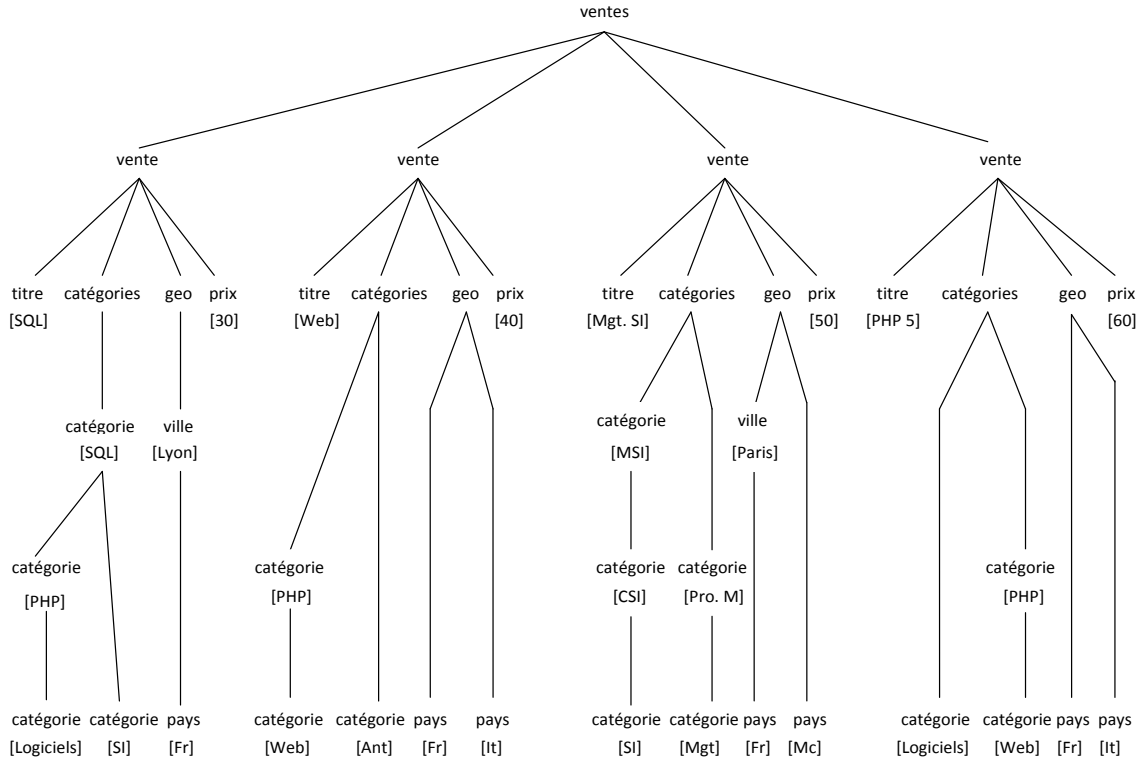



FIGURE 4.3 – Exemple d'arbre de données multidimensionnel

Dans la Figure 4.3, nous modélisons les données multidimensionnelles de l'entrepôt de ventes de la Figure 2.5 à l'aide d'un arbre de données multidimensionnel (conformément à notre métamodèle de la Figure 4.2). Dans cet exemple :

- la racine *ventes* de l'arbre de données multidimensionnel modélise l'entrepôt *ventes* ($W$ dans le métamodèle de la Figure 4.2) ;
- nous avons quatre faits *ventes* ($F$ dans le métamodèle de la Figure 4.2) ;
- pour chaque fait *vente*, nous avons trois dimensions ($D$ dans le métamodèle de la Figure 4.2) et une mesure ($M$ dans le métamodèle de la Figure 4.2) ;
- les dimensions sont livre (identifié par son *titre*), *catégories* et site géographique (*geo*). La mesure est *prix* ;
- les dimensions *catégories* et *geo* sont constituées de hiérarchies ($H$ dans le métamodèle de la Figure 4.2).

*Catégories* est une hiérarchie non-stricte puisqu'un livre peut appartenir à plusieurs catégories et que chaque catégorie peut contenir plusieurs livres. Par exemple, dans le cas du premier livre intitulé `SQL`, la catégorie `SQL` fait partie de deux catégories de niveau supérieur, `Logiciels` et `SI`. La catégorie `Logiciels` décrit, elle, plusieurs livres : `SQL` et `PHP 5`.

Dans l'exemple de la Figure 2.5, la hiérarchie de dimension *catégories* est modélisée de façon récursive, ce qui signifie que le nombre de niveaux hiérarchiques de *catégories* est indéfini. Faute d'espace, dans la Figure 4.3, nous limitons le nombre maximum de niveaux hiérarchiques *catégorie* à trois pour chaque fait *vente*. Les dimensions *catégorie* sont représentées de manière à mettre en évidence les catégories manquantes dans le cas de hiérarchies non-couvrantes. Nous appelons la catégorie la plus générale catégorie de niveau 1 (`SQL, MSI`) et la catégorie la plus détaillée catégorie de niveau 3 (`Logiciels, SI, Web, Ant, SI, Mgt`). Pour le premier fait *vente* (premier fait à gauche), la catégorie de niveau 2 est inconnue pour les catégories `SQL`. Pour le deuxième fait *vente*, les catégories de niveaux 1 et 2 sont inconnues pour la catégorie `Ant`. Cette hiérarchie est donc non-couvrante. La hiérarchie de dimension *catégorie* est donc une hiérarchie complexe puisqu'elle est à la fois non-stricte et non-couvrante.

Maintenant, soit la hiérarchie géographique *geo*. Le deuxième livre `Web` est vendu dans deux sites géographiques différents, la France (`pays[Fr]`) et l'Italie (`pays[It]`). De plus, dans un site géographique donné, plusieurs livres sont vendus, à l'exemple de la France (`Fr`). Donc, *geo* est une hiérarchie non-stricte. De plus, certains éléments sont inconnus dans certaines instances de la hiérarchie géographique. Par exemple, nous ne connaissons ni les villes ni les régions dans lesquelles est vendu le livre `PHP 5`. Cette hiérarchie est donc non-couvrante. Non-stricte et non-couvrante à la fois, *geo* est une hiérarchie complexe.

## 4.4 Modèle d'arbre pour données multidimensionnelles

Une fois modélisées en XML, les données multidimensionnelles sont prêtes à l'analyse. Par conséquent, les requêtes analytiques sur ces données multidimen-

sionnelles peuvent être exprimées avec les langages de requêtes XML. Ces langages doivent permettre à l'utilisateur de formuler les requêtes nécessaires pour (1) naviguer entre les différents éléments XML organisés en hiérarchies et (2) effectuer les regroupements et les agrégations nécessaires à l'analyse. Pour atteindre cet objectif, les langages de requêtes XML doivent inclure des fonctions de regroupement et d'agrégation.

XQuery [1], le langage de requêtes XML standard, ne permet pas de réaliser facilement des regroupements multiples, et par la suite ne prend pas en compte la structure complexe des données décisionnelles (les hiérarchies complexes) [BCC+05]. De plus, certains regroupements simples avec XQuery donnent de faux résultats en présence de hiérarchies complexes [BCC+05].

Par exemple, soit la requête $Q_1$ = "prix des livres pour chaque auteur" exprimée en XQuery. Le résultat de cette requête sur un fichier XML qui modélise des livres écrits par plus d'un auteur affiche le nom de chaque auteur avec le prix moyen des tous ses livres, mais ne prend pas en compte le fait qu'il existe des livres écrits par plus d'un auteur. Par exemple, soit le livre "Transaction Processing" écrit par Jim Gray et Andreas Reuters et modélisé en XML. Le résultat de $Q_1$ exprimée en XQuery retourne ce livre avec un auteur différent à chaque fois : Jim Gray et Andreas Reuters, en plus du prix du livre. Or, ce résultat est incorrect puisque "Transaction Processing" est un seul livre écrit par Gray et Reuters et non deux livres de même titre et de même prix écrits par Gray et Reuters, respectivement.

Pour résoudre le problème du regroupement dans XQuery, certains auteurs étendent la syntaxe de XQuery avec une fonction `group by` [BC04, DPX04, BCC+05, GRS06, KAK09]. En plus du `group by`, Beyer *et al.* étendent le langage XQuery par des fonctions de forage vers le haut et de calcul de cube [BCC+05]. Pour réaliser ces opérations, ils définissent deux fonctions, `paths` et `cube`, qui ne sont pas faciles à écrire. De plus, chacune de ces fonctions nécessite l'appel d'autres fonctions définies dans le travail de Beyer et Ramakrishnan [BR99], ce qui rend cette solution difficile à utiliser.

Dans notre solution pour les problèmes d'additivité dans les hiérarchies complexes, nous avons décidé de présenter un outil de requêtage simple à utiliser. Pour cela, comme nous utilisons les arbres de données multidimensionnels (Section

1. `http://www.w3.org/TR/xquery/`

4.3) pour modéliser les données multidimensionnelles et les hiérarchies complexes, nous utilisons les modèles d'arbres pour concevoir les requêtes décisionnelles. Or, nous avons conclu après l'étude des différents modèles d'arbres présentés dans la littérature (Section 3.4) qu'il n'existe pas de modèle d'arbre pour données multidimensionnelles, ni de modèle d'arbre qui réponde à nos besoins. Nous avons donc adapté le modèle d'arbre de base [JLST01] en lui ajoutant des options de *matching*.

Notre modèle d'arbre pour données multidimensionnelles est présenté dans la Figure 4.4.

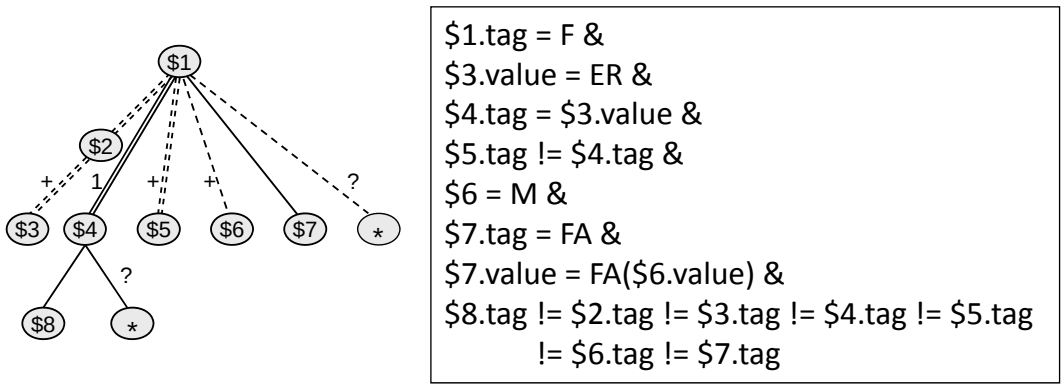



FIGURE 4.4 – Modèle d'arbre pour données multidimensionnelles

Dans ce modèle d'arbre, les nœuds reliés à leur nœud parent [2] par des arêtes continues apparaissent dans l'arbre résultat. Par contre, les nœuds reliés à leur nœud parent par des arêtes discontinues n'apparaissent pas dans l'arbre résultat. Nous associons les annotations suivantes à certaines arêtes de notre modèle d'arbre.

- ? signifie que zéro à un *matching* est autorisé.
- + signifie que un à plusieurs *matchings* sont autorisés.
- 1 signifie que un et un seul *matching* est autorisé.

Les nœuds de notre modèle d'arbre sont étiquetés $\$i$ ou par un astérisque $*$. La racine est étiquetée \$1 et ses nœuds descendants \$2, \$3, etc. en largeur d'abord. Dans notre modèle d'arbre, tous les nœuds marqués avec le symbole $*$ sont reliés à leur nœud parent par des relations de types PE (/) dans la requête. En effet,

2. Les relations de types père-enfant (PE) et ancêtre-descendant (AD) sont définies dans la Section 3.4.1.

dans XPath 2.0 [3], l'expression de chemin $x//*$ retourne un résultat différent de l'expression de chemin $x/*$ (avec $x$ un nœud quelconque). En fait, $x/*$ retourne le résultat correct du *matching* entre le nœud $x$ et les nœuds de même type que $x$ dans un arbre de données. $x//*$ retourne le même résultat que $x/*$ avec des redondances. Nous choisissons donc de respecter le standard XPath 2.0 dans notre modèle d'arbre. Les nœuds de notre modèle d'arbre, repris dans la formule située à gauche dans la Figure 4.4, sont décrits comme suit.

- Le nœud \$1 sélectionne un fait.
- Le nœud \$3 spécifie un ou plusieurs éléments de regroupement ($ER$ dans la formule associée à notre modèle d'arbre). Un critère de regroupement correspond à une dimension.
- Le nœud \$2 modélise tous les nœuds qui peuvent exister entre \$1 et \$3. Par exemple, si le nœud \$3 correspond à la dimension *pays*, alors \$2 prend comme valeur tous les éléments de la hiérarchie géographique de niveau inférieur à *pays* (comme *ville*, *région* et *geo* dans l'exemple de la Figure 4.3). C'est pourquoi les couples de nœuds (\$1, \$2) et (\$2, \$3) sont reliés avec des relations de type AD (ancêtre-descendant) dans notre modèle d'arbre.
- L'étiquette du nœud \$4 prend comme valeur le contenu du nœud \$3. Le nœud \$4 prend comme valeur le contenu d'un groupe $G$. Le nœud \$4 sélectionne un et un seul exemplaire de chaque critère de regroupement dans l'arbre de donnés multidimensionnel. Par exemple, pour un élément de regroupement appartenant à une hiérarchie non-stricte (*pays* dans la hiérarchie *geo* de la Figure 4.3, par exemple), \$4 sélectionne un seul exemplaire de ce nœud. La valeur de ce nœud prend le groupe construit pour ce critère de regroupement (comme le groupe [Fr-It], par exemple).
- Le nœud \$5 sélectionne tous les nœuds des dimensions de l'arbre de données qui sont différents du nœud \$4. Le but de cette opération est d'identifier toutes les dimensions qui ne font pas partie des éléments de regroupement. Ces nœuds n'apparaissent pas dans l'arbre résultat puisque le nœud \$5 est relié au nœud \$1 par une arête discontinue.
- Le nœud \$6 spécifie la (ou les) mesure(s) à agréger.
- Le nœud \$7 stocke le résultat de l'application d'une fonction d'agrégation *FA* (somme, moyenne, etc.) sur la (ou les) mesure(s) spécifiée(s) par le nœud \$6.

3. http ://www.w3.org/TR/xpath20/

- Le *matching* entre \$4/∗ et l'arbre de données permet de sélectionner la hiérarchie reliée au nœud \$4. Cette hiérarchie est conservée dans le résultat. Or, ce *matching* entre \$4/∗ et l'arbre de données peut ne pas retourner tous les nœuds requis pour cette opération. Ceci est dû à la présence des hiérarchies non-couvrantes. Par exemple, si le critère de regroupement est *ville*, l'expression *ville*/∗ ne sélectionne aucune ville pour le quatrième fait de la Figure 4.3 (livre intitulé `[PHP 5]`) puisqu'elle ne trouve pas de nœud *ville*. La hiérarchie qui est normalement liée à ville (`pays [Fr]` et `[It]` dans ce cas) ne sera donc pas prise en compte dans le résultat. Pour pallier ce problème et sauvegarder cette hiérarchie, le nœud \$8 stocke le résultat du *matching* de \$1/∗ avec l'arbre de données, à l'exception des mesures qui ne sont pas prises en compte par la fonction d'agrégation *FA*.

## 4.5 Algorithmes

Dans cette section, nous présentons les algorithmes que nous utilisons pour résoudre les problèmes d'additivité dans les hiérarchies complexes. Nous présentons en premier lieu un algorithme de regroupement et d'agrégation (Section 4.5.1). Nous généralisons ensuite cet algorithme à un opérateur de forage vers le haut (Section 4.5.2). Enfin, nous donnons un exemple d'une opération de forage vers le haut sur les données de la Figure 4.3.

### 4.5.1 Algorithme de regroupement et d'agrégation

Pour résoudre les problèmes d'additivité dans les hiérarchies complexes, nous exploitons notre modèle d'arbre pour données multidimensionnelles (Section 4.4) et les stratégies de traitement des hiérarchies complexes (Section 4.2) dans l'Algorithme 1. Cet algorithme calcule des agrégations en fonction d'un ou de plusieurs critères de regroupement. En même temps que le regroupement et le calcul d'agrégation, l'Algorithme 1 détecte les problèmes d'additivité et les traite en temps réel lors du *matching* entre le modèle d'arbre et un arbre de données multidimensionnel. Lorsque l'algorithme rencontre des hiérarchies non-strictes et/ou des hiérarchies non-couvrantes, il fait appel aux stratégies de traitement des hiérarchies complexes.

**Algorithme 1** Regroupement et d'agrégation

**Entrées :** $D$ //arbre de données multidimensionnel, $TPQ$ //modèle d'arbre
**Sortie :** $WTlist$ //ensemble d'arbres résultats
$WTlist \leftarrow \emptyset$
**pour tout** \$1 **faire**
  // **Étape #1** : traitement de l'additivité
  $Group_list \leftarrow \emptyset$
  **pour tout** \$4 **faire**
    $Group \leftarrow Group \cup$ \$4.value
    **si** (\$4 $\notin$ \$1.`children()` et $Group$.`nbElements()` $<$ \$1.`currentChild().nbChildren()`) **alors**
      $Group \leftarrow Group \cup$ "Autre"
    **fin si**
    $Group_list \leftarrow Group_list \cup Group$
  **fin pour**
  // **Étape #2** : *matching* des groupes
  $WT \leftarrow WTlist$.`exists(`$Group_list$`)`
  **si** $WT \neq \emptyset$ **alors**
    $WT$.`update(`\$6, \$7`)`
  **sinon**
    $WT$.`create(`$D$, $TPQ$`)`
    $WTlist \leftarrow WTlist \cup WT$
  **fin si**
**fin pour**
**retourner** `product(`$WTlist$`)`

L'Algorithme 1 prend en entrée un arbre de données multidimensionnel $D$ et un modèle d'arbre $TPQ$. Il retourne un à plusieurs arbres résultats réunis dans une liste d'arbres $WTlist$. Il est essentiellement composé de deux étapes : (1) traitement de l'additivité et (2) *matching* des groupes (c'est-à-dire la vérification de l'existence de groupes dans le résultat final).

### Paramètres et variables

Dans ce qui suit, nous donnons la description des variables utilisés dans l'Algorithme 1.

- $D$ est un arbre de données multidimensionnel modélisé selon le métamodèle de la Section 4.3.1.
- $TPQ$ est un modèle d'arbre qui modélise la requête utilisateur grâce au modèle de requêtes défini à la Section 4.4.
- $WT$ est un arbre résultant du *matching* de $D$ et $TPQ$.
- $WTlist$ est une liste d'arbres utilisée pour stocker tous les arbres résultats $WT$.

- *Group* est un groupe construit comme défini à la Section 4.2.
- *Group_list* est une liste de tous les groupes *Group* construits pour un fait donné.

**Fonctions**

Dans ce qui suit, nous donnons la description des fonctions utilisés dans l'Algorithme 1.

- $x$.children() retourne la liste d'enfants du nœuds $x$. Plus précisément, elle retourne le nombre d'arêtes enracinées en $x$.
- $G$.nbElements() retourne le nombre d'éléments d'un groupe $G$ donné.
- $x$.currentChild() retourne l'enfant [4] courant du nœud $x$. Un nœud courant est un nœud en cours de traitement ou bien l'un de ses nœuds descendants en cours de traitement.
- $x$.nbChildren() retourne le nombre de nœuds enfants du nœud $x$.
- *Tlist*.exists(*Glist*) retourne l'arbre de données contenant la liste de groupes *Glist* parmi la liste d'arbres de données *Tlist*, s'il existe. Sinon, elle retourne $\emptyset$.
- $T$.update($x$, $y$) permet de mettre à jour la valeur du nœud $y$ avec la valeur du noeud $x$.
- $T$.create($D$, $TPQ$) permet de créer un arbre $T$ suite au *matching* entre le modèle d'arbre $TPQ$ et l'arbre de données $D$.
- product(*Tlist*) permet de réunir tous les arbres de données de *Tlist* sous une seule racine.

**Étape 1 : Traitement de l'additivité**

*WTlist* est en premier lieu initialisée à vide. Ensuite, pour chaque fait (sélectionné par le nœud \$1 de $TPQ$), la liste de groupes *Group_list*, qui stocke tous les groupes *Group* possibles pour chaque critère de regroupement, est aussi initialisée à vide. Un groupe est déterminé suite au *matching* entre le nœud \$4 de $TPQ$ et les nœuds de même type que \$4 dans $D$. Si la condition (\$4 $\notin$ \$1.`children()` et *Group*.`nbElements()` $<$ \$1.`currentChild().nbChildren()`) est vraie, alors la

4. Les relations de types père-enfant (PE) et ancêtre-descendant (AD) sont définies dans la Section 3.4.1.

hiérarchie en cours de traitement est non-couvrante. Cette condition signifie que si le nombre d'éléments d'un groupe est inférieur au nombre de hiérarchies qui contiennent, normalement, le nœud élément de regroupement, c'est qu'il existe une à plusieurs instances de cet élément manquantes dans cette hiérarchie. Dans ce cas, la valeur "Autre" est concaténée à la valeur du groupe. À la fin de la construction de chaque groupe, ce dernier est ajouté à *Group_list*.

#### Étape 2 : *Matching* des groupes

L'Algorithme 1 teste ensuite l'existence de *Group_list* dans *WTlist*. En d'autres termes, il teste s'il existe un arbre résultat dans *WTlist* contenant tous les éléments de *Group_list* par une simple opération de *matching* entre les éléments de *Group_list* et les arbres résultats de *WTlist*. Si ce *matching* est positif pour un arbre résultat, le nœud correspondant à la fonction d'agrégation ($7) dans cet arbre est mis à jour par la mesure en cours ($6). Dans le cas contraire, un nouvel arbre résultat ( $WT$) est créé dans *WTlist* en respectant les spécifications de *TPQ*. Ce traitement est répété pour chaque fait. Une fois tous les faits traités, les arbres résultats de *WTlist* sont tous réunis sous un seul nœud racine à l'aide de la fonction `product()`.

### 4.5.2 Forage vers le haut

Par définition, le forage vers le haut (*roll-up*) consiste à analyser les données en fonction d'un niveau de granularité moins détaillé [Tes09]. Le but du forage vers le haut est de passer d'un niveau de granularité donné à un deuxième niveau de granularité plus général (par exemple, passer d'une représentation multidimensionnelle des données suivant les villes à une autre représentation suivant les pays). Ce passage nécessite l'association d'une fonction d'agrégation (somme, moyenne, etc.) à l'opération de regroupement pour spécifier comment sont calculées les valeurs du niveau supérieur à partir de celles du niveau inférieur.

L'opérateur de forage vers le haut sur les arbres de données multidimensionnels que nous proposons respecte cette définition. Son fonctionnement est décrit dans la Figure 4.5.

Notre opérateur de forage vers le haut appelle, d'une façon séquentielle, l'Algorithme 1 $n$ fois ($n$ étant un entier strictement supérieur à 1), comme le montre la Figure 4.5. Dans une opération de forage vers le haut, l'arbre résultat (les arbres

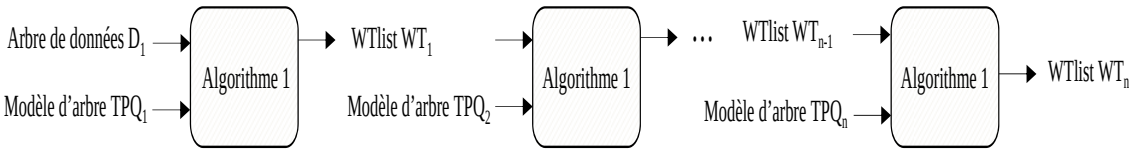



FIGURE 4.5 – Processus de forage vers le haut

de *WTlist* regroupés sous une seule racine) d'une étape donnée sert comme d'arbre de données en entrée de l'étape suivante.

### Exemple de forage vers le haut

Soit l'opération de forage vers le haut suivante : "somme totale des ventes par région et par catégorie de niveau 2" [5] sur l'arbre de données de la Figure 4.3. Cette opération nécessite les deux requêtes $Q_1$ et $Q_2$ suivantes.

- $Q_1$ = "somme totale des ventes par ville et par catégorie de niveau 1". $Q_1$ prend en entrée l'arbre de données de la Figure 4.3 et le modèle d'arbre de la Figure 4.4, auquel nous associons la formule de la Figure 4.6.

```
$1.tag =  vente &
$3.value = {ville, catégorie} &
$4.tag = $3.value &
$5.tag != $4.tag &
$6 = prix &
$7.tag = somme &
$7.value = somme($6.value) &
$8.tag != $2.tag != $3.tag != $4.tag
        != $5.tag != $6.tag != $7.tag
```

FIGURE 4.6 – Formule associée à la requête $Q_1$

- $Q_2$ = "somme totale des ventes par région et par catégorie de niveau 2". $Q_2$ prend en entrée l'arbre résultat de la requête $Q_1$ et le modèle d'arbre de la Figure 4.4, auquel nous associons la formule de la Figure 4.7.

5. Les dimensions *catégorie* sont représentées de manière à mettre en évidence les catégories manquantes dans le cas des hiérarchies non-couvrantes (cf. Section 4.3.2).

$1.tag = vente &
$3.value = {région, catégorie} &
$4.tag = $3.value &
$5.tag != $4.tag &
$6 = prix &
$7.tag = somme &
$7.value = somme($6.value) &
$8.tag != $2.tag != $3.tag != $4.tag
!= $5.tag != $6.tag != $7.tag

FIGURE 4.7 – Formule associée à la requête $Q_2$

**Exécution de la requête $Q_1$.** À partir du premier fait *vente* (premier nœud *vente* à gauche) de l'arbre de données de la Figure 4.3, l'Algorithme 1 construit les deux groupes `catégorie[SQL]` et `ville[Lyon]`, qui correspondent aux critères de regroupement *catégorie* et *ville*, respectivement. Il crée un premier arbre résultat avec les nœuds de dimensions `catégorie[SQL]` et `ville[Lyon]`, et le nœud d'agrégation `somme[30]`. Les hiérarchies liées aux nœuds `catégorie[SQL]` et `ville[Lyon]` sont conservées dans l'arbre résultat, comme le montre la Figure 4.8.

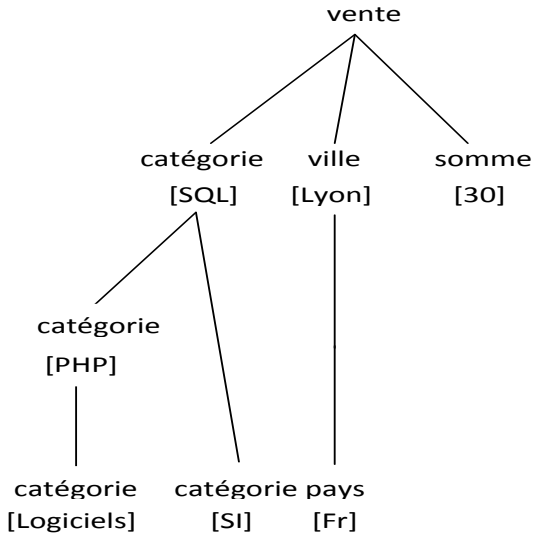



FIGURE 4.8 – Requête $Q_1$ : étape 1

À partir du deuxième fait *vente*, l'Algorithme 1 construit les deux groupes `catégorie[Autre]` et `ville[Autre]` et vérifie l'existence de `Group_list =`

`[catégorie[Autre], ville[Autre]]` dans *WTlist*. En d'autres termes, il vérifie s'il existe un arbre résultat qui comporte les deux nœuds `catégorie[Autre]` et `ville[Autre]` dans *WTlist*. Ce n'est pas le cas ici. Un deuxième arbre résultat est donc créé avec les nœuds de dimensions `catégorie[Autre]` et `ville[Autre]`, et le nœud d'agrégation `somme[40]`. Les hiérarchies liées aux nœuds qui ont formé `catégorie[Autre]` et `ville[Autre]` sont également conservées dans l'arbre résultat, comme le montre la Figure 4.9.

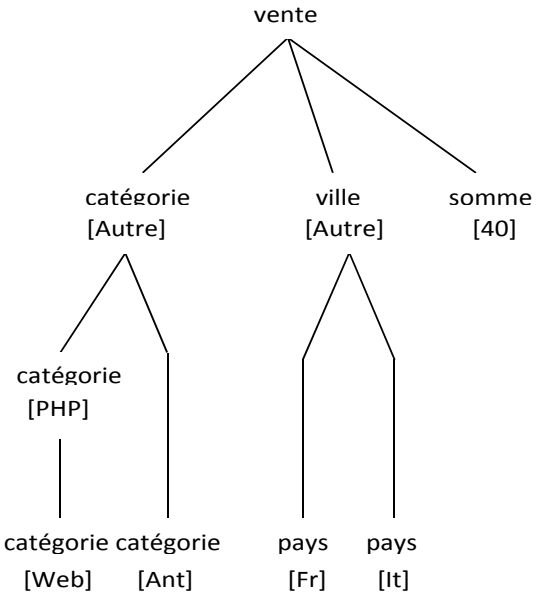



FIGURE 4.9 – Requête $Q_1$ : étape 2

À partir du troisième fait *vente*, l'Algorithme 1 construit les deux groupes `catégorie[MSI-Autre]` et `ville[Paris-Autre]`. L'Algorithme 1 vérifie ensuite si `Group_list = [catégorie[MSI-Autre], ville[Autre]]` existe dans *WTlist*. `Group_list = [catégorie[MSI-Autre], ville[Autre]]` n'existe pas dans les deux arbres résultats déjà créés (Figures 4.8 et 4.9). Un troisième arbre résultat est donc créé avec les nœuds de dimensions `catégorie[MSI-Autre]` et `ville[Paris-Autre]`, et le nœud d'agrégation `somme[50]`. Comme pour les deux arbres résultats précédents, les hiérarchies liées aux nœuds `catégorie[MSI-Autre]` et `ville[Autre]` sont conservées dans l'arbre résultat, comme le montre la Figure 4.10.

À partir du dernier fait *vente*, l'Algorithme 1 construit les deux groupes

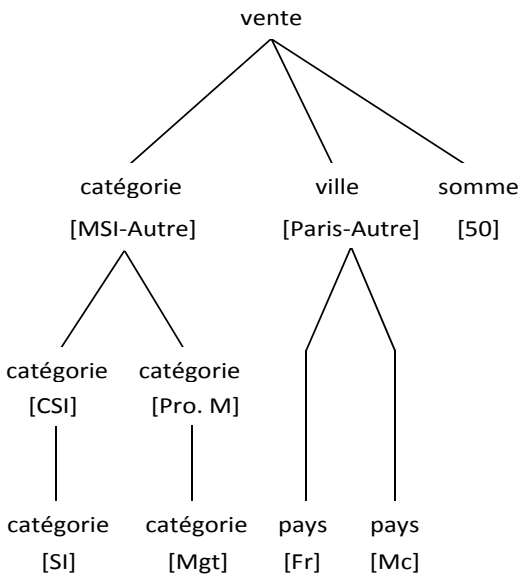



FIGURE 4.10 – Requête $Q_1$ : étape 3

`catégorie[Autre]` et `ville[Autre]`. Et comme `Group_list = [catégorie[Autre], ville[Autre]]` existe dans l'arbre résultat de la Figure 4.9, l'Algorithme 1 met à jour le nœud *somme* avec la valeur du nœud prix ($60 + 40 = 100$). Ensuite, il ajoute le nœud `catégorie[Logiciels]` à la hiérarchie du nœud `catégorie[Autre]`. Le nœud `catégorie[Web]` existe déjà dans le deuxième arbre résultat. Les nœuds `pays[Fr]` et `pays[It]` existent également dans le deuxième arbre résultat, comme le montre la Figure 4.11.

Enfin, les trois arbres résultats des Figures 4.8, 4.10 et 4.11 sont réunis sous une seule racine *ventes*, comme le montre la Figure 4.12.

**Exécution de la requête $Q_2$.** À partir du premier fait *vente* de la Figure 4.12, l'Algorithme 1 construit les deux groupes `catégorie[PHP-Autre]` et `région[Autre]`, qui correspondent aux critères de regroupement catégorie et région, respectivement. Ensuite, il crée un premier arbre résultat avec les nœuds dimensions `catégorie[PHP-Autre]` et `région[Autre]`, et le nœud d'agrégation `somme[30]` dans *WTlist*. Les hiérarchies liées aux nœuds qui ont formé `catégorie[PHP-Autre]` et `région[Autre]` sont conservées dans l'arbre résultat, comme le montre la Fi-

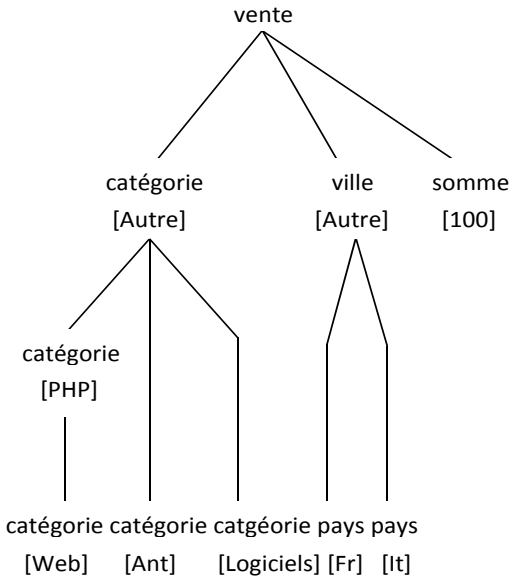



FIGURE 4.11 – Requête $Q_1$ : étape 4

gure 4.13.

À partir du deuxième fait *vente*, l'Algorithme 1 construit les deux groupes `catégorie[PHP-Autre]` et `région[Autre]`, qui correspondent aux deux critères de regroupement catégorie et région, respectivement. Les deux nœuds qui forment `Group_list = [catégorie[PHP-Autre], région[Autre]]` existent dans le premier arbre résultat de *WTlist* (Figure 4.13). Dans ce cas, l'Algorithme 1 met à jour le nœud *somme* du premier arbre résultat avec la mesure en cours (100 + 30 = 130). De plus, il ajoute les nœuds `catégorie[Web]` et `catégorie[Ant]` à la hiérarchie du nœud `catégorie[PHP-Autre]`. La hiérarchie du nœud `région[Autre]` est également mise à jour avec le nœud `pays[It]`, comme le montre la Figure 4.14.

À partir du troisième fait *vente*, l'Algorithme 1 construit les deux groupes `catégorie[CSI-Pro. M]` et `région[Autre]`. Comme les éléments de `Group_list = [C2[CSI-Pro. M], région[Autre]]` n'existent pas dans le seul arbre résultat de *WTlist* (Figure 4.14), l'Algorithme 1 crée un deuxième arbre résultat dans *WTlist* avec les dimensions `catégorie[CSI-Pro. M]` et `région[Autre]`, et l'agrégation `somme[50]` (Figure 4.15).

Enfin, les arbres résultats (Figures 4.14 et 4.15) sont réunis sous une seule

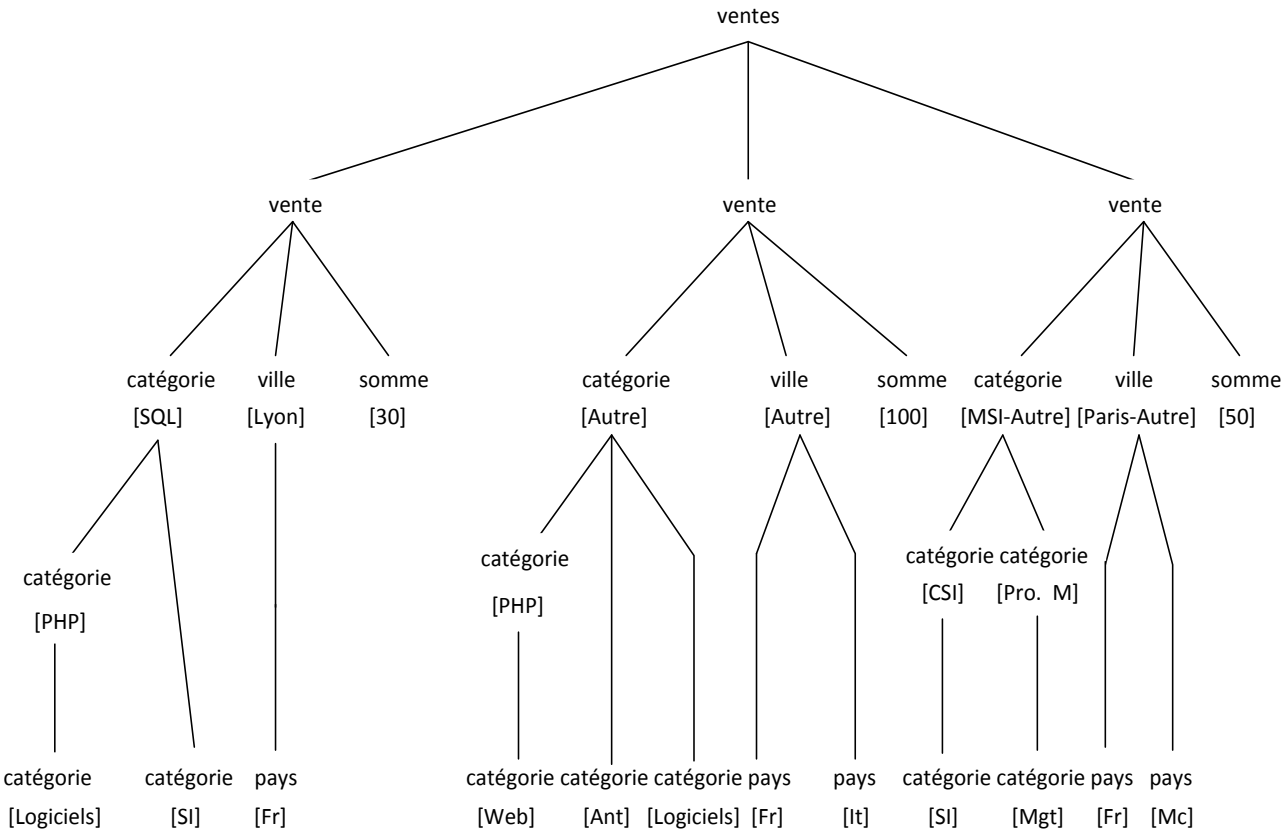



FIGURE 4.12 – Résultat de la requête $Q_1$

racine *ventes* comme le montre la Figure 4.16.

## 4.6 Discussion

La majorité des solutions pour les problèmes d'additivité qui existent dans la littérature sont des approches de normalisation (Section 3.3.4) ou de transformation (Section 3.3.5). Par contre, notre solution est une solution en temps réel. La différence avec les solutions en temps réel existantes (Section 3.3.6) est que nous détectons les problèmes d'additivité et calculons les agrégations en temps réel de manière correcte. Les autres solutions en temps réel se limitent à la détection des problèmes d'additivité sans calculer les agrégations. Notre solution s'adapte à tout type de hiérarchie complexe, contrairement à la solution de référence de Pedersen

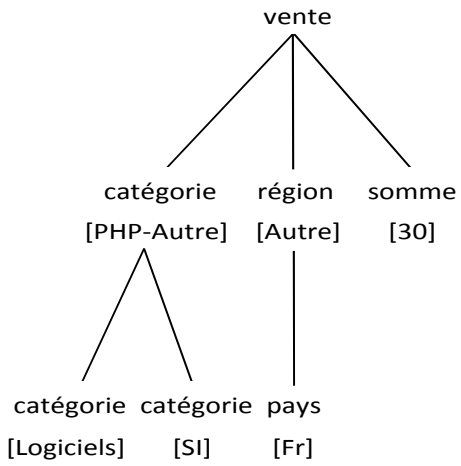



FIGURE 4.13 – Requête $Q_2$ : étape 1

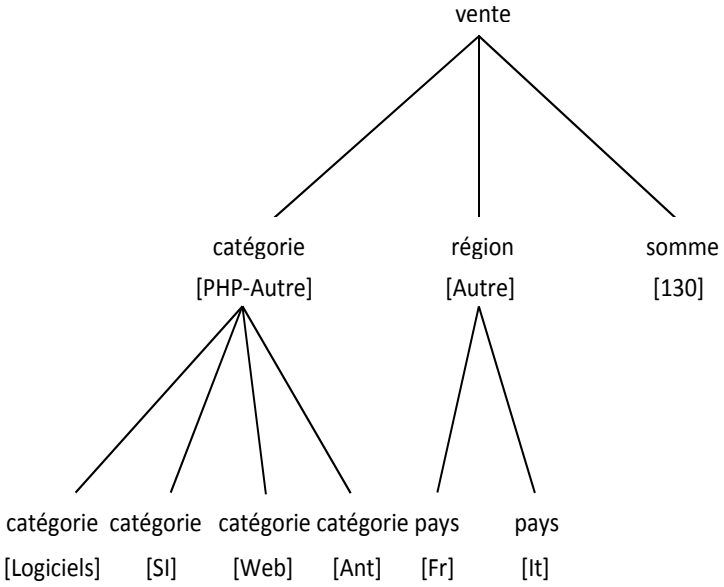



FIGURE 4.14 – Requête $Q_2$ : étape 2

*et al.* [PJD99], qui n'est pas adaptée à quelques situations du monde réel comme les hiérarchies géographiques en Chine [LSZY05].

Nous modélisons les provinces-villes chinoises comme Pékin [LSZY05] dans notre modèle comme suit. Comme les provinces-villes sont plutôt définies comme

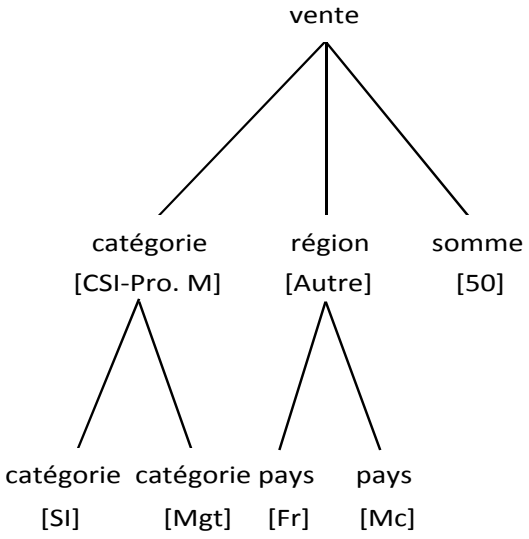



FIGURE 4.15 – Requête $Q_2$ : étape 3

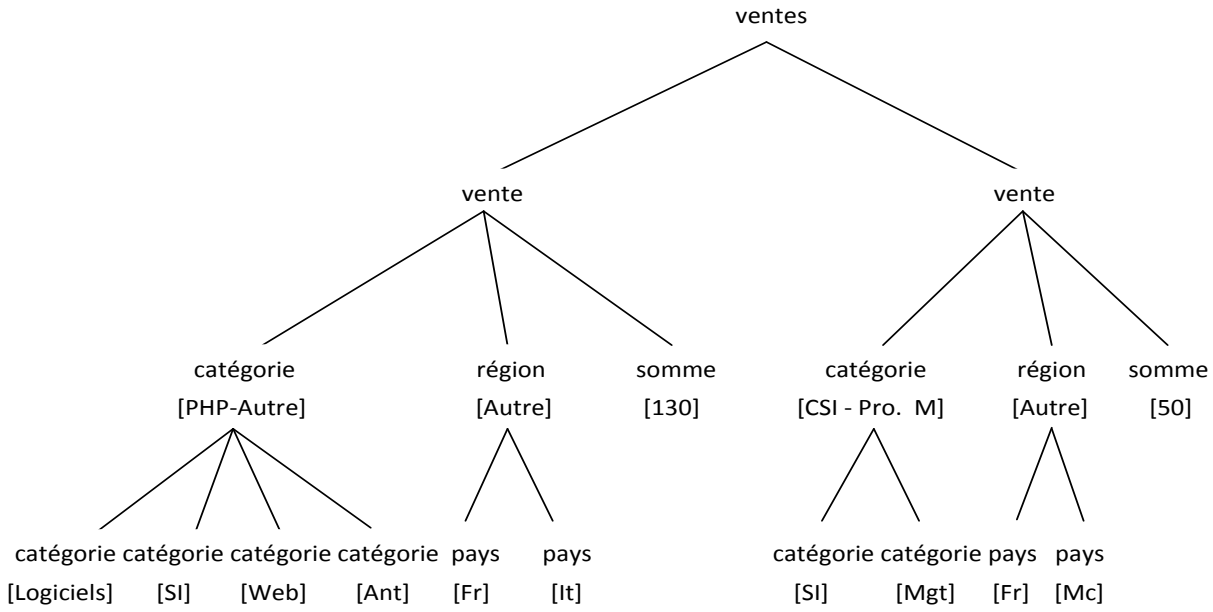



FIGURE 4.16 – Résultat de la requête $Q_2$

des provinces plutôt que des villes dans le système chinois, nous les modélisons comme des provinces dans notre modèle de données. Pour ces provinces, l'in-

formation sur la ville est manquante. D'ailleurs, dans l'exemple de la Figure 4.3, nous modélisons le cas similaire de l'état-ville Monaco (`pays[Mc]`) comme un pays puisque Monaco est toujours défini comme un pays et jamais comme une ville. Au moment de l'exécution de la requête de forage vers le haut "somme totale des ventes par région et par catégorie de niveau 2" (Section 4.5.2), nous avons remplacé l'information sur la région et la ville de Monaco par la valeur artificielle "Autre". Il suffit donc de traiter les provinces-villes (et tous les autres cas particuliers) de la même façon que Monaco.

Par ailleurs, pour vérifier l'efficacité de l'Algorithme 1, nous avons calculé sa complexité en fonction des paramètres suivants :

- $f$ : le nombre de faits ;
- $d$ : le nombre de dimensions ;
- $s$ : le nombre d'instances d'une dimension modélisée comme une hiérarchie non-stricte.

Lors de l'étape 1 (traitement de l'additivité), pour chaque dimension de chaque fait, l'Algorithme 1 fusionne toutes les valeurs possibles sélectionnées par \$4 dans un seul groupe en cas de hiérarchie non-stricte, et ajoute la valeur "Autre" à ce groupe en cas de hiérarchie non-couvrante. Dans le pire des cas, le nombre de tests à effectuer est $f \times d \times (1 + 2 + ... + s - 1)$. Donc, la complexité du traitement de l'additivité est $\mathcal{O}(fds^2)$.

Lors de l'étape 2 (*matching* des groupes), pour chaque fait, l'Algorithme 1 teste l'existence de *Group_list* dans *WTlist*. Si nous suivons le même raisonnement que pour l'étape 1, la complexité du *matching* des groupes est $\mathcal{O}(f^2ds^2)$. Par conséquent, la complexité globale de l'Algorithme 1 est $\mathcal{O}(f^2ds^2) + \mathcal{O}(fds^2) = \mathcal{O}(f^2ds^2) \approx \mathcal{O}((fds)^2)$.

Soit $n = fds$ la taille de l'entrée de l'Algorithme 1, alors la complexité de l'Algorithme 1 est $\mathcal{O}(n^2)$, c'est-à-dire la même que celle des algorithmes de Pedersen *et al.* [PJD99]. De plus, Pedersen *et al.* affirment que la complexité de leurs algorithmes est $\mathcal{O}(n\ log\ n)$ dans la majorité des cas réels [PJD99].

En effet, toutes les étapes des algorithmes de Pedersen *et al.* (*MakeCovering* et *MakeStrict*) sont exprimées à l'aide d'opérateurs relationnels [PJD99]. Comme l'entrée de *MakeCovering* se résume dans des hiérarchies, sa complexité est égale à la complexité de l'opération la plus importante dans son processus, la jointure : $\mathcal{O}(n\ log\ n)$ tel que $n$ la taille de l'entrée de *MakeCovering* (le nombre de

hiérarchies).

De plus, la complexité maximale de *MakeStrict* est $\mathcal{O}(pnk\ log\ n\ log\ k)$ tel que $p$ le nombre d'éléments parents et enfants dans une hiérarchie de dimension, $n$ la taille de l'entrée de l'algorithme (le nombre de hiérarchies) et $k$ le nombre maximum des valeurs fusionnées dans un groupe. Dans la majorité des cas réels, $p$ est $k$ sont des petites constantes et la complexité de *MakeStrict* est $\mathcal{O}(n\ log\ n)$.

Dans le Tableau 4.1, nous comparons notre solution pour les problèmes d'additivité (Algorithme 1) aux autres solutions en temps réel [PRP02a, PPP04, HS05] et aux approches qui ont utilisé les mêmes solutions que nous (les groupes et "Autre") pour résoudre les problèmes d'additivité [PJD99, MS06, MS07].

| Approche | Nature | Traitement d'additivité | Stratégies utilisées | Complexité |
|---|---|---|---|---|
| Algorithme 1 | temps réel | détection et calcul | groupes et "Autre" | $\mathcal{O}(n^2)$ |
| [PRP02a, PPP04, HS05] | temps réel | détection uniquement | N/A | N/A |
| [PJD99] | *a priori* | détection et pré-traitement | groupes, métadonnées et avis d'expert | $\mathcal{O}(n^2)$ |
| [MS06, MS07] | *a priori* | détection et pré-traitement | groupes et "Autre" | $\mathcal{O}(n^2)$ |

TABLE 4.1 – Récapitulatif de notre solution et les solutions pour les problèmes d'additivité similaires

Les critères utilisés dans le Tableau 4.1 sont :

- **la nature** de l'approche : en temps réel ou *a priori* ;
- **le type de traitement de l'additivité** : la logique suivie pour traiter l'additivité ;
- **les stratégies utilisées** pour résoudre l'additivité (groupes, métadonnées...) ;
- **la complexité** des algorithmes utilisés.

Enfin, la complexité des algorithmes de Pedersen *et al.* [PJD99] ($\mathcal{O}(n^2)$ ou $\mathcal{O}(n\ log\ n$ dans la majorité des cas) n'inclut pas la complexité des requêtes OLAP. La complexité de notre algorithme (l'Algorithme 1), $\mathcal{O}(n^2)$, inclut par contre la complexité des requêtes OLAP.

# Chapitre 5

# Implémentation et validation expérimentale

Afin de valider nos contributions, nous avons mené des expériences qui nous ont permis d'évaluer notre solution pour traiter les problèmes d'additivité dans les hiérarchies complexes et de la comparer à l'approche de Pedersen *et al.* [PJD99]. Nous choisissons de comparer notre approche à celle de Pedersen *et al.* pour les raisons suivantes. Tout d'abord, les approches en temps réel existantes présentent des solutions incomplètes pour les problèmes d'additivité, qui se limitent à des directives de correction ou à la détection de ces problèmes [PRP02a, PPP04, HS05]. La comparaison avec notre approche est donc impossible.

Par ailleurs, nous utilisons comme Pedersen *et al.* une solution basée sur les groupes pour corriger les problèmes d'additivité dans les hiérarchies non-strictes. Pour les hiérarchies non-couvrantes, le principe n'est en revanche pas le même, puisque nous remplaçons les valeurs manquantes par des valeurs "Autre" tandis que Pedersen *et al.* les remplacent par des valeurs extraites de métadonnées et/ou validées par un expert.

Enfin, l'approche de Pedersen *et al.* est considérée comme une approche de référence dans le domaine des entrepôts de données. En effet, de nombreux travaux ultérieurs la citent et en reprennent les principes [MS06, MS07].

Dans nos expériences, nous utilisons le banc d'essais XWeB (*XML Data Warehouse Benchmark*) [MD10], qui est le seul banc d'essais pour entrepôts XML à notre connaissance, après l'avoir étendu avec des hiérarchies complexes.

Ce chapitre est organisé comme suit. Dans la Section 5.1, nous décrivons comment nous implémentons les algorithmes nécessaires pour l'évaluation de notre approche. Dans la Section 5.2, nous présentons XWeB et l'extension que nous en faisons. Dans la Section 5.3, nous présentons les expériences que nous avons menées et en discutons les résultats.

## 5.1 Implémentation

Afin de valider notre solution pour les problèmes d'additivité (Chapitre 4), nous avons conçu les prototypes logiciels suivants. En premier lieu, nous avons programmé l'Algorithme 1 (Section 4.5) en Java. Nous avons ensuite programmé les algorithmes de Pedersen *et al.* [PJD99] (*MakeCovering* et *MakeStrict*), également en Java. Dans ce qui suit, nous décrivons comment nous implémentons *MakeCovering* (Section 5.1.1) et *MakeStrict* (Section 5.1.2).

### 5.1.1 Implémentation de *MakeCovering*

Pedersen *et al.* transforment une hiérarchie non-couvrante en hiérarchie couvrante en remplaçant les valeurs manquantes par des valeurs extraites de métadonnées et/ou validées par un expert [PJD99]. Cette solution nécessite donc des connaissances concernant les données et le système d'information. Nous implémentons l'algorithme *MakeCovering* en suivant ce principe (Algorithme 2). Pour cela, nous jouons le rôle de l'expert et nous créons des métadonnées "à la main". Ces métadonnées sont définies en entrée de l'Algorithme 2.

Par exemple, supposons que les données en entrée de *MakeCovering* soient exprimées selon plusieurs hiérarchies et que, parmi ces hiérarchies, nous ayons une hiérarchie géographique *ville-région-pays*. Supposons maintenant que nous ayons quatre instances de cette hiérarchie géographique et que chacune de ces instances contienne une ville européenne : Paris, Lyon, Rome et Barcelone. Avant de vérifier si ces instances de hiérarchies induisent une hiérarchie non-couvrante, nous créons les métadonnées suivantes conformément au schéma *ville-région-pays*, sachant que nous disposons des connaissances nécessaires concernant les hiérarchies géographiques en Europe. Par exemple, cette connaissance peut inclure le fait que la région de "Paris" est "Île-de-France" et que cette ville est la capitale de

"France" (le schéma de la hiérarchie *ville-région-pays* est respecté).

En entrée de l'algorithme *MakeCovering*, les données sont modélisées en hiérarchies complexes [PJD99]. Dans notre retranscription (Algorithme 2), les données sont modélisées à l'aide de notre modèle de données (un arbre de données multidimensionnel) qui supporte les hiérarchies complexes (Section 4.3). Le principe de cet algorithme est le suivant. Il prend en entrée :

- $A$ : un arbre de données multidimensionnel (qui peut contenir des hiérarchies complexes) ;
- $S$ : le schéma de la base de données correspondant aux données de $A$. Le schéma $S$ est considéré comme une métadonnée ;
- $MD$ : les métadonnées associées aux données de $A$, que nous avons créées "à la main" conformément à $S$.

**Algorithme 2** MakeCovering

```
 1: Entrées : A // arbre de données multidimensionnel
              S // schéma de la base de données A
              MD // métadonnées associées à A
 2: pour tout F de A faire
 3:    pour tout D_S de S faire
 4:       si A.containsDimension(D_S.returnType()) = false alors
 5:          A.insert(MD.extractMetadata(A, S))
 6:       fin si
 7:       si D_S.containsHierarchie(H_DS) = true alors
 8:          si H_DS.isNonCovering() = true alors
 9:             A.insert(MD.extractMetadata(A, S))
10:          fin si
11:       fin si
12:    fin pour
13: fin pour
14: retourner A
```

Pour chaque fait $F$ de $A$ et pour chaque dimension $D_S$ de $S$, l'Algorithme 2 vérifie s'il existe dans $A$ une dimension de même type que $D_S$ à l'aide des fonctions `containsDimension()` et `returnType()` (ligne 5). Si ce test s'avère négatif, l'Algorithme 2 extrait la valeur manquante (la dimension de même type que $D_S$ et sa valeur) à partir des métadonnées $MD$ à l'aide de la fonction `extractMetadata()` et l'insère dans $A$, conformément à $S$, à l'aide de la fonction `insert()` (ligne 6).

De plus, si $D_S$ comporte une hiérarchie $H_{D_S}$, l'Algorithme 2 vérifie s'il existe des éléments manquants dans les instances de $H_{D_S}$ dans $A$ à l'aide de la fonction

`isNonCovering()` (ligne 9). Si ce test est positif, l'algorithme détermine les valeurs manquantes à partir des métadonnées *MD* et les insère dans $A$, conformément à $S$, à l'aide des fonctions `extractMetadata()` et `insert()`, respectivement. Enfin, l'Algorithme 2 retourne l'arbre $A$.

### 5.1.2 Implémentation de *MakeStrict*

Pedersen *et al.* transforment une hiérarchie non-stricte en hiérarchie stricte en fusionnant, dans une première étape, toutes les valeurs de ses niveaux hiérarchiques dans des groupes [PJD99]. Ils insèrent ensuite ces groupes comme éléments parents de ces valeurs. Les groupes sont ensuite pris en compte lors du calcul d'agrégation pour garantir l'additivité. Nous implémentons *MakeStrict* en respectant ce même principe : la fusion des valeurs de la hiérarchie non-stricte dans des groupes (Algorithme 3).

**Algorithme 3** MakeStrict

```
1: Entrées : A // arbre de données multidimensionnel
2: pour tout D_A de A faire
3:     si H_{D_A}.nbrInstances() > 1 alors
4:         H_{D_A}.createGroups()
5:         A.insertGroups()
6:     fin si
7: fin pour
8: return A
```

En entrée de l'Algorithme 3, les données sont modélisées par un arbre de données multidimensionnel (Section 4.3). Contrairement à *MakeCovering* (Algorithme 2), notre algorithme ne prend en entrée qu'un arbre de données multidimensionnel $A$, qui peut contenir des hiérarchies non-strictes. À l'aide de la fonction `nbrInstances()` (ligne 4), l'Algorithme 3 vérifie, pour chaque dimension $D_A$ qui comporte une hiérarchie $H_{D_A}$, si le nombre d'instances de $H_{D_A}$ est supérieur à 1 car, dans ce cas, $H_{D_A}$ est non-stricte. L'Algorithme 3 crée alors des groupes et les insère dans $A$ à l'aide des fonctions `createGroups()` et `insertGroups()`, respectivement. Nous rappelons qu'un groupe remplace toutes les valeurs à l'origine de ce groupe dans l'arbre de données résultat. Enfin, l'Algorithme 5 retourne l'arbre $A$.

## 5.2 Banc d'essais

### 5.2.1 XWeB

Les bancs d'essais permettent d'évaluer les performances des SGBD en général et de tester l'efficacité des techniques d'optimisation des performances en particulier. Dans ce sens, Mahboubi et Darmont proposent un banc d'essais pour entrepôts de données XML nommé XWeB (*XML Data Warehouse Benchmark*) [MD10]. Leur motivation principale provient du fait que les bancs d'essais XML existants ne permettent pas d'évaluer les performances des applications décisionnelles [SWK+02, YOK04]. Ces systèmes proposent des bases de données transactionnelles et leurs charges exploitent des requêtes simples. Mahboubi et Darmont présentent par conséquent à travers XWeB un outil décisionnel basé sur un entrepôt modélisé de façon multidimensionnelle qui exploite des requêtes décisionnelles plus complexes. XWeB est constitué de deux composants principaux (Figure 5.1) : un modèle de base de données qui respecte le modèle d'entrepôt de données XML de Mahboubi *et al.* [MRL+09] (Section 3.1.3) et un modèle de charge de requêtes.

#### Entrepôt de données XWeB

**Schéma de la base de données.** Mahboubi et Darmont se sont appuyés sur le banc d'essais décisionnel relationnel standard TPC-H [TPC05b] pour concevoir le modèle de base de données de XWeB [MD10]. Les auteurs ont sélectionné TPC-H pour la simplicité de son modèle, en opposition à d'autres bancs décisionnels existants (comme TPC-DS [TPC05a]), dont la structure en constellation est plus difficile à mettre en œuvre. Mahboubi et Darmont ont donc étendu TPC-H et remodélisé ses données dans un schéma multidimensionnel en flocon de neige. Le modèle conceptuel est présenté sous la forme d'un diagramme de classes UML dans la Figure 5.2.

Ce schéma d'entrepôt XML représente le cas d'école classique de faits *ventes* caractérisés par : (1) les dimensions *produits* (*Dim_Parts*), *clients* (*Dim_Customers*), *fournisseurs* (*Dim_Suppliers*), *date* (*Dim_Date*) et (2) les mesures *quantité* de produits achetés (*quantity*) et *montant* des produits achetés (*amount*). Comme le montre le modèle de la Figure 5.2, la dimension *date* est modélisée en hiérarchie :

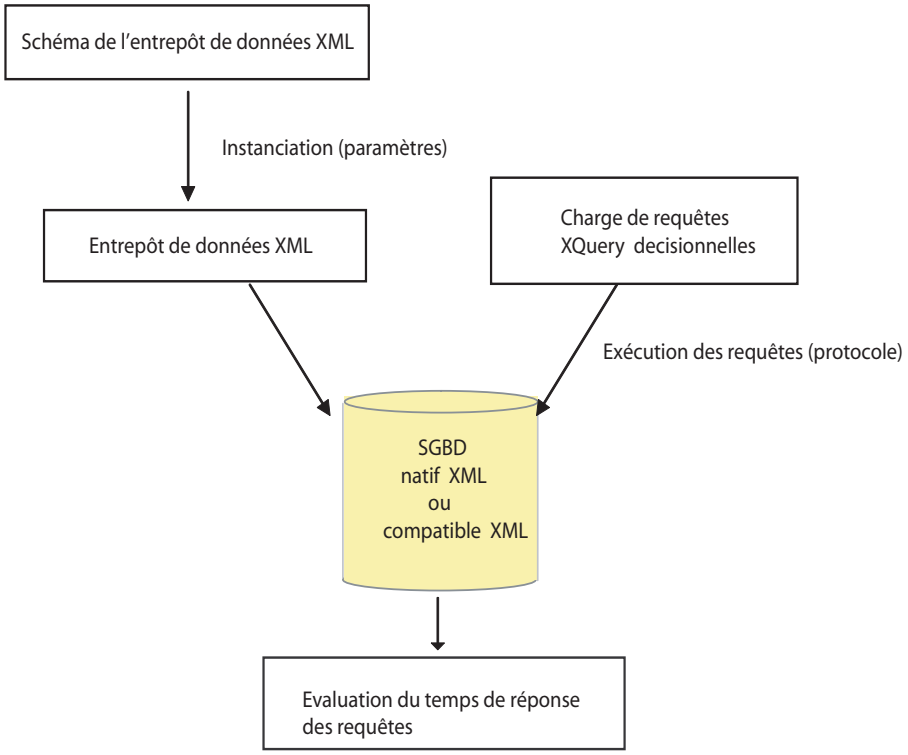



FIGURE 5.1 – Principe d'utilisation de XWeB [Mah08]

*jour*, *mois* et *année*. Ce schéma est stocké dans un document *dw-model.xml*.

**Charge de XWeB.** La charge du banc d'essais XWeB comporte vingt requêtes XQuery, numérotées de Q1 à Q20 (Tableau 5.1). Ces requêtes exploitent la totalité du schéma de l'entrepôt par des opérations de sélection, de jointure et d'agrégation. La spécification de chaque requête de la charge est décrite dans le Tableau 5.1. À titre d'exemple, le code XQuery correspondant à la requête Q1 est illustré dans la Figure 5.3.

### 5.2.2 Extension du schéma de XWeB par des hiérarchies complexes

Dans la version de XWeB de Mahboubi et Darmont, les auteurs modélisent la hiérarchie *part-category* en une hiérarchie complexe [MD10]. Toutefois, c'est la

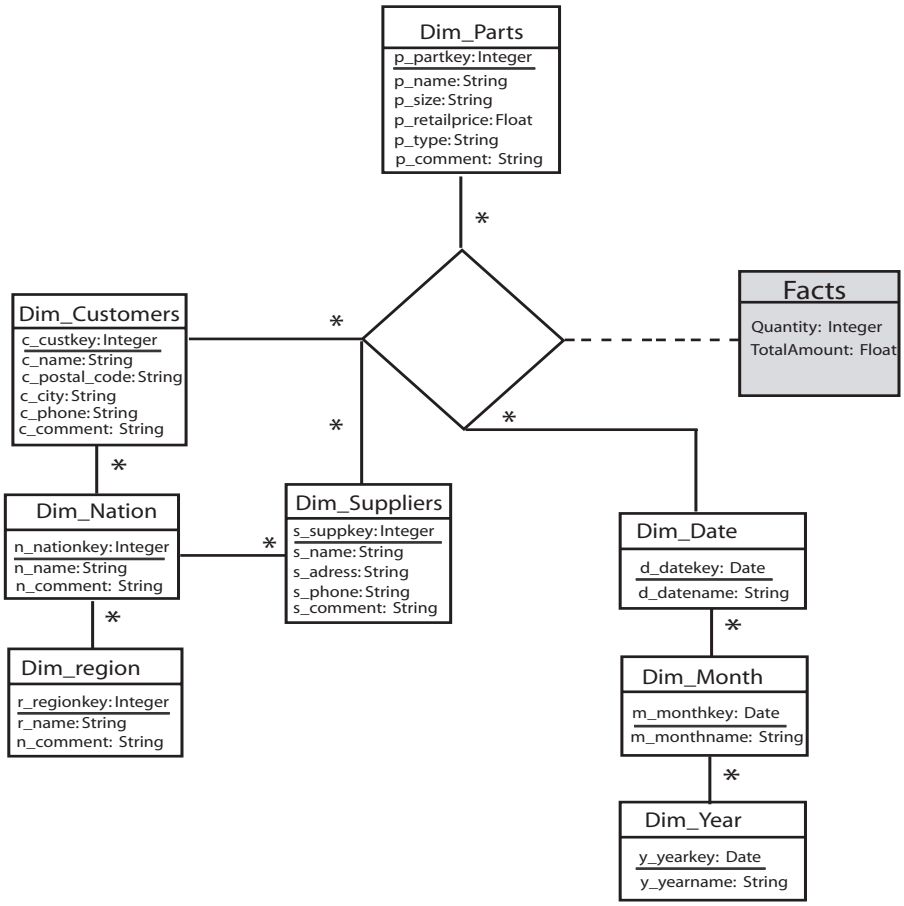



FIGURE 5.2 – Schéma conceptuel de l'entrepôt XWeB [Mah08]

```
for $x in //FactDoc/Fact, $a in
//dimensions/Level[@id='Customers']/instance where
$a/attribute[@id='c_nationkey']/@value<13 and
$x/dimension[@dim-id='Customers']/@value-id= $a/@id
return $x
```

FIGURE 5.3 – Requête Q1 exprimée en XQuery [Mah08]

seule hiérarchie complexe présente dans le modèle et elle n'est pas paramétrable. Pour répondre à nos besoins expérimentaux, nous avons donc étendu le schéma de

| Requête | Spécification |
|---|---|
| Q1 | Nombre d'achats par des consommateurs français, groupé par jour |
| Q2 | Nombre d'achats par des consommateurs français et anglais, groupé par mois |
| Q3 | Nombre d'achats par des consommateurs anglais de produits français |
| Q4 | Nombre d'achats des consommateurs anglais, groupé par produit |
| Q5 | Nombre d'achats par des consommateurs français de produits asiatiques les samedis |
| Q6 | Nombre d'achats des consommateurs lyonnais les samedis |
| Q7 | Nombre d'achats des consommateurs lyonnais, groupé par mois |
| Q8 | Nombre d'achats du produit "SMALL PLATED BRASS" par des consommateurs parisiens |
| Q9 | Nombre d'achats de produits parisiens par des consommateurs lyonnais les samedis |
| Q10 | Nombre d'achats du produit "SMALL PLATED BRASS", groupé par année |
| Q11 | Nombre d'achats du produit "SMALL PLATED BRASS" groupé par les clients anglais et les fournisseurs français |
| Q12 | Nombre d'achats du produit "SMALL PLATED BRASS" livré par des parisiens et acheté par les lyonnais |
| Q13 | Nombre d'achats les lundis |
| Q14 | Nombre d'achats par des consommateurs parisiens, groupé par mois |
| Q15 | Nombre d'achats du produit "SMALL PLATED BRASS" les lundis |
| Q16 | Nombre d'achats par des consommateurs lyonnais du produit "SMALL PLATED BRASS" les mercredis |
| Q17 | Nombre d'achats des produits le mercredi |
| Q18 | Nombre d'achats des produits le lundi, groupé par consommateur |
| Q19 | Nombre d'achats par des consommateurs lyonnais les samedis |
| Q20 | Nombre d'achats de produits livrés par des fournisseurs parisiens, groupé par mois |

TABLE 5.1 – Spécification de la charge de XWeB [Mah08]

XWeB avec des hiérarchies complexes plus nombreuses et plus diverses. Dans ce qui suit, nous présentons en premier lieu le modèle de données qui permet à XWeB de supporter les hiérarchies complexes. Nous présentons ensuite les algorithmes que nous avons définis pour générer des hiérarchies complexes.

### Modèle logique en arbre de données multidimensionnel

Nous modélisons les données XWeB à l'aide de notre modèle de données (les arbres de données multidimensionnels) comme le montre la Figure 5.4.

La racine de l'arbre de données modélise l'entrepôt de données *sales* (ventes). Les nœuds *sale* (vente) modélisent les faits. Un fait *sale* est défini par quatre dimensions et deux mesures :

- la dimension *part* (produit) définie par la hiérarchie *type3*-*type2*-*type1*. *type3* est le niveau hiérarchique le plus détaillé et *type1* est le niveau hiérarchique le plus général. Les valeurs de *type3*, de *type2* et de *type1* sont données dans

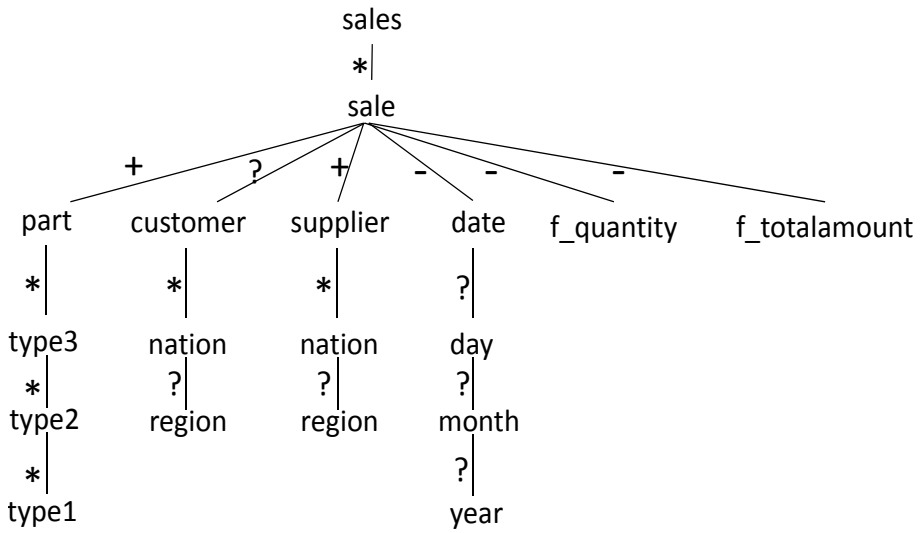



FIGURE 5.4 – Schéma de XWeB exprimé en arbre de données multidimensionnel

le Tableau 5.2 ;

| | |
|---|---|
| *type3* | ECONOMY, LARGE, STANDARD, PROMO, MEDIUM, SMALL |
| *type2* | ANODIZED, BURNISHED, BRUSHED, POLISHED, PLATED |
| *type1* | COPPER, NICKEL, STEEL, TIN, BRASS |

TABLE 5.2 – Niveaux hiérarchiques de la dimension *part*

- la dimension *customer* (client) définie par la hiérarchie *nation-region*. *nation* (pays) représente le niveau hiérarchique le plus détaillé et *region* (continent) représente le niveau hiérarchique le plus général ;
- la dimension *supplier* (fournisseur) définie par la hiérarchie *nation-region*. *nation* (*pays*) représente le niveau hiérarchique le plus détaillé et *region* (continent) représente le niveau hiérarchique le plus général ;
- la dimension *date* définie par la hiérarchie *day-month-year*. *day* (jour) représente le niveau hiérarchique le plus détaillé et *year* (année) représente le niveau hiérarchique le plus général ;
- les mesures *f_quantity* (quantité) et *f_totalamount* (montant total).

Pour exprimer les hiérarchies complexes, nous associons les annotations suivantes aux arêtes $(x, y)$ de l'arbre de données de la Figure 5.4, où $x$ est le nœud

parent du nœud $y$ dans l'arbre de données. Ces annotations sont reprises du modèle d'arbre annoté de Paparizos *et al.* [PWLJ04].

- - : un et un seul nœud de type $y$ est relié au nœud $x$. Cette annotation autorise la modélisation de hiérarchies simples ;
- ? : zéro ou un nœud de type $y$ est relié au nœud $x$. Cette annotation autorise la modélisation des hiérarchies non-couvrantes ;
- $+$ : un à plusieurs nœuds de type $y$ sont reliés au nœud $x$. Cette annotation autorise la modélisation de hiérarchies non-strictes ;
- $*$ : zéro à plusieurs nœuds de type $y$ sont reliés au nœud $x$. Cette annotation autorise la modélisation de hiérarchies complexes.

À travers ces annotations, nous étendons le schéma de XWeB avec le maximum de hiérarchies complexes tout en respectant la sémantique des données. En effet, certaines hiérarchies ne peuvent pas être modélisées en hiérarchies complexes par respect à la sémantique des données.

- Dans la dimension *produit* (*part*), un type donné peut être relié à un ou plusieurs types. Un produit donné peut également être décrit par plusieurs types. Nous modélisons donc la hiérarchie *type3-type2-type1* ainsi que la relation *part-type* en hiérarchies complexes par l'annotation "$*$".
- Un client (*customer*) et un fournisseur (*supplier*) donnés peuvent avoir des adresses dans plusieurs pays (*nation*). Pour cette raison, nous modélisons les relations *customer-nation* et *supplier-nation* en hiérarchies complexes par l'annotation "$*$". Par ailleurs, un pays (*nation*) appartient à un et un seul continent (*region*), ce qui explique que nous modélisons les hiérarchies *nation-region* en hiérarchies non-couvrantes seulement, par l'annotation " ?" dans ces deux dimensions.
- Dans la dimension *date*, nous modélisons les relations *date-day*, *day-month* et *month-year* en hiérarchies non-couvrantes seulement à l'aide du symbole " ?". En adoptant cette modélisation, nous respectons la sémantique des données puisqu'une date donnée correspond à un et un seul jour (*day*), un jour donné appartient à un et un seul mois (*month*) et un mois donné appartient à une et une seule année (*year*).

### Génération de divers types de hiérarchies

Afin de varier les types de hiérarchies utilisés dans nos expériences, nous définissons quatre "degrés" de complexité des hiérarchies : (1) simples, (2) non-couvrantes seulement, (3) non-strictes seulement et (4) complexes. Pour pouvoir générer un entrepôt de données avec ces quatre types de hiérarchies, nous utilisons les paramètres suivants dans les algorithmes que nous définissons pour générer les hiérarchies complexes [KHD12] :

- *fact_number* : nombre de faits à générer ;
- *incomplete_percentage* : pourcentage d'instances de hiérarchies de dimensions non-couvrantes par rapport au nombre total d'instances de hiérarchies de dimensions dans un fait ;
- *nonstrict_percentage* : pourcentage d'instances de hiérarchies de dimensions non-strictes par rapport au nombre total d'instances de hiérarchies de dimensions dans un fait ;
- *nonstrict_number* : nombre d'instances de hiérarchies non-strictes dans une dimension.

Par conséquent, les quatre types de hiérarchies que nous utilisons dans nos expériences sont définies comme suit :

- hiérarchies simples : *incomplete_percentage* $= 0$ et *nonstrict_percentage* $= 0$ ;
- hiérarchies non-couvrantes : *incomplete_percentage* $> 0$ et *nonstrict_percentage* $= 0$ ;
- hiérarchies non-strictes : *incomplete_percentage* $= 0$ et *nonstrict_percentage* $> 0$ ;
- hiérarchies complexes : *incomplete_percentage* $> 0$ et *nonstrict_percentage* $> 0$.

**Génération de hiérarchies simples.** XWeB propose des hiérarchies simples par défaut. La Figure 5.5 illustre un exemple de fait *sale#1* modélisé en hiérarchie simple, conformément au modèle de données de la Figure 5.4.

Dans cet exemple, le produit *part#1* est vendu au client *customer#1*, qui vit aux USA, le 25/06/1998. Le fournisseur de ce produit se trouve en Europe, plus précisément en France. La quantité de produits vendus de 100 pièces, ce qui a coûté au client 2800 $.

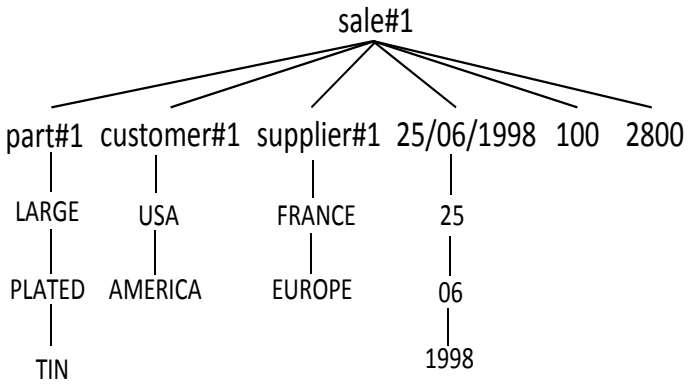



FIGURE 5.5 – Exemple de hiérarchie simple de XWeB

**Génération de hiérarchies non-couvrantes.** Les hiérarchies non-couvrantes sont générées lorsque le paramètre *incomplete_percentage* est supérieur à zéro et lorsque le paramètre *nonstrict_percentage* est égal à zéro. Nous utilisons l'Algorithme 4 pour ajouter des hiérarchies non-couvrantes dans le schéma de XWeB.

**Algorithme 4** Génération de hiérarchies non-couvrantes

```
1: Entrées : dim, incomplete_percentage
2: ic_check ← false
3: tant que ic_check = false faire
4:    pour chaque niveau de dim faire
5:       supprimer le niveau courant de façon aléatoire selon la valeur de incomplete_percentage
6:       si niveau courant supprimé alors
7:          ic_check ← true
8:       fin si
9:    fin pour
10: fin tant que
11: retourner dim
```

Le paramètre *dim* en entrée de l'Algorithme 4 spécifie le nom de la dimension à traiter. Le paramètre *ic_check* permet de spécifier l'existence de hiérarchies non-couvrantes dans la dimension *dim*. Si *dim* ne contient pas de hiérarchie non-couvrante, c'est-à-dire tant que *ic_check* = *false* (ligne 3), l'Algorithme 4 transforme une ou plusieurs hiérarchies en hiérarchies non-couvrantes en supprimant

des niveaux hiérarchiques de façon aléatoire selon la valeur du paramètre *incomplete_percentage* (lignes 4 et 5). Dans l'exemple de la Figure 5.6, nous générons une hiérarchie non-couvrante à partir de la hiérarchie de dimension *part#1* de la Figure 5.5. Dans cet exemple, l'instance de niveau hiérarchique "LARGE" a été supprimé.

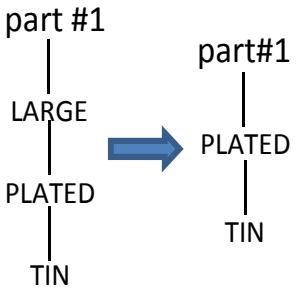



FIGURE 5.6 – Exemple de génération de hiérarchie non-couvrante

**Génération de hiérarchies non-strictes.** Pour générer des hiérarchies non-strictes, nous utilisons les paramètres *nonstrict_percentage* pour spécifier la distribution des hiérarchies non-strictes et *nonstrict_number* pour spécifier le nombre de maximum d'instances de hiérarchies non-strictes dans une dimension. L'Algorithme 5 illustre la procédure que nous définissons pour générer les hiérarchies non-strictes.

**Algorithme 5** Génération de hiérarchies non-strictes

1: **Entrées :** *dim*, *nonstrict_number*, *nonstrict_percentage*
2: *ns_array* ← *null*
3: **tant que** *nonstrict_number* > 0 **faire**
4: La dimension *dim* est sélectionnée et générée sous la forme de hiérarchie non-stricte de façon aléatoire selon les valeurs de *nonstrict_percentage*
5: *ns_array* ← *dim*
6: *nonstrict_number* ← *nonstrict_number* - 1
7: **fin tant que**
8: **retourner** *ns_array*

Dans l'Algorithme 5, nous utilisons des tableaux de hiérarchies non-strictes (*ns_array*) pour enregistrer les instances de hiérarchies non-strictes. Un exemple de tableau de hiérarchies non-strictes est illustré à gauche de la Figure 5.7. Les lignes du tableau *ns_array* modélisent les instances de dimensions (la dimension *supplier* dans le tableau de la Figure 5.7). Ses colonnes modélisent les différents niveaux hiérarchiques de la dimension en question (*nation* et *region* dans le tableau de la Figure 5.7).

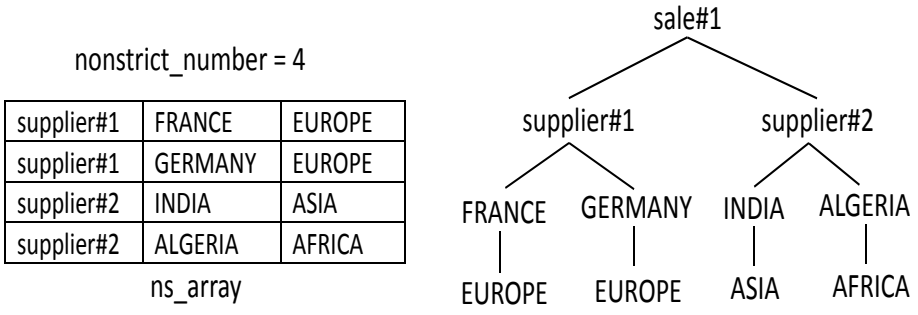



FIGURE 5.7 – Exemple de génération de hiérarchie non-stricte

Le paramètre *dim* en entrée de l'Algorithme 5 spécifie le nom de la dimension à traiter. Les valeur des paramètres *nonstrict_number* et *nonstrict_percentage* sont déterminées de façon aléatoire. Tant que *nonstrict_number* est supérieur à zéro (ligne 3), l'algorithme continue la sélection de dimensions pour y introduire des hiérarchies non-strictes. À chaque fois qu'une dimension est sélectionnée (ligne 4), elle est enregistrée dans le tableau des hiérarchies non-strictes (*ns_array*). Enfin, *ns_array* est transformé en XML. Dans l'exemple de la Figure 5.7, la dimension sélectionnée est fournisseur (*supplier*) et *nonstrict_number* est égal à 4. C'est pourquoi le nombre de lignes (instances de dimensions) dans le tableau de la Figure 5.7 est égal à 4. L'arbre de données à droite de la Figure 5.7 modélise les données du tableau *ns_array* dans notre modèle de données.

**Génération de hiérarchies complexes.** Une hiérarchie est complexe lorsqu'elle est à la fois non-stricte (c'est-à-dire lorsque *nonstrict_percentage* est supérieur à zéro et *nonstrict_number* est supérieur à un) et non-couvrante (c'est-à-dire lorsque *incomplete_percentage* est supérieur à zéro). L'Algorithme 6 illustre la

procédure que nous suivons pour générer les hiérarchies complexes. Nous transformons un tableau de hiérarchie non-stricte *ns_array* en tableau de hiérarchie complexe en remplaçant au moins une de ses cellules par une cellule vide.

**Algorithme 6** Génération de hiérarchies complexes

1: **Entrées :** *ns_array*
2: *ic_check* ← *false*
3: **tant que** *ic_check* = *false* **faire**
4: **pour** chaque ligne de *ns_array* **faire**
5: déterminer la dimension non-couvrante à générer de façon aléatoire et transformer certains cellules de *ns_array* en cellules vides de façon aléatoire également
6: **si** niveau courant supprimé **alors**
7: *ic_check* ← *true*
8: **fin si**
9: **fin pour**
10: **fin tant que**
11: **retourner** *ns_array*

L'Algorithme 6 génère en premier lieu un tableau de hiérarchies non-strictes (*ns_array*) en faisant appel à l'Algorithme 4 et initialise la valeur *ic_check* à *faux*. Tant que *ic_check* est égal à *faux*, l'Algorithme 6 génère une hiérarchie complexe en éliminant une ou plusieurs instances de niveaux hiérarchiques du tableau *ns_array* de façon aléatoire (lignes 4 à 6). Enfin, le tableau *ns_array* modifié est transformé en XML. Par exemple, le tableau *complex_array* de la Figure 5.8 a été généré suite à la suppression des instances "INDIA" et "EUROPE" du tableau *ns_array* de la Figure 5.7. La hiérarchie complexe correspondante au tableau *ns_array* modifié, modélisée à l'aide de notre modèle de données, apparaît à droite de la Figure 5.8.

### 5.2.3 Modèle physique XML

#### Modélisation des faits en XML

Nous modélisons tous les éléments de l'entrepôt de données *sales* dans un seul fichier XML. La racine du fichier XML est <*sales*>. Dans l'arbre de données de la Figure 5.4, les faits *sale* sont modélisés comme les nœuds enfants du nœud *sales*.

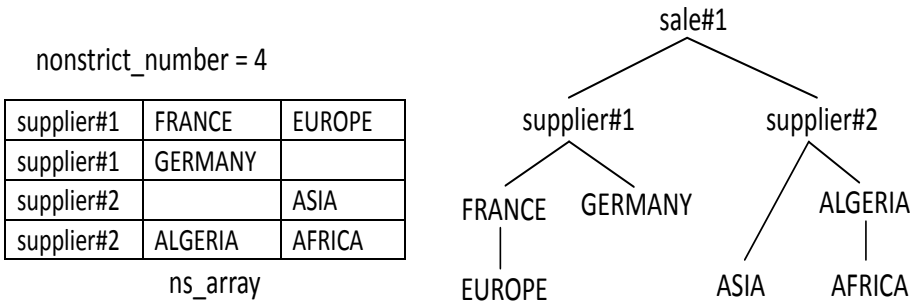



FIGURE 5.8 – Exemple de génération de hiérarchie complexe

Nous conservons cette représentation en XML et nous modélisons les faits par des éléments <*sale*> enfants de <*sales*>.

De même, les dimensions (*part*, *customer*, *supplier* et *date*) et les mesures (*f_quantity* et *f_totalamount*) sont modélisées comme des éléments XML enfants de <*sale*>, les hiérarchies des dimensions *part*, *customer*, *supplier* et *date* sont aussi modélisées en XML conformément à la structure de l'arbre de données XWeB. La Figure 5.9 illustre un fait <*sale*> modélisé en XML conformément à la structure de la Figure 5.4. Dans cet exemple, nous ne modélisons pas les hiérarchies pour l'instant.

```
<sale>
    <part value='PROMO BRUSHED BRASS' > </part>
    <customer value='Customer#000000001'> </customer>
    <supplier value='Supplier#000000121'></supplier>
    <date value='02/06/1998'> </date>
    <f_quantity value='2948'> </f_quantity>
    <f_totalamount value='4228'> </f_totalamount>
</sale>
```

FIGURE 5.9 – Exemple de fait *sale* modélisé en XML

### Modélisation des hiérarchies complexes en XML

Nous modélisons les hiérarchies non-couvrantes, non-strictes et complexes en XML en respectant le modèle logique de la Figure 5.4. Dans toutes ces hiérarchies,

nous représentons les niveaux hiérarchiques détaillés comme éléments parents des niveaux hiérarchiques plus agrégés (règle générale).

**Hiérarchies non-couvrantes.** Pour représenter les hiérarchies non-couvrantes en XML, et en plus de la règle générale, nous ne représentons pas les niveaux correspondants aux valeurs manquantes en XML. Dans l'exemple de la Figure 5.10(a), nous connaissons le pays (*nation*), mais pas le continent (*region*) du fournisseur "Supplier#01". C'est pour cette raison que l'élément <*region*> n'est pas représenté ici.

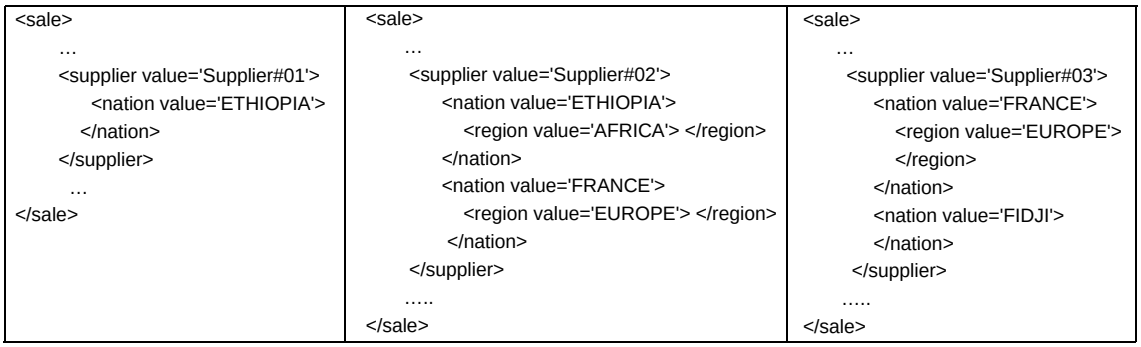



FIGURE 5.10 – Modélisation de (a) hiérarchies non-strictes, (b) non-couvrantes et (c) complexes en XML

**Hiérarchies non-strictes.** En plus de la règle générale citée ci-dessus, nous modélisons tous les éléments hiérarchiques de même type comme éléments enfants d'un seul élément XML parent. Soit la dimension *supplier* (fournisseur) de la Figure 5.4. Dans la Figure 5.10(b), nous représentons le fournisseur "Supplier#02" qui dispose de deux entreprises dans deux pays (*nation*) et deux continents (*region*) différents, et les pays "ETHIOPIA" et "FRANCE" comme les éléments enfants de <supplier value='Supplier#02'>.

**Hiérarchies complexes.** Pour modéliser les hiérarchies complexes, nous associons les représentations des hiérarchies non-strictes et non-couvrantes dans une

même représentation. Dans la Figure 5.10(b), nous modélisons le fournisseur “Supplier#03” qui a des entreprises dans deux pays différents, La “FRANCE” et Les “FIDJI”. De plus, l'information concernant le continent (*region*) des “FIDJI” est manquante.

### 5.2.4 Extension de la charge de requêtes de XWeB

La charge de requêtes de XWeB, outre qu'elle ne couvre pas toutes les hiérarchies complexes que nous avons introduites dans le schéma du banc d'essais, propose différents types de requêtes correspondants à différents usages. Comme nos besoins ne concernent que les requêtes de regroupement et d'agrégation, nous n'avons retenu que le principe de ces dernières et proposons une charge de quatre nouvelles requêtes de complexité croissante.

Comme le schéma de l'entrepôt XML contient quatre dimensions (Figure 5.4), nous utilisons quatre requêtes dans nos expériences avec les degrés de complexité suivants : une dimension (1D), deux dimensions (2D), trois dimensions (3D) et quatre dimensions (4D), comme le montre le Tableau 5.3.

| | part | customer | supplier | date |
|---|---|---|---|---|
| 1D | | | | day |
| 2D | type3 | | | day |
| 3D | type3 | nation | | day |
| 4D | type3 | nation | nation | day |

TABLE 5.3 – Dimensions et niveaux hiérarchiques accédés par les requêtes utilisées dans nos expériences

## 5.3 Expériences

### 5.3.1 Conditions expérimentales

#### Conditions matérielles et logicielles

Nous exécutons nos programmes sur un ordinateur portable Toshiba ayant les caractéristiques suivantes :

- processeur : Intel(R) Core(TM) i7-2670QM CPU @ 2.20 GHz ;
- mémoire : 4 GB ;
- système d'exploitation : 64-bit Windows 7 Home Premium, Service Pack 1.

Nous avons programmé l'extension de XWeB, l'Algorithme 1 (Section 4.5), les algorithmes *MakeCovering* et *MakeStrict* (Sections 5.1.1 et 5.1.2) à l'aide de Java JDK 1.7, en utilisant l'API SAX [1] pour parcourir, lire et créer les fichiers XML. Les programmes correspondant aux algorithmes 1 (Section 4.5), *MakeCovering* et *MakeStrict* sont disponibles en ligne [2], ainsi que l'extension de XWeB [3].

### Données utilisées dans les expériences

Pour pouvoir évaluer notre approche de manière efficace, nous avons généré plusieurs entrepôts XML à l'aide de XWeB, qui contiennent de 50 000 à 250 000 faits, comme le montre la première ligne du Tableau 5.4. La deuxième ligne du Tableau 5.4 illustre la taille des données XWeB originales. Leur taille minimale est égale à 13 Mo (50 000 faits), tandis que leur taille maximale est égale à 67 Mo (250 000 faits). La troisième ligne du Tableau 5.4 illustre la taille des données XWeB modélisées à l'aide de notre modèle de données (Figure 5.4) et utilisées dans nos expériences sur les hiérarchies simples. Leur taille varie de 27 Mo (50 000 faits) à 134 Mo (250 000 faits).

| Nombre de faits | 50 000 | 100 000 | 150 000 | 200 000 | 250 000 |
|---|---|---|---|---|---|
| XWeB | 13 661 | 27 366 | 41 070 | 54 775 | 68 479 |
| Arbres XWeB | 27 700 | 55 390 | 82 800 | 110 577 | 138 015 |
| Hiérarchies non-couvrantes 5 % | 27 626 | 55 242 | 82 543 | 110 249 | 137 573 |
| Hiérarchies non-strictes 5 % | 28 669 | 57 328 | 85 671 | 114 422 | 142 786 |
| Hiérarchies complexes 5 % | 28 376 | 56 742 | 84 791 | 113 252 | 141 319 |
| Hiérarchies non-couvrantes 50 % | 25 020 | 50 030 | 74 769 | 99 842 | 124 601 |
| Hiérarchies non-strictes 50 % | 35 412 | 70 826 | 105 914 | 141 397 | 176 527 |
| Hiérarchies complexes 50 % | 32 522 | 65 031 | 97 263 | 129 839 | 162 088 |

TABLE 5.4 – Tailles des données utilisées dans les expériences (en Ko)

1. http://www.saxproject.org/
2. http://eric.univ-lyon2.fr/~mhachicha/XOLAP.zip
3. http://eric.univ-lyon2.fr/~jdarmont/?page_id=131#xweb

Dans le but de varier nos expériences sur les hiérarchies complexes, nous générons plusieurs types d'entrepôts XML : (1) avec des hiérarchies non-couvrantes seulement, (2) avec des hiérarchies non-strictes seulement et (3) avec des hiérarchies complexes (les six dernières lignes du Tableau 5.4). Dans ce qui suit, nous donnons les significations des hiérarchies de types 5 % et 50 % mentionnées dans les six dernières lignes du Tableau 5.4. Par exemple, soit le cas de l'entrepôt XML composé de 50 000 faits. Puisque chaque fait est relié à quatre dimensions (Figure 5.4), les 50 000 faits sont reliés à 200 000 instances de dimensions au total ($4 \times 50\ 000 = 200\ 000$). "5 % de hiérarchies non-couvrantes" (respectivement non-strictes ou complexes) signifie alors que 10 000 (5 % de 200 000) de ces instances induisent des hiérarchies non-couvrantes (respectivement non-strictes ou complexes). "50 % de hiérarchies non-couvrantes" (respectivement non-strictes ou complexes) signifie alors que 100 000 (50 % de 200 000) instances induisent des hiérarchies non-couvrantes (respectivement non-strictes ou complexes). Le choix des éléments XML correspondants à des hiérarchies non-couvrantes (respectivement non-strictes ou complexes) est aléatoire. Dans le cas des hiérarchies de type 5 %, pour chaque fait, une instance de dimension sur vingt ($100 / 5 = 20$) est modifiée. Pour les hiérarchies de type 50 %, pour chaque fait, deux instances de dimension sur quatre sont modifiées.

Nous avons choisi le seuil 50 % pour les raisons suivantes. Si nous dépassons 50 %, nous aurons une hiérarchie non-couvrante (respectivement non-stricte ou complexe) par dimension. Par exemple, si 80 % des dimensions sont non-couvrantes, plus de trois instances de dimensions sur quatre sont inconnues (manquantes) pour chaque fait ($80\ \% \times 4 = 3{,}2$), ce qui signifie que toutes les dimensions sont non-couvrantes, un cas peu réaliste.

La taille des entrepôts XML avec des hiérarchies non-couvrantes est nettement inférieure aux tailles des entrepôts XML avec des hiérarchies non-strictes et complexes, respectivement (Tableau 5.4). Ceci est dû au fait que, dans un entrepôt XML donné, nous supprimons de chaque hiérarchie de dimension un élément XML au moins pour introduire des hiérarchies non-couvrantes. Par contre, nous ajoutons à chaque hiérarchie de dimension un élément XML au moins pour introduire des hiérarchies non-strictes. Enfin, la taille des entrepôts XML avec des hiérarchies complexes est plus proche de la taille des entrepôts XML avec des hiérarchies non-strictes que de la taille des entrepôts avec des hiérarchies non-couvrantes, parce

que la structure d'une hiérarchie complexe est très proche de la structure d'une hiérarchie non-stricte.

### 5.3.2 Plan d'expérience

Dans la suite de ce chapitre, nous utilisons les termes `QBS` (*Query-Based Summarizability*) [HKD12] pour désigner notre Algorithme 1 et `Pedersen` pour désigner l'approche de Pedersen *et al.* [PJD99], qui consiste à appliquer les Algorithmes 2 et 3, en séquence, *a priori*. Dans nos expériences, nous calculons les temps d'exécution des quatre requêtes du Tableau 5.3 pour `QBS` et pour `Pedersen` afin de (1) montrer la faisabilité de `QBS` en temps réel et (2) montrer que `QBS` peut passer à l'échelle.

La fonction d'agrégation choisie pour ces expériences est la *somme*. Toutefois, `QBS` et `Pedersen` supportent toutes les autres fonctions d'agrégation (*moyenne*, *maximum*, etc.). Nous choisissons d'évaluer ces deux approches sur deux types de hiérarchies : les hiérarchies simples et les hiérarchies complexes, puisque ces deux approches supposent que le traitement de l'additivité s'adapte à tout type de hiérarchie. Dans toutes ces expériences, le nombre de faits varie de 50 000 à 250 000 (Tableau 5.4).

#### Hiérarchies simples

Dans une première série d'expériences, nous testons si `QBS` est susceptible de passer à l'échelle dans le cas de hiérarchies simples. Pour cela, nous mesurons les temps d'exécution des quatre requêtes du Tableau 5.3 quand le nombre de faits (de 50 000 à 250 000) et la complexité des requêtes (1D à 4D) varient.

Dans une deuxième série d'expériences, nous vérifions que la surcharge due à l'exécution en temps réel de `QBS` n'est pas trop importante par rapport à celle de `Pedersen`. Pour atteindre cet objectif, nous comparons les temps d'exécution des quatre requêtes du Tableau 5.3 avec `QBS` et `Pedersen` en fonction du nombre de faits (de 50 000 à 250 000) et de la complexité des requêtes (1D à 4D). Pour `Pedersen`, nous distinguons deux cas : `Pedersen sans surcharge` et `Pedersen avec surcharge`. La surcharge de `Pedersen` est le traitement de l'additivité effectué avant l'exécution des requêtes utilisateur (application de *MakeCovering* et *MakeStrict*). En revanche, cette surcharge, appliquée *a priori*, n'est prise qu'une

seule fois en compte. Dans QBS, cette surcharge est intégrée à l'exécution des requêtes et s'applique à chaque exécution.

Dans une troisième série d'expériences, nous vérifions, pour QBS et Pedersen, laquelle des deux étapes, le traitement de l'additivité (étape 1 de l'Algorithme 1) et le *matching* des groupes (étape 2 de l'Algorithme 1) est la plus consommatrice de ressources de calcul. Pour atteindre cet objectif, nous mesurons les temps de traitement de l'additivité et du *matching* des groupes de manière séparée, pour les deux approches.

### Hiérarchies complexes

Dans le cas des hiérarchies complexes, nous utilisons dans nos expériences plusieurs types de hiérarchies : non-couvrantes seulement, non-strictes seulement et complexes.

Dans une première série d'expériences, nous vérifions que la surcharge due à l'exécution en temps réel de QBS, n'est pas trop importante pour des données organisées en hiérarchies non-couvrantes. Pour cela, nous mesurons le temps d'exécution des quatre requêtes du Tableau 5.3 lorsque le nombre de faits (50 000 à 250 000) et la complexité des requêtes (1D à 4D) varient.

Dans une deuxième et troisième séries d'expériences, nous avons le même objectif que la première série d'expériences, mais pour les hiérarchies non-strictes et les hiérarchies complexes, respectivement. Pour cela, nous mesurons les temps d'exécution des quatre requêtes du Tableau 5.3 lorsque le nombre de faits et la complexité des requêtes varient.

Dans une quatrième série d'expériences, nous comparons les temps de traitement des hiérarchies non-couvrantes, non-strictes et complexes. Ces expériences nous sont utiles pour mieux interpréter les résultats précédents. Pour cela, nous calculons les temps des traitements de ces hiérarchies en fonction des nombre de faits et de dimensions exploitées par les requêtes.

### 5.3.3 Résultats expérimentaux

#### Résultats sur des hiérarchies simples

Les résultats de la Figure 5.11 montrent que le temps d'exécution des requêtes avec `QBS` augmente de façon pratiquement linéaire en fonction du nombre de faits et de la complexité de la requête (nombre de dimensions). Ce résultat indique que `QBS` est susceptible de passer à l'échelle.

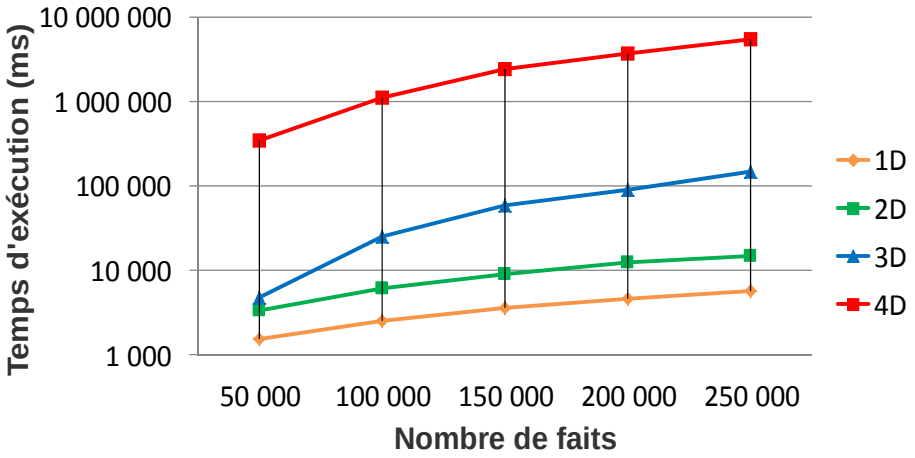



FIGURE 5.11 – Temps d'exécution des requêtes en fonction du nombre de faits et de dimensions, avec `QBS`

Les résultats de la Figure 5.11 montrent aussi que le temps d'exécution des requêtes est quatre fois supérieur en passant de 50 000 à 250 000 faits, pour les requêtes 1D et 2D. Par contre, il est quinze fois supérieur en passant de 50 000 à 250 000 faits dans le cas des requêtes 4D. Nous expliquons ce résultat par le fait que les requêtes 4D nécessitent plus de temps pour traiter les groupes (parcours, création et *matching*) correspondants aux quatre dimensions de la requête, puisque ces groupes contiennent plus d'éléments que dans les autres cas.

Les résultats de la Figure 5.12 montrent que le temps d'exécution des requêtes avec `Pedersen` augmente également de façon linéaire en fonction des nombres de faits et de dimensions exploités par les requêtes.

La comparaison du temps d'exécution des requêtes avec `QBS` au temps d'exécution des requêtes avec `Pedersen avec surcharge` montre que `QBS` permet un gain

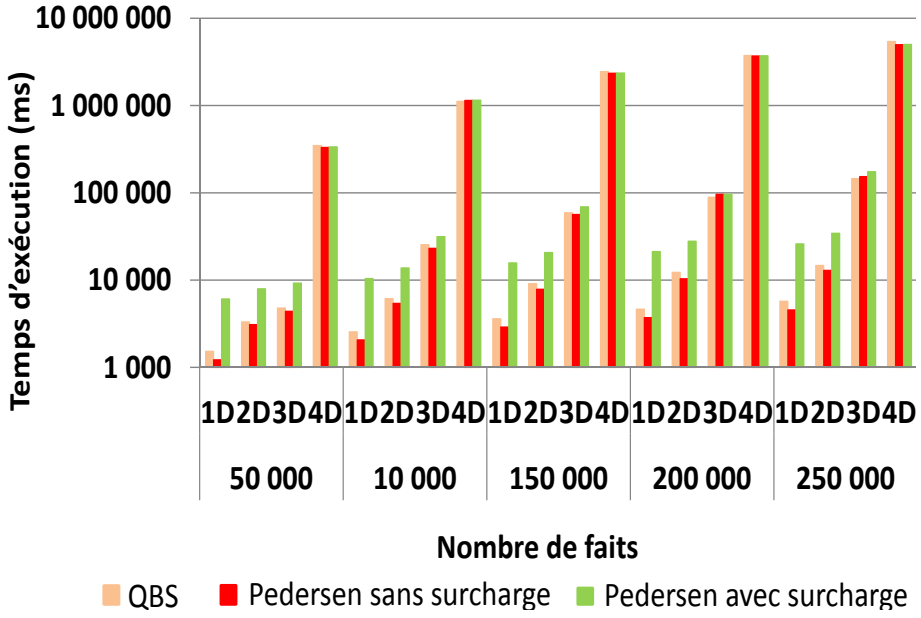



FIGURE 5.12 – Comparaison de `QBS` et `Pedersen` sur des hiérarchies simples

de performance moyen de 117 % par rapport à `Pedersen`. Ce résultat confirme que la surcharge de `QBS` est plus légère. Le meilleur gain de performance est réalisé pour les requêtes 1D, où `QBS` permet un gain de performance moyen de 310 % par rapport à `Pedersen`. Par contre, la comparaison du temps d'exécution des requêtes 4D avec `QBS` au temps d'exécution des requêtes 4D avec `Pedersen avec surcharge` montre que `QBS` est 2 % moins rapide que `Pedersen`.

La comparaison du temps d'exécution des requêtes avec `QBS` au temps d'exécution des requêtes avec `Pedersen sans surcharge` montre que `QBS` est 17 % moins rapide que `Pedersen sans surcharge`, en moyenne. En fait, pour les requêtes simples (1D et 2D), le temps de `QBS` est toujours supérieur au temps de `Pedersen sans surcharge`. Par contre, pour les requêtes les plus complexes (3D et 4D), `QBS` arrive au même niveau que `Pedersen sans surcharge`, en moyenne. La comparaison du temps d'exécution des requêtes 4D avec `QBS` au temps d'exécution des requêtes 4D avec `Pedersen sans surcharge` montre que `QBS` est seulement 3 % moins rapide que `Pedersen sans surcharge`, en moyenne. Cette perte de 17 % est alors due au poids des requêtes 1D et 2D. En général, le temps d'exécution de `QBS` est comparable à celui de `Pedersen` pour les requêtes les plus complexes,

donc la surcharge de QBS est acceptable.

Dans ces expériences (Figures 5.11 et 5.12), le temps d'exécution des requêtes est relativement élevé (autour d'une heure). Afin de mieux comprendre les raisons de ces faibles performances, nous avons mené des expériences complémentaires où nous calculons les temps de traitement de l'additivité et du *matching* des groupes de manière séparée, pour QBS et Pedersen.

Pour QBS, nous évaluons trois cas (Figure 5.13) :

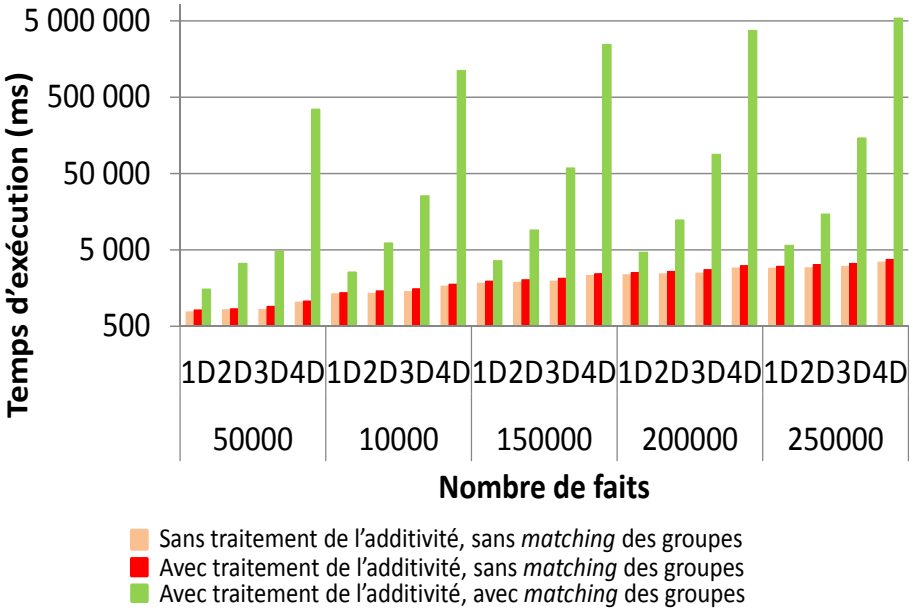



FIGURE 5.13 – Temps de traitement de l'additivité et du *matching* des groupes dans QBS

- **cas 1** : sans traitement de l'additivité, sans *matching* des groupes ;
- **cas 2** : avec traitement de l'additivité, sans *matching* des groupes ;
- **cas 3** : avec traitement de l'additivité, avec *matching* des groupes.

En comparant les cas 1 et 2, c'est-à-dire lorsque le temps de traitement de l'additivité est considéré, le temps d'exécution des requêtes n'augmente presque pas. Par contre, en comparant les cas 2 et 3, c'est-à-dire lorsque le temps du *matching* des groupes est considéré, le temps d'exécution des requêtes augmente de 237 % en moyenne. Nous concluons donc que le traitement du *matching* des groupes prend beaucoup plus de temps que le traitement de l'additivité dans les

requêtes avec QBS.

Pour Pedersen, nous évaluons également trois cas (Figure 5.14) :

- **cas 1** : sans surcharge (c'est-à-dire sans traitement de l'additivité), sans *matching* des groupes ;
- **cas 2** : sans surcharge, avec *matching* des groupes ;
- **cas 3** : avec surcharge, avec *matching* des groupes.

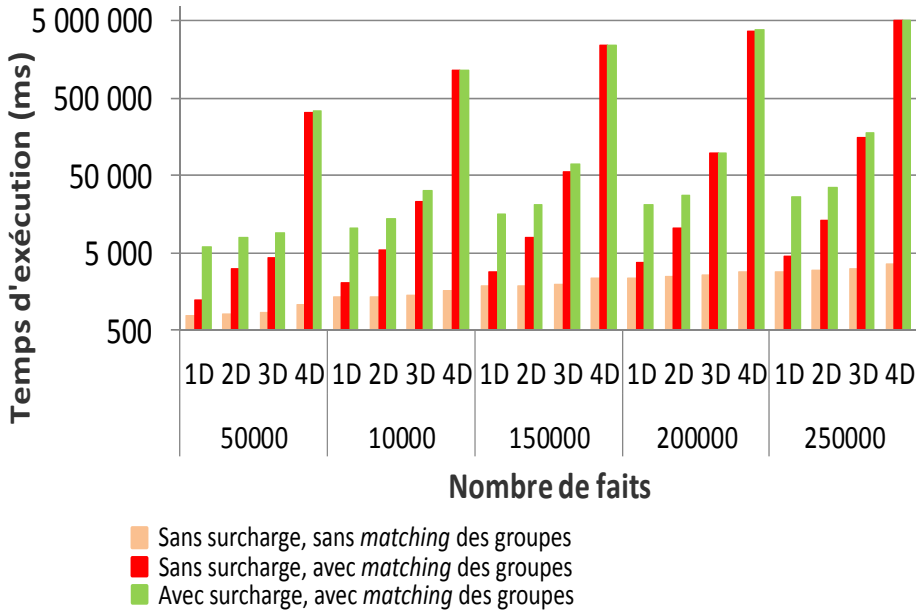



FIGURE 5.14 – Temps de traitement de l'additivité (surcharge) et du *matching* des groupes dans Pedersen

En comparant les cas 1 et 2, c'est-à-dire lorsque le *matching* des groupes est considéré, le temps d'exécution des requêtes augmente de 244 % en moyenne. Par contre, en comparant les cas 2 et cas 3, c'est-à-dire lorsque traitement de l'additivité est considéré, ce temps augmente de 2 % en moyenne. Nous concluons donc que, comme dans le cas des requêtes avec QBS, le traitement du *matching* des groupes prend plus de temps que le traitement de l'additivité. Même si nous sommes dans le cas de hiérarchies simples, ce résultat s'explique par le fait que le *matching* des groupes est une opération coûteuse qui inclut la recherche des groupes dans tous les arbres du résultat final, le calcul d'agrégations et la mise à jour de ces arbres (ou la création de nouveaux arbres dans le résultat). Ce

problème est général pour tout calcul de regroupement sur des données XML, et donc *a fortiori* dans un contexte XOLAP.

### Résultats sur des hiérarchies complexes

**Hiérarchies non-couvrantes.** Les résultats de ces expériences sont illustrés dans les Figures 5.15 et 5.16. Les résultats obtenus pour les requêtes à partir de 4D, longues à exécuter, et leur interprétation étant les mêmes que pour les requêtes 3D, nous ne les intégrons pas dans les Figures 5.15 et 5.16, ni dans le reste des expériences de ce mémoire.

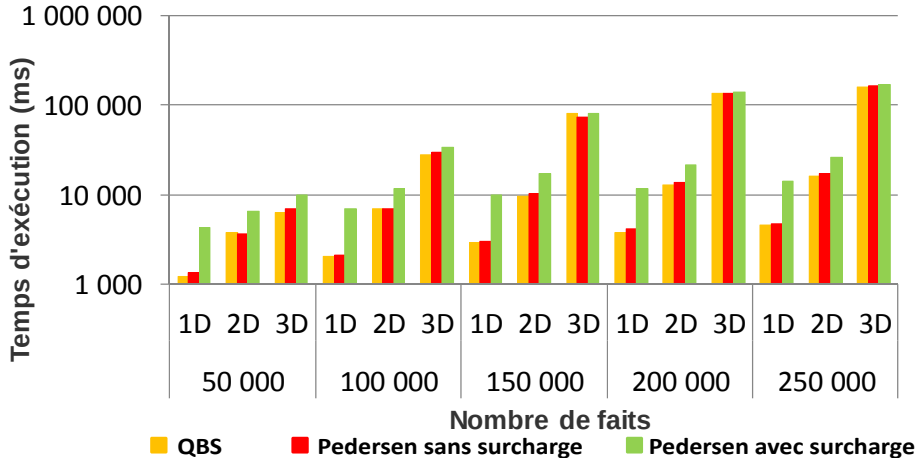



FIGURE 5.15 – Comparaison de `QBS` et `Pedersen` sur des hiérarchies non-couvrantes de type 5 %

La comparaison du temps d'exécution des requêtes avec `QBS` au temps d'exécution des requêtes avec `Pedersen avec surcharge` montre que `QBS` permet un gain de performance moyen de 90 % sur les hiérarchies non-couvrantes de type 5 %. Ce gain de performance moyen passe à 91 % sur les hiérarchies non-couvrantes de type 50 %. Ces résultats montrent que la surcharge de `QBS` est toujours légère, même si nous multiplions par dix le nombre de hiérarchies non-couvrantes. Nous expliquons ce résultat par le fait que le traitement de l'additivité dans `QBS` (remplacement des valeurs manquantes par "Autre") est plus simple que celui dans `Pedersen` (remplacement des valeurs manquantes par les valeurs extraites de métadonnées).

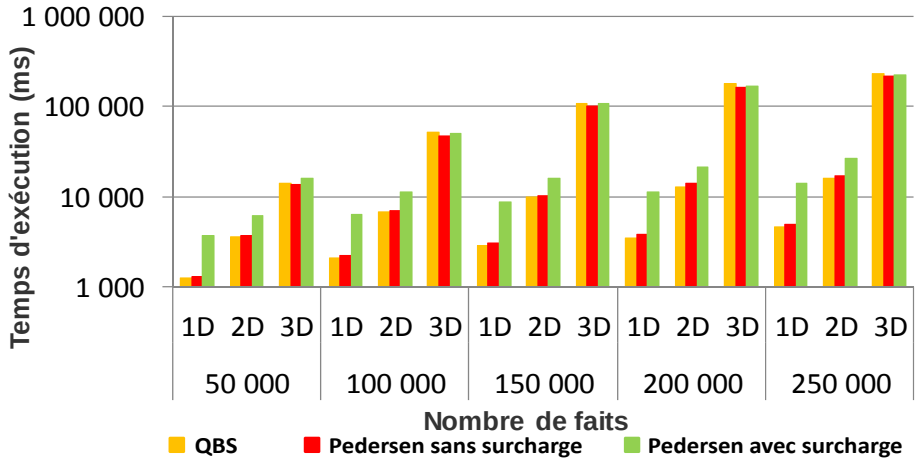



FIGURE 5.16 – Comparaison de `QBS` et `Pedersen` sur des hiérarchies non-couvrantes de type 50 %

La comparaison du temps d'exécution des requêtes avec `QBS` au temps d'exécution des requêtes avec `Pedersen sans surcharge` montre que `QBS` permet des gains de performance moyens de 4 % sur les hiérarchies non-couvrantes de type 5 % et de 2 % sur les hiérarchies non-couvrantes de type 50 %, respectivement. Donc, et même quand la surcharge n'est pas prise en compte dans les requêtes avec `Pedersen`, notre approche reste viable avec le temps de réponse du même ordre que ceux de `Pedersen`. Ceci est dû au fait que `QBS` ne prend pas beaucoup de temps pour remplacer les valeurs manquantes par "Autre" dans les hiérarchies non-couvrantes.

Dans le cas des requêtes 3D, lorsque la surcharge n'est pas considérée dans les requêtes avec `Pedersen`, `QBS` permet le même gain de performance sur les requêtes de type 5 %. Par contre, nous obtenons une perte de performance moyenne de 6 % sur les requêtes de type 50 %. Ceci s'explique par le fait que le *matching* des groupes devient une opération plus complexe lorsque le nombre de hiérarchies augmente.

Par contre, lorsque la surcharge est considérée dans les requêtes 3D avec `Pedersen`, `QBS` permet des gains de performance moyens de 18 % sur les hiérarchies de type 5 % et de 6 % sur les hiérarchies de type 50 %, respectivement. Ces résultats confirment de nouveau que l'exécution en temps réel de `QBS` n'engendre

pas une surcharge trop importante, même pour les requêtes les plus complexes (3D et 50 %).

**Hiérarchies non-strictes et Hiérarchies complexes.** Les expériences que nous avons menées sur les hiérarchies non-strictes (Figures 5.17 et 5.18) et les hiérarchies complexes (Figures 5.19 et 5.20) confirment les résultats que nous avons obtenus sur les hiérarchies non-couvrantes. En effet, la surcharge due à l'exécution en temps réel de `QBS` est acceptable dans ces deux cas.

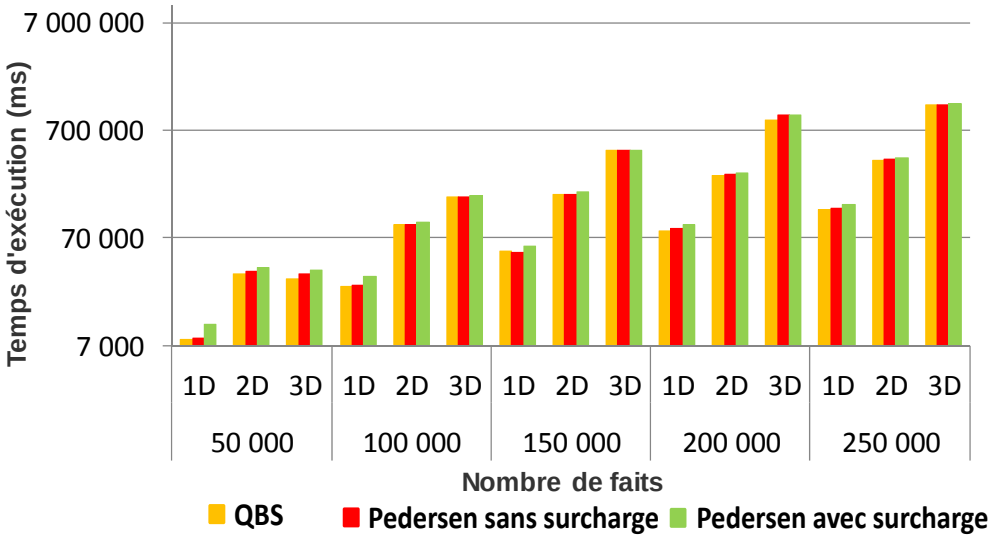



FIGURE 5.17 – Comparaison de `QBS` et `Pedersen` sur des hiérarchies non-strictes de type 5 %

`QBS` permet des gains de performance moyens de 11 % pour les hiérarchies non-strictes de type 5 %, et de 3 % pour les hiérarchies non-strictes de type 50 %, par rapport aux requêtes avec `Pedersen avec surcharge`. D'autre part, `QBS` permet des gains de performance moyens de 5 % pour les hiérarchies non-strictes de type 5 %, et 1 % pour les hiérarchies non-strictes de type 50 %, avec `Pedersen sans surcharge`.

Dans le cas des requêtes 3D sur des hiérarchies non-strictes de type 5 %, nous obtenons un gain de performance de 2 % lorsque la surcharge de `Pedersen` est prise en compte, et un gain de performance de 1 % dans le cas contraire. Dans le cas des requêtes 3D sur des hiérarchies non-strictes de type 50 %, nous obtenons

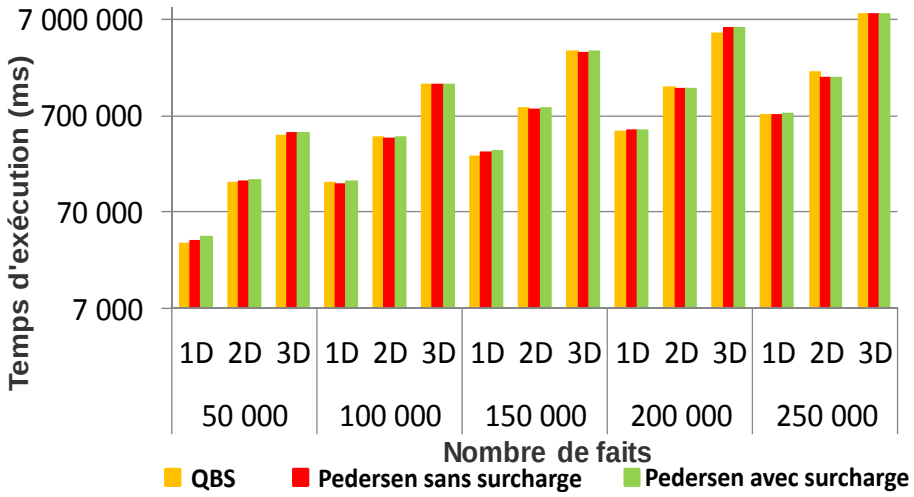



FIGURE 5.18 – Comparaison de `QBS` et `Pedersen` sur des hiérarchies non-strictes de type 50 %

un gain de performance de 3 % lorsque la surcharge de `Pedersen` est prise en compte, et un gain de performance de 2 % dans le cas contraire.

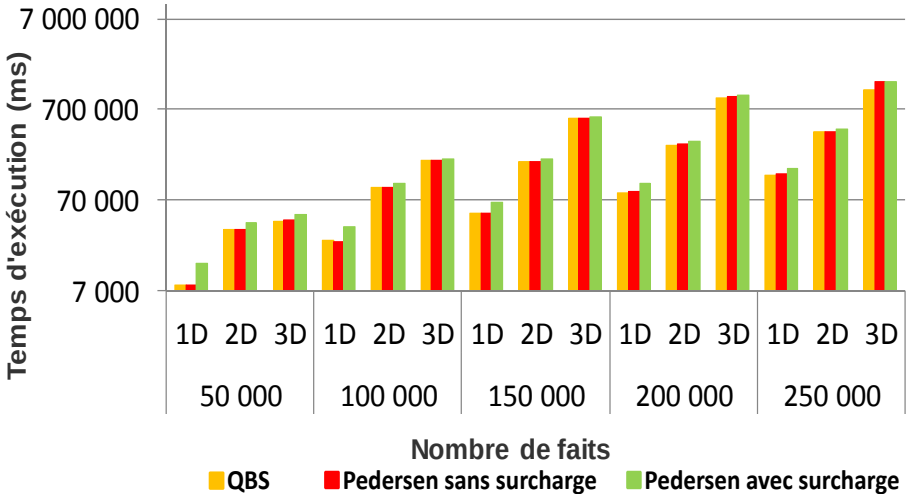



FIGURE 5.19 – Comparaison de `QBS` et `Pedersen` sur des hiérarchies complexes de type 5 %

Par ailleurs, `QBS` permet des gains de performance moyens de 18 % pour les hiérarchies complexes de type 5 %, et de 9 % pour les hiérarchies complexes de

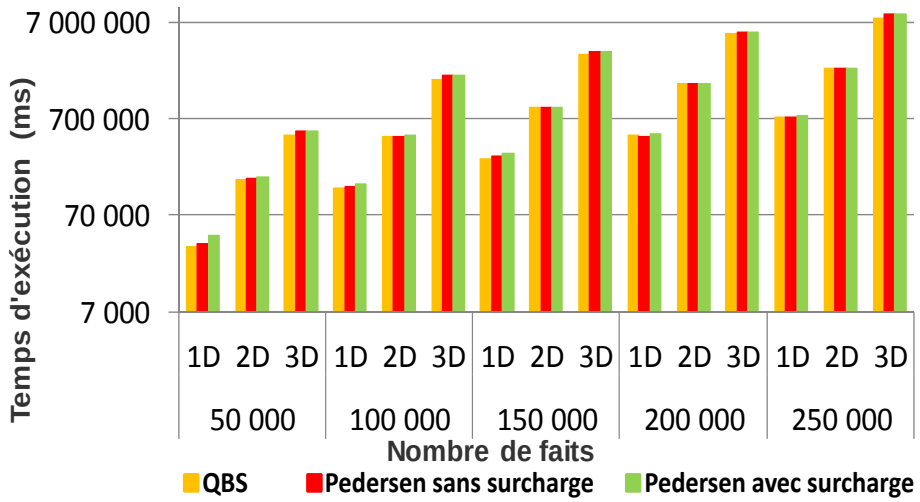



FIGURE 5.20 – Comparaison de `QBS` et `Pedersen` sur des hiérarchies complexes de type 50 %

type 50 %, lorsque la surcharge de `Pedersen` est considérée. Il permet des gains de performance moyens de 1 % pour les hiérarchies complexes de type 5 %, et de 5 % pour les hiérarchies complexes de type 50 % dans le cas contraire.

Dans le cas des requêtes 3D sur des hiérarchies complexes de type 5 %, nous obtenons un gain de performance de 10 % lorsque la surcharge de `Pedersen` est prise en compte, et un gain de performance de 5 % dans le cas contraire. Dans le cas des requêtes 3D sur des hiérarchies complexes de type 50 %, nous obtenons un gain de performance de 3 % lorsque la surcharge de `Pedersen` est prise en compte, et un gain de performance de 2 % dans le cas contraire.

En conclusion, l'ensemble de ces résultats sur les hiérarchies non-strictes et complexes confirment que l'exécution en temps réel de `QBS` n'est pas trop importante par rapport à celle de `Pedersen`. Elles confirment aussi que le gain obtenu dans le cas de hiérarchies complexes est toujours meilleur que le gain obtenu dans le cas des hiérarchies non-strictes. Dans ce qui suit, nous menons de nouvelles expériences pour expliquer ces résultats.

**Évaluation des trois types de hiérarchies dans les requêtes avec `QBS`.** Les résultats obtenus dans les Figures 5.17, 5.18, 5.19 et 5.20 montrent que le temps d'exécution des requêtes sur des hiérarchies non-strictes et complexes est sept fois

supérieur au temps de traitement des hiérarchies non-couvrantes (Figures 5.15 et 5.16). De plus, le gain de performance moyen des requêtes avec `QBS` par rapport aux requêtes avec `Pedersen avec surcharge` sur les hiérarchies non-couvrantes de type 50 % que nous avons obtenu (91 %) est largement supérieur aux gains obtenus pour les hiérarchies non-strictes et complexes, respectivement.

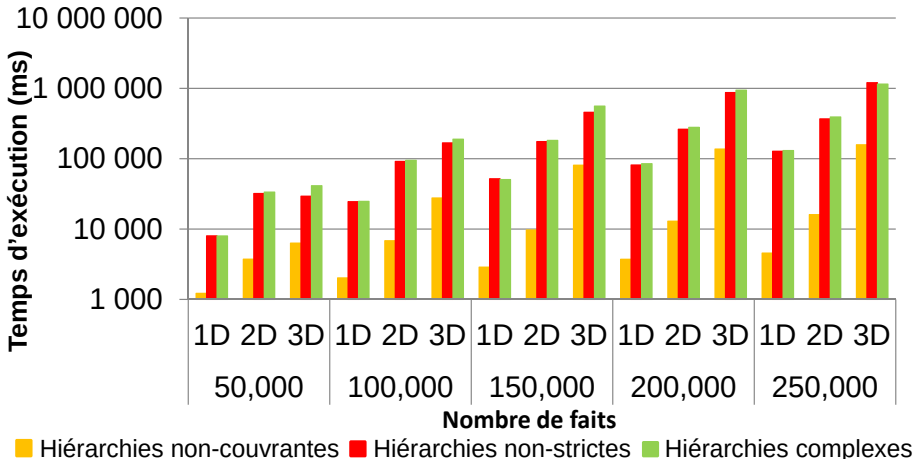



FIGURE 5.21 – Évaluation des trois types de hiérarchies de type 5 % dans les requêtes avec `QBS`

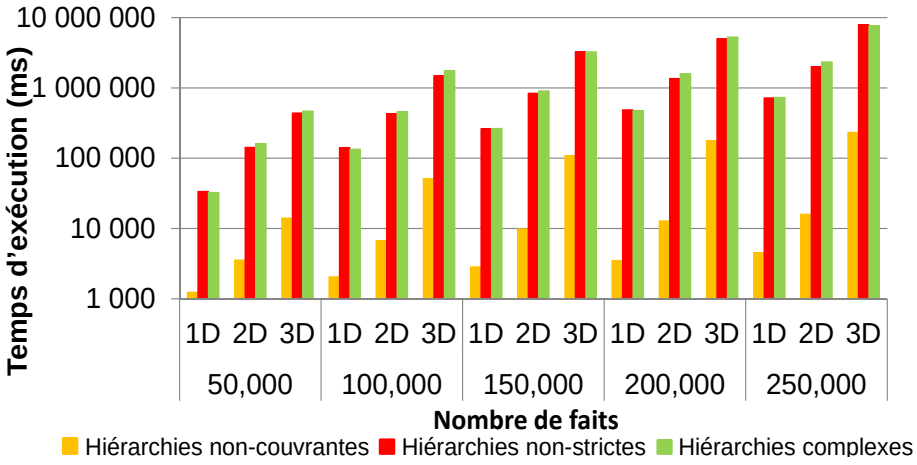



FIGURE 5.22 – Évaluation des trois types de hiérarchies de type 50 % dans les requêtes avec `QBS`

Pour expliquer ces résultats, nous avons calculé les temps de traitement des trois types de hiérarchies (non-couvrantes, non-strictes et complexes) avec `QBS`, en fonction des nombres de faits et de dimensions. Les résultats de ces expériences sont illustrés dans les Figures 5.21 et 5.22.

Dans le cas des hiérarchies de type 5 %, le temps nécessaire au traitement des hiérarchies non-couvrantes est neuf fois supérieur au temps de traitement des hiérarchies non-strictes, en moyenne. Par ailleurs, le temps de traitement des hiérarchies non-strictes est pratiquement le même que le temps de traitement des hiérarchies complexes, en moyenne.

Dans le cas des hiérarchies de type 50 %, le temps de traitement des hiérarchies non-couvrantes est trente sept fois supérieur au temps de traitement des hiérarchies non-strictes en moyenne. Par ailleurs, le temps de traitement des hiérarchies non-strictes est pratiquement le même que le temps de traitement des hiérarchies complexes en moyenne.

Ces résultats confirment que le temps du traitement des hiérarchies non-strictes est prépondérant dans le temps d'exécution de la requête. Ceci s'explique par le fait que le processus de création de groupes dans les hiérarchies non-strictes nécessite plus de temps que le remplacement des valeurs manquantes par la valeur artificielle "Autre" dans les hiérarchies non-couvrantes.

En conclusion, le fait que le gain obtenu dans le cas des hiérarchies complexes est toujours meilleur que le gain obtenu dans les hiérarchies non-strictes est dû à la simplicité de traitement des hiérarchies non-couvrantes par rapport au traitement des hiérarchies non-strictes.

# Chapitre 6

# Conclusion et perspectives

## 6.1 Bilan général

Les travaux présentés dans ce mémoire ont été réalisés au sein du laboratoire ERIC [1], dont l'activité de recherche est principalement dédiée à deux domaines : (1) les entrepôts de données (la modélisation multidimensionnelle et l'intégration intelligente des données complexes, l'analyse en ligne personnalisée et la sécurité du processus d'entreposage) ; et (2) la fouille de données (apprentissage automatique, fouille et analyse de données complexes, fouille d'opinion, etc.). Les recherches menées dans ce mémoire dépendent du premier domaine. Elle relèvent plus particulièrement de l'équipe SID (Systèmes d'Information Décisionnels) dont l'un des principaux thèmes de recherche est la présentation de nouvelles solutions OLAP pour les entrepôts de données complexes XML (solutions XOLAP). Dans ce contexte, nous nous sommes plus particulièrement intéressés aux erreurs d'analyse que peut rencontrer l'utilisateur dans ses analyses OLAP, et qui sont dues à des problèmes d'additivité. Nous avons conjointement abordé dans ces travaux la problématique de modélisation efficace des données multidimensionnelles et des hiérarchies complexes, et proposé une nouvelle solution en temps réel pour les problèmes d'additivité.

Dans un premier temps, nous nous sommes intéressés à la proposition d'un nouveau modèle de données pour modéliser les données multidimensionnelles et, surtout, les hiérarchies complexes (Section 4.3). Comme notre solution s'inscrit

1. `http://eric.univ-lyon2.fr`

dans un cadre XOLAP, nous avons proposé une adaptation du modèle multidimensionnel classique (faits, dimensions, mesures et hiérarchies) aux arbres XML. Ce choix est motivé par le fait que les arbres XML représentent le modèle de données le plus simple et le plus utilisé dans la majorité des travaux de recherche sur XML. Nous avons appelé notre modèle de données les arbres de données multidimensionnels. Les arbres de données multidimensionnels offrent surtout la possibilité de modéliser, de manière simple, les hiérarchies complexes.

Nous avons ensuite proposé un modèle de requêtes, basé sur un modèle d'arbre, pour les arbres de données multidimensionnelles (Section 4.4). Dans l'étude que nous avons menée sur les modèles d'arbres XML, nous avons déduit que l'efficacité d'un modèle d'arbre dépend essentiellement de sa simplicité et des options de *matching* associées à ses nœuds et/ou à ses arêtes [HD12]. Nous avons donc associé au modèle d'arbre de base [JLST01] les options de *matching* que nous avons jugées nécessaires au parcours des hiérarchies complexes, au regroupement et au calcul d'agrégation. À notre connaissance, notre modèle d'arbre est le premier modèle d'arbre pour données multidimensionnelles [HKD12].

Dans une troisième contribution, nous avons proposé un nouvel algorithme de regroupement et d'agrégation pour la résolution des problèmes d'additivité dans les hiérarchies complexes (Section 4.5). En entrée de notre algorithme, les données multidimensionnelles sont modélisées par des arbres de données multidimensionnels et les requêtes utilisateur sont exprimées à l'aide de notre modèle d'arbre pour données multidimensionnelles. Le but de notre algorithme est de parcourir les arbres de données multidimensionnels et de calculer les agrégations mentionnées dans la requête utilisateur de sorte que l'additivité soit garantie. Dans cet algorithme, nous employons des solutions déjà utilisées dans la littérature pour le traitement de l'additivité (les groupes et les valeurs artificielles "Autre"). Toutefois, notre solution pour les problèmes d'additivité est la première solution en temps réel qui permet non seulement de détecter les problèmes d'additivité, mais aussi de présenter à l'utilisateur des résultats d'agrégation corrects. Sur la base de cet algorithme, nous proposons un nouvel opérateur XOLAP de forage vers le haut (*roll-up*).

Le développement de nos méthodes inclut une phase de validation expérimentale. Pour la mener à bien, nous avons étendu le banc d'essais XWeB en introduisant des hiérarchies complexes paramétrables dans son schéma et en ajoutant à sa charge

des requêtes de regroupement et d'agrégation qui portent sur ces hiérarchies. Cela nous a permis de comparer notre approche à l'approche de référence de Pedersen *et al.* [PJD99] en fonction du nombre de dimensions dans la requête et de la taille de données. Les fichiers utilisés dans ces expériences sont générés à l'aide de XWeB. Les résultats obtenus montrent que notre approche est susceptible de passer à l'échelle et que sa surcharge due à l'exécution en temps réel n'est pas trop importante par rapport à la surcharge de Pedersen *et al.* De plus, la surcharge nécessaire au traitement des hiérarchies complexes est tout à fait acceptable et valide l'idée qu'une solution en temps réel est viable, et même préférable lorsque les données et/ou le schéma évoluent.

## 6.2 Perspectives de recherche

Même si l'utilisation de XML représente la meilleure solution pour modéliser les hiérarchies complexes [BCC+05], les expériences que nous avons menées montrent que le traitement de l'additivité est une opération coûteuse, surtout pour les scénarios OLAP réels, en raison du coût élevé du *matching* des groupes qui augmente avec la taille des données. Par conséquent, notre première perspective est d'optimiser la performance de notre approche par le stockage des données dans un autre format que XML et/ou employer des solutions efficaces de stockage, d'indexation et de traitement du *matching* des groupes.

Plus précisément, nous comptons proposer dans nos travaux futurs une approche holistique pour résoudre les problèmes d'additivité dans les hiérarchies complexes. En premier lieu, nous comptons étiqueter les nœuds des arbres de données multidimensionnels avec un étiquetage Dewey (Section 3.4.5) qui permet d'identifier, à partir de l'étiquette d'un nœud donné, les étiquettes de ses nœuds ancêtres et descendants, ainsi que leurs positions dans l'arbre de données. De plus, la comparaison des étiquettes correspondant à deux nœuds de même type dans l'arbre de données peut nous indiquer si la hiérarchie est complexe ou pas.

En effet, si un nœud $x$ est étiqueté avec un étiquette Dewey (1.1, par exemple) qui contient moins d'éléments que l'étiquette Dewey d'un nœud $y$ (2.1.1, par exemple) de même type que $x$, alors la hiérarchie contenant le nœud $x$ est non-couvrante. Dans cet exemple, nous avons obtenu ce résultat puisque la comparaison entre ces deux étiquettes signifient que $x$ est situé au troisième niveau de l'arbre

et que $y$ est situé au quatrième niveau de l'arbre (Section 3.4.5). Par ailleurs, si nous détectons que deux nœuds $x$ et $y$ de même type ont le même nœud parent après analyse de leurs étiquettes Dewey, nous pouvons conclure que la hiérarchie en question est non-stricte. Lors du processus de *matching* des groupes, nous pouvons également utiliser un étiquetage Dewey dynamique [XLWB09] pour déterminer les étiquettes des nouveaux nœuds à insérer (groupes et valeurs artificielles "Autre").

Par ailleurs, l'objectif initial de cette thèse était de concevoir une algèbre XOLAP. Nous gardons cet objectif en ligne de mire puisque nous comptons proposer, dans nos travaux futurs, d'autres opérateurs XOLAP (forage vers le bas, cube, etc.) qui formeront le noyau d'une algèbre XOLAP. Ces opérateurs doivent aussi prendre en compte les problèmes d'additivité dans les hiérarchies complexes, si nécessaire, en reprenant les principes introduits dans cette thèse.

Enfin, nous comptons compléter notre travail d'implémentation avec ces nouveaux opérateurs. Notre objectif est d'intégrer ces programmes dans une solution XOLAP logicielle complète qui puisse s'interfacer à tous les SGBD compatibles et natifs XML. Plus concrètement, nous comptons implémenter un logiciel qui offre à l'utilisateur une interface facile à utiliser qui lui permette de charger des données XML et de choisir les opérations XOLAP nécessaires à l'analyse de ces données.

# Bibliographie


[ABM+02] Abiteboul S., Benjelloun O., Manolescu I., Milo T., Weber R., «Active XML : Peer-to-Peer Data and Web Services Integration», in *28th International Conference on Very Large Data Bases (VLDB 02), Hong Kong, China*, pp. 1087–1090, Morgan Kaufmann. 2002.

[ABM09] Abiteboul S., Bourhis P., Marinoiu B., «Satisfiability and Relevance for Queries over Active Documents», in *28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 09), Providence, USA*, pp. 87–96, ACM. 2009.

[AGS97] Agrawal R., Gupta A., Sarawagi S., «Modeling Multidimensional Databases», in *13th International Conference on Data Engineering (ICDE 97), Birmingham, UK*, pp. 232–243, IEEE Computer Society. 1997.

[ALAA06] Aghili S.A., Li H.G., Agrawal D., Abbadi A.E., «TWIX : twig structure and content matching of selective queries using binary labeling», in *1st International Conference on Scalable Information Systems (INFOSCALE 06), Hong Kong, China*, volume 152 de *ACM International Conference Proceeding Series*, p. 42, ACM. 2006.

[AS94] Agrawal R., Srikant R., «Fast Algorithms for Mining Association Rules in Large Databases», in *20th International Conference on Very Large Data Bases (VLDB 94), Santiago de Chile, Chile*, pp. 487–499. 1994.

[ASS06] Abelló A., Samos J., Saltor F., «YAM$^2$ : a multidimensional conceptual model extending UML», *Inf. Syst.*, 31(6) :541–567. 2006.

[AYCLS01] Amer-Yahia S., Cho S., Lakshmanan L.V.S., Srivastava D., «Minimization of Tree Pattern Queries», in *20th ACM SIGMOD International Conference on Management of Data (SIGMOD 01), Santa Barbara, California, USA*, pp. 497–508. 2001.

[AYCS02] Amer-Yahia S., Cho S., Srivastava D., «Tree Pattern Relaxation», in *8th International Conference on Extending Database Technology (EDBT 02), Prague, Czech Republic*, volume 2287 de *LNCS*, pp. 496–513. 2002.

[BB03] Baril X., Bellahsène Z., *Designing and Managing an XML Warehouse*, pp. 455–473, XML Data Management : Native XML and XML-enabled Database Systems, Addison Wesley. 2003.

[BBCA06] Boussaid O., BenMessaoud R., Choquet R., Anthoard S., «X-Warehousing : An XML-Based Approach for Warehousing Complex Data», in *10th East-European Conference on Advances in Databases and Information Systems (ADBIS 06), Thessaloniki, Greece*, volume 4152 de *LNCS*, pp. 39–54, Springer. 2006.

[BBLR06] BenMessaoud R., Boussaid O., Loudcher-Rabaséda S., «A Data Mining-Based OLAP Aggregation of Complex Data : Application on XML Documents», *International Journal of Data Warehousing & Mining*, 2(4) :1–26. 2006.

[BC04] Borkar V.R., Carey M.J., «Extending XQuery for Grouping, Duplicate Elimination, and Outer Joins», in *XML 2004 Conference, Washington, DC, USA*. November 2004.

[BC09] Benedikt M., Cheney J., «Schema-Based Independence Analysis for XML Updates», *PVLDB*, 2(1) :61–72. 2009.

[BCC+05] Beyer K.S., Chamberlin D.D., Colby L.S., Ozcan F., Pirahesh H., Xu Y., «Extending XQuery for Analytics», in *24th ACM SIGMOD International Conference on Management of Data (SIGMOD 05), Baltimore, Maryland, USA*, pp. 503–514. 2005.

[Ben06] BenMessaoud R., *Couplage de l'analyse en ligne et de la fouille de données pour l'exploration, l'agrégation et l'explication des données complexes*, Thèse de Doctorat, Université Lumière Lyon 2, France. 2006.

[BFG08] Benedikt M., Fan W., Geerts F., «XPath Satisfiability in the Presence of DTDs», *J. ACM*, 55(2). 2008.

[BFLC03] Bose S., Fegaras L., Levine D., Chaluvadi V., «A Query Algebra for Fragmented XML Stream Data», in *9th International Workshop on Database Programming Languages (DBPL 03), Potsdam, Germany*, volume 2921 de *LNCS*, pp. 195–215, Springer. 2003.

[BKS02] Bruno N., Koudas N., Srivastava D., «Holistic Twig Joins : Optimal XML Pattern Matching», in *2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 02), Madison, Wisconsin, USA*, pp. 310–321, ACM. 2002.

[BKS08] Baca R., Krátký M., Snásel V., «On the Efficient Search of an XML Twig Query in Large DataGuide Trees», in *12th International Database Engineering and Applications Symposium (IDEAS 08), Coimbra, Portugal*, volume 299 de *ACM International Conference Proceeding Series*, pp. 149–158, ACM. 2008.

[BL05] Bordawekar R., Lang C.A., «Analytical Processing of XML Documents : Opportunities and Challenges», *SIGMOD Record*, 34(2) :27–32. 2005.

[BOB+04] Balmin A., Ozcan F., Beyer K.S., Cochrane R., Pirahesh H., «A Framework for Using Materialized XPath Views in XML Query Processing», in *30th International Conference on Very Large Data Bases (VLDB 04), Toronto, Canada*, pp. 60–71, Morgan Kaufmann. 2004.

[BR99] Beyer K.S., Ramakrishnan R., «Bottom-Up Computation of Sparse and Iceberg CUBEs», in *1999 ACM SIGMOD International Conference on Management of Data (SIGMOD 99), Philadelphia, Pennsylvania, USA*, pp. 359–370, ACM Press. 1999.

[BT99] Beeri C., Tzaban Y., «SAL : An Algebra for Semistructured Data and XML», in *WebDB (Informal Proceedings), Philadelphia, USA*, pp. 37–42. 1999.

[CC06] Chen Y., Che D., «Efficient Processing of XML Tree Pattern Queries», *Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII)*, 10(5) :738–743. 2006.

[CC08] Chen D., Chan C.Y., «Minimization of Tree Pattern Queries with Constraints», in *2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 08), Vancouver, BC, Canada*, pp. 609–622, ACM. 2008.

[CCS93] Codd E., Codd S., Salley C., «Providing OLAP (On-line Analytical Processing) to User-Analysts : An IT Mandate», White paper, E.F. Codd Associates. 1993.

[CFF+02] Chan C.Y., Fan W., Felber P., Garofalakis M.N., Rastogi R., «Tree Pattern Aggregation for Scalable XML Data Dissemination», in *28th International Conference on Very Large Data Bases (VLDB 02), Hong Kong, China*, pp. 826–837, Morgan Kaufmann. 2002.

[CFG07] Chand R., Felber P., Garofalakis M.N., «Tree-Pattern Similarity Estimation for Scalable Content-based Routing», in *23rd International Conference on Data Engineering (ICDE 07), Istanbul, Turkey*, pp. 1016–1025, IEEE. 2007.

[Che04] Chen Y., «Discovering Ordered Tree Patterns from XML Queries», in *8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 04), Sydney, Australia*, volume 3056 de *Lecture Notes in Computer Science*, pp. 559–563, Springer. 2004.

[CJLP03] Chen Z., Jagadish H.V., Lakshmanan L.V.S., Paparizos S., «From Tree Patterns to Generalized Tree Patterns : On Efficient Evaluation of XQuery», in *29th International Conference on Very Large Data Bases (VLDB 03), Berlin, Germany*, pp. 237–248. 2003.

[CL05] Che D., Liu Y., «Efficient Minimization of XML Tree Pattern Queries», in *1st International Conference on Next Generation Web Services Practices (NWeSP 05), Seoul, Korea*, IEEE Computer Society. 2005.

[CLT+06] Chen S., Li H.G., Tatemura J., Hsiung W.P., Agrawal D., Candan K.S., «Twig$^2$Stack : Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents», in *32nd International Conference on Very Large Data Bases (VLDB 06), Seoul, Korea*, pp. 283–294, ACM. 2006.

[CT97] Cabibbo L., Torlone R., «Querying Multidimensional Databases», in *6th International Workshop of Database Programming Languages (DBPL 97), Estes Park, USA*, volume 1369 de *LNCS*, pp. 319–335, Springer. 1997.

[CT98] Cabibbo L., Torlone R., «A Logical Approach to Multidimensional Databases», in *6th International Conference on Extending Database Technology (EDBT 98), Valencia, Spain*, volume 1377 de *LNCS*, pp. 183–197, Springer. 1998.

[CT00] Cabibbo L., Torlone R., «The Design and Development of a Logical System for OLAP», in *2nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK 00), London, UK*, volume 1874 de *Lecture Notes in Computer Science*, pp. 1–10, Springer. 2000.

[DBRA05] Darmont J., Boussaid O., Ralaivao J.C., Aouiche K., «An Architecture Framework for Complex Data Warehouses», in *7th International Conference on Enterprise Information Systems (ICEIS 05), Miami, USA*, pp. 370–373. 2005.

[DPJ03] Dyreson C.E., Pedersen T.B., Jensen C.S., «Incomplete Information in Multidimensional Databases», in Rafanelli M., editor, *Multidimensional Databases : Problems and Solutions*, pp. 282–309, Idea Group. 2003.

[DPX04] Deutsch A., Papakonstantinou Y., Xu Y., «Minimization and Group-By Detection for Nested XQueries», in *20th International Conference on Data Engineering (ICDE 04), Boston, MA, USA*, p. 839, IEEE Computer Society. 2004.

[DT99] Datta A., Thomas H., «The Cube Data Model : A Conceptual Model and Algebra for On-Line Analytical Processing in Data Warehouses», *Decision Support Systems*, 27(3) :289–301. 1999.

[Egh10] Egho E., *OLAP manipulation of XML data : generic solution and application to spoken language corpora*, Mémoire de Master, Université Lumière Lyon 2, France. 2010.

[FFM03] Flesca S., Furfaro F., Masciari E., «On the minimization of Xpath queries», in *29th International Conference on Very Large Data Bases (VLDB 03), Berlin, Germany*, pp. 153–164. 2003.

[FHP02] Frasincar F., Houben G.J., Pau C., «XAL : An Algebra For XML Query Optimization», in *13th Australasian Database Conference (ADC 02), Melbourne, Victoria*, volume 5 de *CRPIT*, Australian Computer Society. 2002.

[FSW00] Fernández M.F., Siméon J., Wadler P., «An Algebra for XML Query», in *20th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 00), New Delhi, India*, volume 1974 de *LNCS*, pp. 11–45, Springer. 2000.

[GB10] Grimsmo N., Bjørklund T.A., «Towards Unifying Advances in Twig Join Algorithms», in *21st Australasian Database Conference (ADC 10), Brisbane, Australia*, volume 104 de *CRPIT*, pp. 57–66. 2010.

[GBLP96] Gray J., Bosworth A., Layman A., Pirahesh H., «Data Cube : A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals», in *12th International Conference on Data Engineering (ICDE 96), New Orleans, Louisiana, USA*, pp. 152–159, IEEE Computer Society. 1996.

[GC07] Gou G., Chirkova R., «Efficiently Querying Large XML Data Repositories : A Survey», *IEEE Transactions on Knowledge and Data Engineering*, 19(10) :1381–1403. 2007.

[GL97] Gyssens M., Lakshmanan L.V.S., «A Foundation for Multi-dimensional Databases», in *23rd International Conference on Very Large Data Bases (VLDB 97), Athens, Greece*, pp. 106–115, Morgan Kaufmann. 1997.

[GLS96] Gyssens M., Lakshmanan L.V.S., Subramanian I.N., «Tables as a Paradigm for Querying and Restructuring», in *15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 96), Montreal, Canada*, pp. 93–103, ACM. 1996.

[GR99] Golfarelli M., Rizzi S., «Designing the Data Warehouse : Key Steps and Crucial Issues», *Journal of Computer Science and Information Management*, 2. 1999.

[GRS06] Ghelli G., Ré C., Siméon J., «XQuery ! : An XML Query Language with Side Effects», in *International Workshop on Database Technologies for Handling XML Information on the Web (DataX 06), In conjunction with EDBT 06, Munich, Germany*, volume 4254 de *Lecture Notes in Computer Science*, pp. 178–191, Springer. 2006.

[GRV01] Golfarelli M., Rizzi S., Vrdoljak B., «Data Warehouse Design from XML Sources», in *4th International ACM Workshop on Data Warehousing and OLAP (DOLAP 01), Atlanta, USA*, pp. 40–47, ACM Press. 2001.

[HBH03] Hümmer W., Bauer A., Harde G., «XCube - XML For Data Warehouses», in *6th International ACM Workshop on Data Warehousing and OLAP (DOLAP 03), New Orleans, USA*, pp. 33–40. 2003.

[HD10a] Hachicha M., Darmont J., «Modèles d'arbre pour XOLAP», in *6èmes Journées francophones sur les Entrepôts de Données et l'Analyse en ligne (EDA 10), Djerba, Tunisie*, volume B-6 de *Revues des Nouvelles Technologies de l'Information*, pp. 97–106, Toulouse : Cépaduès Editions. Juin 2010.

[HD10b] Hachicha M., Darmont J., «Pattern Tree-based XOLAP Rollup Operator for XML Complex Hierarchies», in *1st International Conference on Machine and Web Intelligence (ICMWI 10), Algiers, Algeria*. 2010.

[HD12] Hachicha M., Darmont J., «A Survey of XML Tree Patterns», *IEEE Transactions on Knowledge and Data Engineering*, in preprint. 2012.

[HGM05] Hurtado C.A., Gutiérrez C., Mendelzon A.O., «Capturing Summarizability with Integrity Constraints in OLAP», *ACM Trans. Database Syst.*, 30(3) :854–886. 2005.

[Hid04] Hidders J., «Satisfiability of XPath Expressions», in *9th International Workshop On Database Programming Languages (DBPL 03),*

*Potsdam, Germany*, volume 2921 de *Lecture Notes in Computer Science*, pp. 21–36, Springer. 2004.

[HKD12] Hachicha M., Kit C., Darmont J., «A Novel Query-Based Approach for Addressing Summarizability Issues in XOLAP», in *18th International Conference on Management of Data (COMAD 12), Pune, India*, CSI. December 2012.

[HLBS02] Hümmer W., Lehner W., Bauer A., Schlesinger L., «A Decathlon in Multidimensional Modeling : Open Issues and Some Solutions», in *4th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 02), Aix-en-Provence, France*, volume 2454 de *Lecture Notes in Computer Science*, pp. 275–285, Springer. 2002.

[HM01] Hurtado C.A., Mendelzon A.O., «Reasoning about Summarizability in Heterogeneous Multidimensional Schemas», in *8th International Conference on Database Theory (ICDT 01), London, UK*, volume 1973 de *Lecture Notes in Computer Science*, pp. 375–389, Springer. 2001.

[HMD07] Hachicha M., Mahboubi H., Darmont J., «Vers une algèbre XML-OLAP : État de l'art», in *2ème Atelier sur les Systèmes Décisionnels (ASD 07), Sousse, Tunisie*. 2007.

[HMD08] Hachicha M., Mahboubi H., Darmont J., «Expressing OLAP operators with the TAX XML algebra», in *3rd International Workshop on Database Technologies for Handling XML Information on the Web (DataX 08), In conjunction with EDBT 08, Nantes, France*. March 2008.

[HS05] Horner J., Song I.Y., «A Taxonomy of Inaccurate Summaries and Their Management in OLAP Systems», in *24th International Conference on Conceptual Modeling (ER 05), Klagenfurt, Austria*, volume 3716 de *Lecture Notes in Computer Science*, pp. 433–448, Springer. 2005.

[HSC04] Horner J., Song I.Y., Chen P.P., «An Analysis of Additivity in OLAP Systems», in *7th International ACM Workshop on Data*

*Warehousing and OLAP (DOLAP 04), Washington, DC, USA*, pp. 83–91, ACM. 2004.

[HWC06] Hsieh M.C.E., Wu Y.H., Chen A.L.P., «Discovering Frequent Tree Patterns over Data Streams», in *6th SIAM International Conference on Data Mining (SDM 06), Bethesda, MD, USA*, SIAM. 2006.

[IHH09] Izadi S.K., Härder T., Haghjoo M.S., «$S^3$ : Evaluation of tree-pattern queries supported by structural summaries», *Data & Knowledge Engineering*, 68(1) :126–145. 2009.

[Inm05] Inmon W., *Building the Data Warehouse*, John Wiley & Sons, fourth édition. 2005.

[JLS99] Jagadish H.V., Lakshmanan L.V.S., Srivastava D., «What can Hierarchies do for Data Warehouses ?», in *25th International Conference on Very Large Data Bases (VLDB 99), Edinburgh, Scotland, UK*, pp. 530–541, Morgan Kaufmann. 1999.

[JLST01] Jagadish H.V., Lakshmanan L.V.S., Srivastava D., Thompson K., «TAX : A Tree Algebra for XML», in *8th International Workshop on Database Programming Languages (DBPL 01), Frascati, Italy*, volume 2397 de *LNCS*, pp. 149–164. 2001.

[JMP01] Jensen M.R., Møller T.H., Pedersen T.B., «Specifying OLAP cubes on XML data», *Journal of Intelligent Information Systems*, 17(2-3) :255–280. 2001.

[KAK08] Kit C., Amagasa T., Kitagawa H., «Algorithms for efficient structure-based grouping in XML-OLAP», in *10th International Conference on Information Integration and Web-based Applications Services (iiWAS 08), Linz, Austria*, pp. 170–177, ACM. 2008.

[KAK09] Kit C., Amagasa T., Kitagawa H., «Algorithms for structure-based grouping in XML-OLAP», *International Journal of Web Information Systems (IJWIS)*, 5(2) :122–150. 2009.

[KHD12] Kit C., Hachicha M., Darmont J., «Benchmarking Summarizability Processing in XML Warehouses with Complex Hierarchies», in *ACM 15th International Workshop on Data Warehousing and OLAP (DOLAP 12), Maui, Hawaii, USA*. November 2012.

[Kit10] Kit C., *A Study on OLAP Analysis in XML Databases*, Thèse de Doctorat, University of Tsukuba, Japan. 2010.

[KKNR04] Kaushik R., Krishnamurthy R., Naughton J.F., Ramakrishnan R., «On the Integration of Structure Indexes and Inverted Lists», in *23rd ACM SIGMOD International Conference on Management of Data (SIGMOD 04), Paris, France*, pp. 779–790, ACM. 2004.

[KR02] Kimball R., Ross M., *The Data Warehouse Toolkit*, John Wiley & Sons, second édition. 2002.

[KS06] Kimelfeld B., Sagiv Y., «Twig Patterns : From XML Trees to Graphs», in *9th International Workshop on the Web and Databases (WebDB 06), Chicago, Illinois, USA*. 2006.

[LAW98] Lehner W., Albrecht J., Wedekind H., «Normal Forms for Multidimensional Databases», in *10th International Conference on Scientific and Statistical Database Management (SSDBM 98), Capri, Italy*, pp. 63–72, IEEE Computer Society. 1998.

[LCL04] Lu J., Chen T., Ling T.W., «Efficient Processing of XML Twig Patterns with Parent Child Edges : A Look-ahead Approach», in *13th ACM CIKM International Conference on Information and Knowledge Management (CIKM 04), Washington, DC, USA*, pp. 533–542, ACM. 2004.

[LLBW11] Lu J., Ling T.W., Bao Z., Wang C., «Extended XML Tree Pattern Matching : Theories and Algorithms», *IEEE Transactions on Knowledge and Data Engineering*, 23(3). March 2011.

[LLCC05] Lu J., Ling T.W., Chan C.Y., Chen T., «From Region Encoding To Extended Dewey : On Efficient Processing of XML Twig Pattern Matching», in *31st International Conference on Very Large Data Bases (VLDB 05), Trondheim, Norway*, pp. 193–204, ACM. 2005.

[LLY+05] Lu J., Ling T.W., Yu T., Li C., Ni W., «Efficient Processing of Ordered XML Twig Pattern», in *16th International on Database and Expert Systems Applications (DEXA 05), Copenhagen, Denmark*, vo-

lume 3588 de *Lecture Notes in Computer Science*, pp. 300–309, Springer. 2005.

[LM01] Li Q., Moon B., «Indexing and Querying XML Data for Regular Path Expressions», in *27th International Conference on Very Large Data Bases (VLDB 01), Roma, Italy*, pp. 361–370, Morgan Kaufmann. 2001.

[LMTS06] Luján-Mora S., Trujillo J., Song I.Y., «A UML profile for multidimensional modeling in data warehouses», *Data Knowl. Eng.*, 59(3) :725–769. 2006.

[LRWZ04] Lakshmanan L.V.S., Ramesh G., Wang W.H., Zhao Z.J., «On Testing Satisfiability of Tree Pattern Queries», in *30th International Conference on Very Large Data Bases (VLDB 04), Toronto, Canada*, pp. 120–131. 2004.

[LS97] Lenz H.J., Shoshani A., «Summarizability in OLAP and Statistical Data Bases», in *9th International Conference on Scientific and Statistical Database Management (SSDBM 97), Olympia, Washington, USA*, pp. 132–143, IEEE Computer Society. 1997.

[LSZY05] Li Z., Sun J., Zhao J., Yu H., «Transforming Non-covering Dimensions in OLAP», in *7th Asia-Pacific Web Conference (APWeb 05), Shanghai, China*, volume 3399 de *Lecture Notes in Computer Science*, pp. 381–393, Springer. 2005.

[Lu06] Lu J., *Efficient Processing of XML Twig Pattern Matching*, Thèse de Doctorat, National University of Singapore. 2006.

[Lu10] Lu J., «Benchmarking Holistic Approaches to XML Tree Pattern Query Processing (Extended Abstract of Invited Talk)», in *15th International Conference on Database Systems for Advanced Applications (DASFAA 10), Tsukuba, Japan*, volume 6193 de *Lecture Notes in Computer Science*, pp. 170–178, Springer. 2010.

[LV03] Lechtenbörger J., Vossen G., «Multidimensional Normal Forms for Data Warehouse Design», *Inf. Syst.*, 28(5) :415–434. 2003.

[LW96] Li C., Wang X.S., «A Data Model for Supporting On-Line Analytical Processing», in *5th International Conference on Information*

*and Knowledge Management (CIKM 96), Rockville, USA*, pp. 81–88, ACM. 1996.

[LWZ06] Lakshmanan L.V.S., Wang W.H., Zhao Z.J., «Answering Tree Pattern Queries Using Views», in *32nd International Conference on Very Large Data Bases (VLDB 06), Seoul, Korea*, pp. 571–582. 2006.

[Mah08] Mahboubi H., *Optimisation de la performance des entrepôts de données XML par fragmentation et répartition*, Thèse de Doctorat, Université Lumière Lyon 2, France. 2008.

[MD10] Mahboubi H., Darmont J., «XWeB : the XML Warehouse Benchmark», in *2nd TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC 10), Singapore*, volume 6417 de *Lecture Notes in Computer Science*, pp. 185–203, Springer. September 2010.

[MHD09] Mahboubi H., Hachicha M., Darmont J., *XML Warehousing and OLAP*, volume IV de *Encyclopedia of Data Warehousing and Mining, Second Edition*, pp. 2109–2116, Hershey, PA, USA : IGI Publishing. 2009.

[MHZ08] Ma H.T., Hao Z.X., Zhu Y., «Checking Satisfiability of Tree Pattern Queries for Active XML Documents», *INFOCOMP Journal of Computer Science*, 7(1) :11–18. 2008.

[MLC09] Min J.K., Lee J., Chung C.W., «An Efficient Encoding and Labeling for Dynamic XML data», *Journal of Systems and Software*, 82(3) :503–515. 2009.

[MLT08] Mazón J.N., Lechtenbörger J., Trujillo J., «Solving Summarizability Problems in Fact-Dimension Relationships for Multidimensional Models», in *11th International ACM Workshop on Data Warehousing and OLAP (DOLAP 08), Napa Valley, California, USA*, pp. 57–64. 2008.

[MLT09] Mazón J.N., Lechtenbörger J., Trujillo J., «A survey on summarizability issues in multidimensional modeling», *Data & Knowledge Engineering*, 68(12) :1452–1469. 2009.

[MMS07] Michiels P., Mihaila G.A., Siméon J., «Put a Tree Pattern in Your Algebra», in *23rd International Conference on Data Engineering (ICDE 07), Istanbul, Turkey*, pp. 246–255, IEEE. 2007.

[MRL+09] Mahboubi H., Ralaivao J.C., Loudcher S., Boussaid O., Bentayeb F., Darmont J., *X-WACoDa : An XML-based approach for Warehousing and Analyzing Complex Data*, pp. 38–54, Hershey, PA, USA : IGI Publishing, in L. Bellatreche, Ed., Data Warehousing Design and Advanced Engineering Applications : Methods for Complex Construction. August 2009.

[MS02] Miklau G., Suciu D., «Containment and Equivalence for an XPath Fragment», in *21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 02), Madison, USA*, pp. 65–76, ACM. 2002.

[MS06] Mansmann S., Scholl M.H., «Extending Visual OLAP for Handling Irregular Dimensional Hierarchies», in *8th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 06), Krakow, Poland*, volume 4081 de *Lecture Notes in Computer Science*, pp. 95–105, Springer. 2006.

[MS07] Mansmann S., Scholl M.H., «Empowering the OLAP Technology to Support Complex Dimension Hierarchies», *International Journal of Data Warehousing and Mining*, 3(4) :31–50. 2007.

[MZ04] Malinowski E., Zimányi E., «OLAP Hierarchies : A Conceptual Perspective», in *16th International Conference on Advanced Information Systems Engineering (CAiSE 04), Riga, Latvia*, volume 3084 de *Lecture Notes in Computer Science*, Springer. 2004.

[MZ06] Malinowski E., Zimányi E., «Hierarchies in a multidimensional model : From conceptual modeling to logical representation», *Data & Knowledge Engineering*, 59(2) :348–377. 2006.

[MZ08] Malinowski E., Zimányi E., *Advanced Data Warehouse Design : From Conventional to Spatial and Temporal Applications*, Berlin, Heidelberg, Germany : Springer. 2008.

[NNNT02] Niemi T., Niinimäki M., Nummenmaa J., Thanisch P., «Constructing an OLAP cube from distributed XML data», in *5th*

*International Workshop on Data Warehousing and OLAP (DOLAP 02), McLean, USA*, pp. 22–27, ACM. 2002.

[NNT01] Niemi T., Nummenmaa J., Thanisch P., «Logical Multidimensional Database Design for Ragged and Unbalanced Aggregation», in *3rd Intl. Workshop on Design and Management of Data Warehouses (DMDW 01), Interlaken, Switzerland*, volume 39 de *CEUR Workshop Proceedings*, p. 7, CEUR-WS.org. 2001.

[NNT03] Niemi T., Nummenmaa J., Thanisch P., «Normalising OLAP cubes for controlling sparsity», *Data Knowl. Eng.*, 46(3) :317–343. 2003.

[NRDR05] Nassis V., Rajagopalapillai R., Dillon T.S., Rahayu J.W., «Conceptual and Systematic Design Approach for XML Document Warehouses», *International Journal of Data Warehousing and Mining*, 1(3) :63–87. 2005.

[NS03] Neven F., Schwentick T., «XPath Containment in the Presence of Disjunction, DTDs, and Variables», in *9th International Conference on Database Theory (ICDT 03), Siena, Italy*, volume 2572 de *Lecture Notes in Computer Science*, pp. 312–326, Springer. 2003.

[PAKC+03] Paparizos S., Al-Khalifa S., Chapman A., Jagadish H.V., Lakshmanan L.V.S., Nierman A., Patel J.M., Srivastava D., Wiwatwattana N., Wu Y., Yu C., «TIMBER : A Native System for Querying XML», in *22th ACM SIGMOD International Conference on Management of Data (SIGMOD 03), San Diego, USA*, p. 672, ACM. 2003.

[PAKJ+02] Paparizos S., Al-Khalifa S., Jagadish H., Nierman A., Wu Y., «A physical algebra for XML», Technical report, University of Michigan, USA. 2002.

[PHS05] Park B.K., Han H., Song I.Y., «XML-OLAP : A Multidimensional Analysis Framework for XML Warehouses», in *7th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 05), Copenhagen, Denmark*, volume 3589 de *LNCS*, pp. 32–42, Springer. 2005.

[PJ05] Paparizos S., Jagadish H.V., «Pattern Tree Algebras : Sets or Sequences ?», in *31st International Conference on Very Large Data Bases (VLDB 05), Trondheim, Norway*, pp. 349–360, ACM. 2005.

[PJD99] Pedersen T.B., Jensen C.S., Dyreson C.E., «Extending Practical Pre-Aggregation in On-Line Analytical Processing», in *25th International Conference on Very Large Data Bases (VLDB 99), Edinburgh, Scotland, UK*, pp. 663–674, Morgan Kaufmann. 1999.

[PJD01] Pedersen T.B., Jensen C.S., Dyreson C.E., «A foundation for capturing and querying complex multidimensional data», *Inf. Syst.*, 26(5) :383–423. 2001.

[Pok02] Pokorný J., «XML Data Warehouse : Modelling and Querying», in *5th International Baltic Conference (Baltic DB&IS 02), Tallin, Estonia*, pp. 267–280, Institute of Cybernetics at Tallin Technical University. 2002.

[PP03] Pedersen D., Pedersen T.B., «Achieving Adaptivity for OLAP-XML Federations», in *6th International ACM Workshop on Data Warehousing and OLAP (DOLAP 03), New Orleans, USA*, pp. 25–32, ACM. 2003.

[PPP04] Pedersen D., Pedersen J., Pedersen T.B., «Integrating XML Data in the TARGIT OLAP System», in *20th International Conference on Data Engineering (ICDE 04), Boston, USA*, pp. 778–781, IEEE Computer Society. 2004.

[PR00] Pourabbas E., Rafanelli M., «Hierarchies and Relative Operators in the OLAP Environment», *SIGMOD Record*, 29(1) :32–37. 2000.

[Pra06] Pradhan S., «An Algebraic Query Model for Effective and Efficient Retrieval of XML Fragments», in *32nd International Conference on Very Large Data Bases (VLDB 06), Seoul, Korea*, pp. 295–306, ACM. 2006.

[PRP02a] Pedersen D., Riis K., Pedersen T.B., «A Powerful and SQL-Compatible Data Model and Query Language for OLAP», in *13th Australasian Database Conference (ADC 02), Monash University,*

*Melbourne, Victoria, Australia*, volume 5 de *CRPIT*, Australian Computer Society. 2002.

[PRP02b] Pedersen D., Riis K., Pedersen T.B., «Query Optimization For OLAP-XML Federations», in *5th International Workshop on Data Warehousing and OLAP (DOLAP 02), McLean, USA*, pp. 57–64, ACM. 2002.

[PWLJ04] Paparizos S., Wu Y., Lakshmanan L.V.S., Jagadish H.V., «Tree Logical Classes for Efficient Evaluation of XQuery», in *23rd ACM SIGMOD International Conference on Management of Data (SIGMOD 04), Paris, France*, pp. 71–82, ACM. 2004.

[RALT06] Rizzi S., Abelló A., Lechtenbörger J., Trujillo J., «Research in Data Warehouse Modeling and Design : Dead or Alive ?», in *9th International ACM Workshop on Data Warehousing and OLAP (DOLAP 06), Arlington, Virginia, USA*, pp. 3–10, ACM. 2006.

[Ram02] Ramanan P., «Efficient Algorithms for Minimizing Tree Pattern Queries», in *2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 02), Madison, Wisconsin, USA*, pp. 299–309, ACM. 2002.

[RCD05] Rajugan R., Chang E., Dillon T.S., «Conceptual Design of an XML FACT Repository for Dispersed XML Document Warehouses and XML Marts», in *5th International Conference on Computer and Information Technology (CIT 05), Shanghai, China*, pp. 141–149, IEEE Computer Society. 2005.

[Riz07] Rizzi S., «Conceptual Modeling Solutions for the Data Warehouse», in Wrembel R., Koncilia C., editors, *Data Warehouses and OLAP : Concepts, Architectures and Solutions*, pp. 1–26, Hershey, USA : IRM Press. 2007.

[RRT05] Rusu L.I., Rahayu J.W., Taniar D., «A Methodology for Building XML Data Warehouses», *International Journal of Data Warehousing and Mining*, 1(2) :23–48. 2005.

[RS90] Rafanelli M., Shoshani A., «STORM : A Statistical Object Representation Model», in *5th International Conference on Statistical*

*and Scientific Database Management (SSDBM 90), Charlotte, NC, USA*, volume 420 de *Lecture Notes in Computer Science*, Springer. 1990.

[RTZ01] Ravat F., Teste O., Zurfluh G., «Modélisation multidimensionnelle des systèmes décisionnels», in *1ères journées Extraction et Gestion des Connaissances (EGC 01), Nantes, France*, pp. 201–212, Hermès. 2001.

[RTZ06a] Ravat F., Teste O., Zurfluh G., «Algèbre OLAP et langage graphique», in *24ème Congrès en Informatique des Organisation et Systèmes d'information de décision (INFORSID 06), Hammamet, Tunisie*, pp. 1039–1054. 2006.

[RTZ06b] Ravat F., Teste O., Zurfluh G., *Constraint-Based Multi-Dimensional Databases*, pp. 323–368, Database Modeling for Industrial Data Management, Idea Group Publishing. 2006.

[SRME01] Song I.Y., Rowen W., Medsker C., Ewen E.F., «An Analysis of Many-to-Many Relationships Between Fact and Dimension Tables in Dimensional Modeling», in *3rd International Workshop on Design and Management of Data Warehouses (DMDW 01), Interlaken, Switzerland*, volume 39 de *CEUR Workshop Proceedings*, p. 6, CEUR-WS.org. 2001.

[SWK+02] Schmidt A., Waas F., Kersten M.L., Carey M.J., Manolescu I., Busse R., «Assessing XML Data Management with XMark», in *VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb (Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web)*, volume 2590 de *Lecture Notes in Computer Science*, pp. 144–145, Springer. 2002.

[TD01] Thomas H., Datta A., «A Conceptual Model and Algebra for On-Line Analytical Processing in Decision Support Databases», *Information Systems Research*, 12(1) :83–102. 2001.

[Tes09] Teste O., *Modélisation et manipulation des systèmes OLAP : de l'intégration des documents à l'usager*, Mémoire d'HDR, Université Paul Sabatier - Toulouse III, France. 2009.

[TH04] Tatarinov I., Halevy A.Y., «Efficient Query Reformulation in Peer-Data Management Systems», in *2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 04), Paris, France*, pp. 539–550, ACM. 2004.

[Tor03] Torlone R., «Conceptual Multidimensional Models», in Rafanelli M., editor, *Multidimensional Databases : Problems and Solutions*, pp. 69–90, Hershey, USA : IDEA Group Publishing. 2003.

[TPC05a] TPC, *TPC Benchmark DS (Decision Support) Draft Specification revision 32*, Transaction Processing Performance Council. `http://www.tpc.org`. 2005.

[TPC05b] TPC, *TPC Benchmark H Standard Specification revision 2.3.0*, Transaction Processing Performance Council. `http://www.tpc.org`. 2005.

[TVB+02] Tatarinov I., Viglas S., Beyer K.S., Shanmugasundaram J., Shekita E.J., Zhang C., «Storing and Querying Ordered XML Using a Relational Database System», in *2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 02), Madison, Wisconsin, USA*, pp. 204–215, ACM. 2002.

[TW09] Theodoratos D., Wu X., «Eager Evaluation of Partial Tree-Pattern Queries on XML Streams», in *14th International Conference on Database Systems for Advanced Applications (DASFAA 09), Brisbane, Australia*, volume 5463 de *Lecture Notes in Computer Science*, pp. 241–246, Springer. 2009.

[Ull88] Ullman J.D., *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press. 1988.

[VBR03] Vrdoljak B., Banek M., Rizzi S., «Designing Web Warehouses from XML Schemas», in *5th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 03), Prague, Czech Republic*, volume 2737 de *LNCS*, pp. 89–98, Springer. 2003.

[WJLS07] Wiwatwattana N., Jagadish H.V., Lakshmanan L.V.S., Srivastava D., «X^3 : A Cube Operator for XML OLAP», in *23rd International Conference on Data Engineering (ICDE 07), Istanbul, Turkey*, pp. 916–925. 2007.

[WLH04] Wu X., Lee M.L., Hsu W., «A Prime Number Labeling Scheme for Dynamic Ordered XML Trees», in *20th International Conference on Data Engineering (ICDE 04), Boston, MA, USA*, pp. 66–78, IEEE Computer Society. 2004.

[WLHG05] Wang H., Li J., He Z., Gao H., «OLAP for XML Data», in *1st International Conference on Computer and Information Technology (CIT 05), Shanghai, China*, pp. 233–237. 2005.

[Woo03] Wood P.T., «Containment for XPath Fragments under DTD Constraints», in *9th International Conference on Database Theory (ICDT 03), Siena, Italy*, volume 2572 de *Lecture Notes in Computer Science*, pp. 297–311, Springer. 2003.

[WWL09] Wang J., Wang K., Li J., «Finding Irredundant Contained Rewritings of Tree Pattern Queries Using Views», in *Advances in Data and Web Management, Joint International Conferences (APWeb/WAIM 09), Suzhou, China*, pp. 113–125. 2009.

[XLWB09] Xu L., Ling T.W., Wu H., Bao Z., «DDE : From Dewey to a Fully Dynamic XML Labeling Scheme», in *2009 ACM SIGMOD International Conference on Management of Data (SIGMOD 09), Providence, Rhode Island, USA*, pp. 719–730, ACM. 2009.

[Xyl01] Xyleme L., «Xyleme : A Dynamic Warehouse for XML Data of the Web», in *International Database Engineering & Applications Symposium (IDEAS 01), Grenoble, France*, pp. 3–7, IEEE Computer Society. 2001.

[YASU01] Yoshikawa M., Amagasa T., Shimura T., Uemura S., «XRel : A Path-based Approach to Storage and Retrieval of XML Documents Using Relational Databases», *ACM Transactions on Internet Technology*, 1(1) :110–141. 2001.

[YC08] Yun J.H., Chung C.W., «Dynamic interval-based labeling scheme for efficient XML query and update processing», *Journal of Systems and Software*, 81(1) :56–70. 2008.

[YLH03] Yang L.H., Lee M.L., Hsu W., «Efficient Mining of XML Query Patterns for Caching», in *29th International Conference on Very Large Data Bases (VLDB 03), Berlin, Germany*, pp. 69–80. 2003.

[YLHA03] Yang L.H., Lee M.L., Hsu W., Acharya S., «Mining Frequent Query Patterns from XML Queries», in *8th International Conference on Database Systems for Advanced Applications (DASFAA 03), Kyoto, Japan*, pp. 355–362, IEEE Computer Society. 2003.

[YOK04] Yao B.B., Ozsu M.T., Khandelwal N., «XBench Benchmark and Performance Testing of XML DBMSs», in *20th International Conference on Data Engineering (ICDE 04), Boston, MA, USA*, pp. 621–632, IEEE Computer Society. 2004.

[ZLBT03] Zhang J., Ling T.W., Bruckner R.M., Tjoa A.M., «Building XML Data Warehouse Based on Frequent Patterns in User Queries», in *5th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 03), Prague, Czech Republic*, volume 2737 de *Lecture Notes in Computer Science*, pp. 99–108, Springer. 2003.

[ZPR02] Zhang X., Pielech B., Rundensteiner E.A., «Honey, I Shrunk the XQuery ! - An XML Algebra Optimization Approach», in *4th International Workshop on Web Information and Data Management (WIDM 02), McLean, USA*, pp. 15–22, ACM. 2002.

[ZWLZ05] Zhang J., Wang W., Liu H., Zhang S., «X-Warehouse : Building Query Pattern-driven Data», in *14th international conference on World Wide Web (WWW 05), Chiba, Japan*, pp. 896–897. 2005.

[ZY04] Zhang M., Yao J., «XML Algebras for Data Mining», in *Data Mining and Knowledge Discovery : Theory, Tools, and Technology VI, Orlando, USA*, volume 5433 de *SPIE proceedings series*, pp. 209–217, SPIE. 2004.